

Object Pascal Handbook: Delphi 11 Alexandria Edition

by Marco Cantù

Copyright © 1995-2021 Marco Cantù, Piacenza, Italy. World rights reserved

Korean translation copyright © 2024 by DevGear Co., Ltd

All rights reserved.

This Korean edition was published by agreement with Marco Cantù.

이 책의 한국어판 저작권은 저작권자와 계약한 데브기어에 있습니다.

저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단전제와 무단복제를 금합니다.

이 책의 예제 코드는 독자가 자유롭게 사용할 수 있습니다. 이 책의 소스 코드는 저작권이 있는 프리웨어이며, 깃허브 [GitHub](#) 프로젝트를 통해 배포됩니다. 경로는 이 책과 이 책의 웹 사이트에 안내 되어있습니다. 저작권에 따라 허가 없이 소스 코드를 인쇄물이나 전자 매체를 통해 재공개 할 수 없습니다. 독자는 코드 자체를 배포, 판매 또는 상업적으로 악용하지 않는 경우, 자신의 애플리케이션에서 이 코드를 사용할 수 있다는 제한된 권한이 부여됩니다.

최선을 다해 이 책을 준비했지만, 내용의 완전성 또는 정확성에 대해 어떠한 진술이나 보증도 하지 않으며, 이 책으로 인해 성능, 상품성, 특정 목적에 대한 적합성 등 직간접적으로 발생되거나 발생되었다고 주장되는 모든 종류의 손실 또는 손해에 대해 어떠한 책임도 지지 않습니다.

오브젝트 파스칼 핸드북: 델파이 11 알렉산드리아 에디션

전자책 발행 2024년 5월 26일

지은이 마르코 칸투

옮긴이 박범용, 황경운

편집/검토 김종진

펴낸곳 데브기어 출판부

등록 2009년 04월 06일

주소 서울특별시 서초구 사평대로 359, 3층, 주식회사 데브기어

전화 02-595-4288 | 팩스 02-536-4288

이 책의 한국어판 저작권은 저작권자와 계약한 데브기어에 있습니다.

저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단전제와 무단복제를 금합니다.

값은 뒤표지에 있습니다.

전자책 ISBN 978-89-962516-8-2

홈페이지 www.devgear.co.kr/

페이스북 facebook.com/devgear

유튜브 youtube.com/user/embarcaderoKR

전자우편 ask@devgear.co.kr

데브기어 출판부는 독자의 의견이나 수정 요청을 고마운 마음으로 받습니다.

마르코 칸투

오브젝트 파스칼 핸드북
델파이 11 알렉산드리아 에디션

오브젝트 파스칼 프로그래밍 언어 완전 가이드

델파이 11 알렉산드리아용

최초 출간: 피아첸차(이탈리아), 2015년 7월
델파이 10.4 에디션 출간: 피아첸차(이탈리아), 2021년 3월
델파이 11 알렉산드리아 에디션 출간: 피아첸차(이탈리아), 2021년 11월

begin

내 삶을 기대 이상으로 만들어 준
우리 가족, 라파엘라, 베니, 자코포에게
내 모든 사랑과 큰 감사를 전한다.

강력함과 단순함, 표현력과 가독성, 학습용과 전문 개발자용 양쪽 모두 똑같이 훌륭함!
이 수식어들은 오늘날 오브젝트 파스칼의 특징이다. 또한 역사가 오래되었고, 지금도
활기차고, 미래가 찬란한 언어다.

오브젝트 파스칼은 여러 가지 얼굴을 가진 언어다. 객체 지향 프로그래밍이라는 강력한
파워를 가지고 있으면서도 제네릭^{generic} 프로그래밍과 (애트리뷰트^{attribute} 등) 동적 구조를
수준 높게 지원한다. 또한 절차적^{procedural} 프로그래밍이라는 훨씬 더 오래된 스타일도
버리지 않고 지원하고 있다. 그 도구는 모든 분야에서 사용할 수 있다. 모바일 시대를
수용하는 컴파일러와 개발 도구를 갖추고 있다. 그 언어는 미래를 향한 준비가 되어
있으면서도 과거에 단단한 뿌리를 두고 있다.

오브젝트 파스칼 언어는 어디에 사용될까? 데스크탑, 클라이언트-서버 애플리케이션,
대규모 웹 서버 모듈, 미들웨어, 사무 자동화, 최신 휴대폰용 및 태블릿용 앱, 산업
자동화 시스템, 인터넷 가상 전화 네트워크... 등등 다양한 분야에서 사용되고 있다.
단지 이런 것을 만들 수 있을 것이라고 말하는 것이 아니다. 이 언어로 만들어져서
지금 바로 이 순간 실제로 세상에서 작동되고 있는 것들을 말하고 있는 것이다.

현대 오브젝트 파스칼 언어의 핵심은 1995년에 정의된 것에서 비롯되었는데, 그 해는
프로그래밍 언어에 있어 매우 중요한 해로서 자바^{Java}와 자바스크립트^{JavaScript}가 발명된
해이기도 하다. 오브젝트 파스칼 언어는 파스칼 조상으로부터 진화되어왔고, 1995년에
크게 도약했지만, 거기서 멈추지 않고, 지금도 핵심 기능이 지속적으로 개선되고 있다.
데스크탑용과 모바일용 컴파일러들이 델파이와 RAD 스튜디오 개발 환경 안에 있다.
이는 엠바카데로 테크놀로지스에서 해낸 작품이다.

오늘날의 언어에 관한 책

파스칼 언어는 그 역할이 변화하고, 해를 거듭하며 확장되어 왔다는 점과 새 개발자들이 지금 유입되고 있는 사실을 보면서, 지금 해야 할 중요한 일은 오브젝트 파스칼 언어를 완전하게 다루는 책을 쓰는 것이라는 생각했다. 이 책의 목표는 모두를 위한 핸드북을 제공하는 것이다. 신입 개발자나 다른 유사한 언어에서 넘어온 개발자뿐만 아니라, 예전에 파스칼 계열 언어를 사용했는데 이 언어가 최근 어떻게 변했는지를 배우고 싶은 개발자에게 좋은 설명서가 되기를 바란다.

이 책은 분명 초보자에게 필요한 기초들이 들어있다. 하지만, 이 언어가 광범위하게 변해왔기 때문에 기존 사용자도 새로운 것을 이 책 곳곳에서 발견할 것이다.

이 책은 오브젝트 파스칼 언어의 역사를 간략하게 다루고 있는 부록 외에는, 지금의 언어를 기준으로 설명한다. 참고로 오브젝트 파스칼의 핵심들은 대부분 그것들이 맨 처음 구현되었던 1995년 델파이의 초기 버전이나 지금이나 크게 달라지지 않았다.

책을 읽으면서 느끼겠지만, 이 언어는 그 동안 전혀 정체되지 않았고 오히려 상당히 빠른 속도로 발전해 왔다.

과거에 저술했던 다른 책들은, 고전적인 파스칼을 먼저 다루고 나서, 그 후에 도입된 확장 기능들을 시간 순서대로 설명하는 *연대기적* 접근 방식을 따랐다. 하지만, 이 책에서는 보다 *논리적* 접근 방식을 따른다. 즉, 각 주제를 중심으로 오브젝트 파스칼이 현재 어떻게 되어 있고, 가장 잘 사용하려면 어떻게 하면 되는지를 설명한다. 시간을 따라가며 서술하는 방식이 아니다.

예를 들어, 기본^{native} 데이터 타입들은 원래 파스칼 언어부터 있던 것이지만, 지금은 최근에 도입된 메서드^{method} 같은 기능(내장식^{intrinsic} 타입^{type} 헬퍼^{helper} 덕분임)들도 들어 있다. 따라서 이 기능은 데이터 타입을 다루는 장인 2장에서 설명한다. 한편, 사용자 정의 타입을 개발자가 직접 확장하는 방법은 더 뒤에 있는 장에서 설명한다.

즉, 이 책은 지금의 오브젝트 파스칼을 가지고, 기본부터 가르친다. 역사적 관점으로 설명하는 경우는 많지 않다. 과거에 이 언어를 사용해 본 개발자라도 책 내용 전체를 훑어보면서 새로운 특징들을 찾아보면 좋다. 뒤에 있는 장들만 집중하지 않기 바란다.

실습을 통한 학습

이 책은 핵심 개념을 설명하고, 곧 이어 간단한 예제를 제시한다. 독자는 예제를 실행, 실험, 확장해 보고 설명된 개념을 더 잘 이해하고 자기 것으로 만들기 바란다. 이것은 이 언어가 어떻게 되어 있는지에 대해 이론적 해설과 특이 사례들을 모두 나열하는 참고용 매뉴얼이 아니다. 이 책은 정확한 설명을 지향하면서도, 그 초점은 언어를 잘 쓸 수 있도록 알려주는 실용적인 단계별 가이드 제공에 맞춰져 있다. 예제는 대체로 매우 간단하다. 한 번에 한 가지 기능에 집중하기가 각 예제의 목표이기 때문이다.

전체 소스 코드는 온라인 코드 저장소^{repository}인 깃허브^{GitHub}에 있으므로, 단일 파일로 다운로드하거나, 리포지토리를 복제^{clone}하거나, 온라인으로 둘러보고 특정 프로젝트관련 코드만 다운로드할 수도 있다. 리포지토리를 복제하면, 이 리포지토리에 추가 또는 변경이 생겼을 때, 쉽게 파악하고 업데이트할 수 있다. 깃허브의 위치는 델파이 10.4 시드니 버전을 기준으로 출간된 지난번 에디션에서 사용한 링크와 동일하다.

■ <https://github.com/MarcoDelphiBooks/ObjectPascalHandbook104>

예제 코드를 컴파일하고 테스트하기 위해서는 델파이 최신 버전이 있어야 한다 (모든 예제를 실행하려면 10.4 이상이어야 한다. 하지만, 예제 대부분이 10.x 및 11 버전에서도 작동할 것이다).

델파이 라이선스가 없다면, 평가판 버전을 쓸 수 있다. 대체로 30일 동안 컴파일러와 IDE를 무료로 사용할 수 있다. 또한 델파이 커뮤니티 에디션도 있다 (출간일: 2023년 5월 현재, 11.3 버전이 제공되고 있음). 소프트웨어 개발 수입이 아예 없거나 약관에 명시된 기준보다 적다면 누구나 커뮤니티 에디션을 무료로 사용할 수 있다.

고마움

여느 책이 그렇듯이, 이 책 역시 도움을 준 분들이 많다. 너무 많아 일일이 열거할 수 없다. 초판 작업 대부분은 편집자인 피터 우드가 함께 했다. 그는 시시각각 변하는 내 스케줄을 계속 견뎌주고 내 기술 영어를 향상시켜 지금의 이 책이 되도록 해주었다.

초판이 출간된 후, 독자이자 개발자인 Andreas Toth가 방대한 의견을 보내주었다. 그 후, 그는 2판의 편집자로 참여하여 영어 문법, 책의 일관성, 오브젝트 파스칼 코딩 스타일 등 책의 내용을 종합적으로 검토해주었다. 이 새 판은 그의 공로가 매우 크다.

이 새 판은 다른 델파이 전문가(대부분 엠바카데로 MVP) 몇몇의 검토 역시 거쳤다. 특히 François Piette는 100 개가 넘는 수정 사항과 제안을 보내주었다. 최종 내용에 반영된 좋은 의견들이었다.

나는 현재 엠바카데로 테크놀로지스에서 제품 관리자를 맡고 있기 때문에, 동료들, R&D 팀원들 모두로부터 수많은 도움을 받았다. 델파이와 관련 기술들을 더 잘 이해할 수 있게 된 것도 이 회사에서 수많은 대화, 회의, 이메일을 통해서 통찰력을 얻을 기회가 많았기 때문이다.

일일이 언급하기 힘들다고 했지만, 이 책의 초판에 직접 영향을 준 세 사람만 꼽으면, 개발자 관계 부문의 David I, RAD 제품 관리 책임자였던 John Thomas(JT), 그리고 RAD 아키텍트였던 Allen Bauer를 들 수 있다.

더 최근에는 RAD 스튜디오 제품 관리자인 다른 두 사람 (Sarina DuPont, David Millington), 에반젤리스트인 Jim McKeeth, 그리고 뛰어난 R&D 아키텍트들로 구성된 지금의 엔지니어들과 함께 폭넓게 일하고 있다.

엠바카데로 외부에 있는 많은 사람들도 계속해서 중요한 연락을 취하고 때로는 직접 의견을 제공해주었다. 이탈리아에 있는 많은 델파이 전문가, 수많은 고객, 엠바카데로 영업과 기술 파트너, 델파이 커뮤니티 회원, MVP, 게다가 자주 만나는 다른 언어와 도구를 사용하는 개발자에 이르기까지 다양한 사람들이 도움을 주었다.

내가 엠바카데로에 합류하기 전에, 나와 많은 시간을 함께 보낸 사람들 중에 한 명을 꼽자면 Cary Jensen이 있다. 그와 나는 유럽과 미국에서 델파이 '개발자의 날' 행사 몇 차례를 함께 주최했었다.

마지막으로 내 여행 일정, 야간 회의, 주말 집필 등을 함께 참아 준 가족들에게도 큰 감사를 전한다. Lella, Benny, Jacopo 다시 한번 고맙다.

글쓴이 소개

나는 지난 25년 대부분을 오브젝트 파스칼 언어로 소프트웨어를 개발하는 것에 관한 글을 쓰고, 가르치고, 컨설팅을 했다. 베스트셀러인 '마스터링 델파이' 시리즈를 집필했고, 그 후에는 개발 도구에 대한 핸드북(델파이 2007 버전에서부터 델파이 XE까지 다양한 버전에 대하여) 여러 권을 직접 출판했다.

많은 대륙에서 열린 수많은 프로그래밍 컨퍼런스에서 발표를 했고, 컨퍼런스, 델파이 개발자 행사, 기업 주최 강좌, 온라인 웨비나, 코드레이지 [CodeRage](#) 컨퍼런스에서 개발자 수천 명을 대상으로 강의했다.

수년간 독립 컨설턴트와 강사로 활동하다가, 2013년부터 갑작스럽게 경력을 바꾸어, 이 개발 도구를 제작하고 판매하는 회사인 엠바카데로 테크놀로지스에서 델파이 제품 매니저를 맡았고 지금은 RAD 스튜디오 제품 매니저로 일하고 있다.

끝으로, 지금 이탈리아에 살고 있고, 캘리포니아로 출퇴근하며(최근에는 조금 줄었음), 사랑스러운 아내와 멋진 두 자녀가 있고, 프로그래밍을 가능한 많이 즐기고 있다는 점만 덧붙이겠다.

이 개정판을 집필하면서 내가 즐거웠던 것처럼 독자도 이 책을 즐겁게 읽길 바란다. 자세한 내용은 다음 웹 사이트와 소셜 미디어 채널을 참조하면 된다:

<https://www.marcocantu.com/objectpascalhandbook>
<https://blog.marcocantu.com>
<https://twitter.com/marcocantu>

차례

begin	4
오늘날의 언어에 관한 책	5
실습을 통한 학습	5
고마움	6
글쓴이 소개	7
차례	8
파트 I 기초	19
파트 I 요약	20
01: 파스칼Pascal로 코딩하기	21
코드Code부터 시작하자	21
첫 콘솔Console 애플리케이션	22
첫 비주얼Visual 애플리케이션	23
구문Syntax 및 코딩 스타일Coding Style	25
주석Comment	26
주석Comment 및 XML 문서XML Doc	27
심볼 식별자 Symbolic Identifier	28
공백 Whitespace	30
들여쓰기 Indentation	31
구문Syntax 강조 표시Highlighting	32
이 언어의 키워드들 Language Keywords	33
프로그램의 구조Structure	37
유닛과 프로그램의 이름 Unit and Program Names	38
유닛Unit 과 범위Scope	41
프로그램 파일	43
컴파일러 지시어Compiler Directives	44
조건부 정의 Conditional Define	44
컴파일러 버전들	45
인클루드 파일들 Include Files	46
02: 변수variable와 데이터 타입data type	47
변수Variable와 할당Assignment	48
리터럴 값Literal Value	49
할당 문 Assignment Statement	50
할당Assignment과 변환Conversion	51
글로벌global/전역 변수를 초기화하기 Initializing Global Variables	51
로컬Local/지역 변수를 초기화하기 Initializing Local Variables	51
인라인 변수 Inline Variables	52

상수 Constants	54
변수의 수명과 가시성 Lifetime and Visibility of Variables	56
데이터 타입들 Data Types	57
순서 Ordinal 및 숫자 Numeric 타입	57
불리언 Boolean	62
문자들 Characters	62
부동 소수점 타입 Floating Point Types	65
간단한 사용자 정의 User-Defined 데이터 타입	67
명명된 Named 타입과 명명되지 않은 Unnamed 타입	68
타입의 별칭 Type Alias	69
하위범위 타입 Subrange Types	70
열거되는 타입 Enumerated Types	71
세트 타입 Set Types	72
표현식과 연산자 Expressions and Operators	74
연산자 사용하기 Using Operators	74
연산자 및 우선 순위 Operators and Precedence	75
날짜 Date와 시간 Time	77
날짜 시간 헬퍼 Date Time Helper	80
타입 캐스팅 Casting 및 타입 변환 Conversion	80
03: 언어의 문장 language statement	83
단순 문장 Simple Statement과 복합 문장 Compound Statement	84
If 문 The If Statement	85
Case 문 Case Statement	86
For 루프 loop/순환	88
for-in 루프	91
While 문 및 Repeat 문	92
루프 예문들 Examples of Loops	93
Break와 Continue를 사용해 흐름 끊기	95
04: 프로시저와 함수 Procedures and Functions	98
프로시저와 함수 Procedures and Functions	98
포워드 선언 Forward Declarations	101
재귀 함수 A Recursive Function	102
메서드란 무엇인가? What Is a Method?	103
파라미터와 반환값 Parameters and Return Value	104
결과를 가지고 빠져나가기 Exit with a Result	105
참조 파라미터 Reference Parameters	106
상수 파라미터 Constant Parameters	108
함수 오버로딩 Function Overloading	108
오버로드와 모호한 호출 Overloading and Ambiguous Calls	110

기본 파라미터 Default Parameters	112
인라인 처리 Inlining	113
함수의 고급 기능들 Advanced Features of Functions	116
오브젝트 파스칼의 호출 규약들 Object Pascal Calling Conventions	117
프로시저 타입들 Procedural Types	117
외부 함수 선언 External Functions Declarations	120
05: 배열과 레코드 Arrays and Records	123
배열 데이터 타입 Array Data Types	123
정적 배열 Static Arrays	124
배열 크기 및 경계 Array Size and Boundaries	125
다-차원 정적 배열 Multi-Dimensional Static Arrays	126
동적 배열 Dynamic Arrays	127
오픈 배열 파라미터 Open Array Parameters	131
레코드 데이터 타입 Record Data Types	134
레코드 배열 사용 Using Arrays of Records	136
배리언트 레코드 Variant(변형) Records	137
필드 정렬 Fields Alignments	138
with 문은 어떨까? What About the With Statement?	139
메서드를 가지는 레코드 Records with Methods	141
Self: 레코드 뒤에 숨겨진 마법 Self: The Magic Behind Records	143
레코드 초기화하기 Initializing Records	144
레코드와 생성자 Records and Constructors	145
연산자에게 새 기반을 제공하기 Operators Gain New Ground	146
연산자와 사용자 정의 매니지드 레코드 Operators and Custom Managed Record	150
배리언트 Variants(변형)	154
배리언트는 타입을 갖지 않는다 Variants Have No Type	155
배리언트를 자세히 보기 Variants in Depth	156
배리언트는 느리다 Variants Are Slow	157
포인터는 어떤가? What About Pointers?	158
파일 타입을 알고 있나요? File Types, Anyone?	161
06: 문자열 String에 관한 모든 것	163
유니코드 Unicode: 전 세계를 위한 문자 집합	164
문자의 과거: ASCII부터 ISO 인코딩 Encoding까지	164
유니코드 코드 포인트 Code Point들과 시각적 문자 Grapheme들	165
코드 포인트에서 바이트(UTF)로 변환하기	166
바이트 순서 표식 (BOM) Byte Order Mark	167
유니코드 살펴보기	168
Char ^{문자} 타입을 다시 살펴보기	171
Character 유닛을 사용하여 유니코드를 연산하기	171

유니코드 문자 리터럴 Literal	173
1-바이트 byte 문자는 어떻게?	175
문자열 String 데이터 타입	175
문자열 String을 파라미터 Parameter로 전달하기	178
[] 사용 및 문자열 String에서 문자 카운팅 모드 Character Counting Mode	179
문자열 String 합치기 concatenating	181
String 헬퍼 Helper 연산	183
RTL에 있는 문자열 함수들 더 보기	186
문자열에 서식을 반영하기 Formatting Strings	186
문자열 String의 내부 구조 Structure	189
문자열 String 메모리 안쪽을 보기	190
문자열 String과 인코딩 Encoding	192
문자열용 기타 타입들	195
UCS4String 타입	195
이전의 문자열 타입들	196
파트 II OOP와 오브젝트 파스칼	197
파트 II 요약	198
07: 오브젝트 Objects	199
클래스와 오브젝트 소개 Introducing Classes and Objects	199
클래스 정의 The Definition of a Class	200
다른 OOP 언어들 안에 있는 클래스 Classes in Other OOP Languages	201
클래스 메서드 The Class Methods	202
오브젝트 생성하기 Creating an Object	203
오브젝트 참조 모델 The Object Reference Model	204
오브젝트 폐기하기 Disposing of Objects	205
“Nil”이란 무엇인가? What is “Nil”?	206
레코드와 클래스를 메모리 관점에서 비교 Records vs. Classes in Memory	206
비공개, 보호, 공개 Private, Protected, and Public	207
비공개 데이터 예시 An Example of Private Data	208
캡슐화와 폼 Encapsulation and Forms	210
Self 식별자 The Self Identifier	213
컴포넌트를 동적으로 생성하기 Creating Components Dynamically	214
생성자 Constructors	215
로컬 클래스 데이터를 관리하기 (생성자와 소멸자 사용)	217
오버로드되는 메서드와 오버로드되는 생성자 Overloaded Methods and Constructors	218
TDate 클래스 전체 The Complete TDate Class	220
중첩된 타입과 중첩된 상수 Nested Types and Nested Constants	223
08: 상속 Inheritance	226
기존 타입으로부터 상속하기 Inheriting from Existing Types	226

공통 기반 클래스 A Common Base Class	229
보호된 필드와 캡슐화 Protected Fields and Encapsulation	229
"보호된 멤버 해킹" 사용하기 Using the "Protected Hack"	230
상속부터 다형성까지 From Inheritance to Polymorphism	232
상속과 타입 호환성 Inheritance and Type Compatibility	232
나중에 바인딩하기와 다형성 Late Binding and Polymorphism	234
메서드를 오버라이딩, 다시 정의하기, 다시 도입하기 Overriding, Redefining, Reintroducing	236
상속과 생성자 Inheritance and Constructors	238
가상 메서드와 동적 메서드 Virtual versus Dynamic Methods	239
메서드 추상화와 클래스 추상화 Abstracting Methods and Classes	240
추상 메서드 Abstract Methods	241
봉인된 클래스와 최종 메서드 Sealed Classes and Final Methods	243
안전한 타입 캐스트 연산자 Safe TypeCast Operators	243
비주얼 폼 상속 Visual Form Inheritance	246
기본 클래스로부터 상속하기 Inheriting From a Base Form	247
09: 예외 다루기 Handling Exceptions	250
Try-Except 블록 Try-Except Blocks	251
예외의 계층 구조 The Exceptions Hierarchy	253
예외 발생시키기 Raising Exceptions	255
예외와 스택 Exceptions and the Stack	256
Finally 블록 The Finally Block	257
Finally 블록을 사용해 커서 회복하기 Restore the Cursor with a Finally Block	258
매니지드 레코드를 사용해 커서 회복하기 Restore the Cursor with a Managed Record	259
실제 세상에 있는 예외 Exceptions in the Real World	260
글로벌(전역) 수준에서 예외 다루기 Global Exceptions Handling	260
예외와 생성자 Exceptions and Constructors	261
예외의 고급 기능들 Advanced Features of Exceptions	263
중첩된 예외와 내부 예외 메커니즘 Nested Exceptions and the Inner Exception Mechanism	264
예외 가로채기 Intercepting an Exception	267
10: 프로퍼티와 이벤트 Properties and Events	269
프로퍼티 정의하기 Defining Properties	269
다른 프로그래밍 언어들과 프로퍼티를 비교	271
프로퍼티는 캡슐화 Encapsulation 를 구현한다	272
프로퍼티를 위한 코드 완성 Code Completion for Properties	273
폼 form에 프로퍼티 추가하기	273
TDate 클래스에 프로퍼티를 추가하기	275
배열 프로퍼티 사용하기 Using Array Properties	277
프로퍼티를 참조로 설정하기 Setting Properties by Reference	278
published 접근 지정자 Setting Properties by Reference	280

디자인할 때의 프로퍼티 Design-Time Properties	281
published와 폼 Published and Forms	282
자동 RTTI Automatic RTTI	283
이벤트 기반 프로그래밍 Event-Driven Programming	284
메서드 포인터 Method pointers	285
델리게이션이라는 개념 The Concept of Delegation	286
이벤트는 프로퍼티다 Events are Properties	289
TDate 클래스에 이벤트 추가하기 Adding an Event to the TDate Class	290
TDate 컴포넌트를 만들기 Creating a TDate Component	292
클래스 안에 열거형 Enumeration 지원 구현하기	295
RAD와 OOP 섞어 쓰기와 관련된 15가지 팁.....	297
팁 1: 폼은 클래스다 A form is a class	298
팁 2: 컴포넌트에 이름을 붙여라 Name Components	298
팁 3: 이벤트에 이름을 붙여라 Name Events	298
팁 4: 폼 메서드를 사용하라 Use Form Methods	299
팁 5: 폼 생성자를 추가하라 Add Form Constructors	299
팁 6: 글로벌/global/전역 변수는 가급적 사용을 피하라 Avoid Global Variables	299
팁 7: 인스턴스 변수 Instance Variable를 그 구현 Implementation 안에서 사용하지 말라.....	300
팁 8: 폼의 변수는 가급적 쓰지 말라 Seldom Use a Form's Variable	300
팁 9: 글로벌/global/전역 변수인 Form1을 제거하라 Remove the Global Form1 Variable	300
팁 10: 폼 프로퍼티를 추가하라 Add Form Properties	301
팁 11: 컴포넌트 프로퍼티를 노출하라 Expose Component Properties	301
팁 12: 필요하다면 배열 프로퍼티를 쓰라 Use Array Properties when Needed	301
팁 13: 프로퍼티의 시작 동작 Starting Operations in Properties	301
팁 14: 컴포넌트를 숨겨라 Hide Components	302
팁 15: OOP 폼 마법사를 사용하라 Use an OOP Form Wizard	303
팁 마무리	303
11: 인터페이스 Interfaces	304
인터페이스 사용하기 Using Interfaces	305
인터페이스 선언하기 Declaring an Interface	305
인터페이스 구현하기 Implementing an Interface	306
인터페이스와 참조 카운팅 Interfaces and Reference Counting	308
참조 혼용으로 인한 오류 Errors in Mixing References	309
약한 Weak 인터페이스 참조와 안전하지 않은 Unsafe 인터페이스 참조	311
고급 인터페이스 기술들 Advanced Interface Techniques	313
인터페이스 프로퍼티 Interface Properties	313
인터페이스 델리게이션 Interface Delegation	314
다중 인터페이스와 메서드 별칭 Multiple Interfaces and Methods Aliases	316
인터페이스의 다형성 Interface Polymorphism	317

인터페이스 참조로부터 오브젝트 추출하기	Extracting Objects from Interface References	319
인터페이스로 어댑터 패턴(Adapter Pattern)을 구현하기		320
12: 클래스 조작하기	Manipulating Classes	323
클래스 메서드와 클래스 데이터	Class Methods and Class Data	323
클래스 데이터	Class Data	324
가상 클래스 메서드와 숨겨진 Self 파라미터	Virtual Class Methods and the Hidden Self Parameter	325
클래스의 정적 메서드	Class Static Methods	325
클래스 프로퍼티	Class Properties	327
인스턴스 카운터를 가지는 클래스	A Class with an Instance Counter	327
클래스 생성자(와 소멸자)	Class Constructors (and Destructors)	329
RTL 안에 있는 클래스 생성자들	Class Constructors in the RTL	330
싱글톤 패턴 구현하기	Implementing the Singleton Pattern	331
클래스 참조	Class Reference	331
RTL 안에 있는 클래스 참조들	Class References in the RTL	333
클래스 참조를 사용하여 컴포넌트 만들기	Creating Components Using Class References	333
클래스 헬퍼와 레코드 헬퍼	Class and Record Helpers	336
클래스 헬퍼	Class Helpers	336
클래스 헬퍼와 상속	Class Helpers and Inheritance	339
클래스 헬퍼에 컨트롤 열거형 추가하기	Adding Control Enumeration with a Class Helper	340
내장된 타입을 위한 레코드 헬퍼	Record Helpers for Intrinsic Types	342
타입 별칭에 대한 헬퍼	Helpers for Type Aliases	344
13: 오브젝트와 메모리	Objects and Memory	346
전역 데이터, 스택, 힙	Global Data, the Stack, and the Heap	347
전역 메모리	The Global Memory	347
스택	The Stack	348
힙	The Heap	349
오브젝트 참조 모델	The Object Reference Model	349
오브젝트를 파라미터로 전달하기	Passing Objects as Parameters	350
메모리 관리 팁들	Memory Management Tips	351
생성한 오브젝트 소멸하기	Destroying Objects You Create	352
오브젝트를 한 번만 소멸하기	Destroying Objects Only Once	353
메모리 관리와 인터페이스	Memory Management and Interfaces	355
Weak 참조에 대한 더 많은 것들	More on Weak References	355
Unsafe 애트리뷰트	The Unsafe Attribute	359
메모리를 추적하고 확인하기	Tracking and Checking Memory	359
메모리 상태	Memory Status	360
FastMM4		360
누수 추적하기와 다른 전역 설정들	Tracking Leaks and Other Global Settings	361
완전한 FastMM4에서의 버퍼 오버런 탐지	Buffer Overruns in the Full FastMM4	362

윈도우가 아닌 플랫폼에서의 메모리 관리	Memory Management on Platforms Other than Windows	364
클래스별로 할당을 추적하기	Tracking Per-Class Allocations	365
견고한 애플리케이션 작성하기	Writing Robust Applications	365
생성자, 소멸자, 그리고 예외	Constructors, Destructors, and Exceptions	365
중첩된 Finally 블록들	Nested Finally Blocks	367
동적으로 타입을 확인하기	Dynamic Type Checking	368
이 포인터는 오브젝트 참조인가?	Is this Pointer an Object Reference?	369
파트 III 고급 기능들		372
파트 III 요약		373
14: 제네릭스	Generics	374
제네릭 키-값 쌍	Generic Key-Value Pairs	375
인라인 변수와 제네릭 타입 추론	Inline Variables and Generics Type Inference	378
제네릭의 타입 규칙	Type Rules on Generics	378
오브젝트 파스칼의 제네릭	Generics in Object Pascal	379
제네릭 타입 호환성 규칙	Generic Types Compatibility Rules	380
표준 클래스용 제네릭 메서드	Generic Methods for Standard Classes	381
제네릭 타입의 인스턴스화	Generic Type Instantiation	383
제네릭 타입 함수들	Generic Type Functions	384
제네릭 클래스를 위한 클래스 생성자	Class Constructors for Generic Classes	387
제네릭 제약들	Generic Constraints	388
클래스 제약들	Class Constraints	389
특정 클래스 제약들	Specific Class Constraints	391
인터페이스 제약들	Interface Constraints	391
인터페이스 참조 vs 제네릭 인터페이스 제약	Interface References vs Generic Interface Constraints	393
기본 생성자 제약	Default Constructor Constraint	394
제약에 대한 요약 및 조합해서 사용하기	Constraints Summary and Combining them	396
미리 정의된 제네릭 컨테이너들	Predefined Generic Containers	396
TList<T> 사용하기	Using TList<T>	397
TList<T> 정렬하기	Sorting a TList<T>	398
익명 메서드를 사용해 정렬하기	Sorting with an Anonymous Method	400
오브젝트 컨테이너들	Object Containers	401
제네릭 딕셔너리 사용하기	Using a Generic Dictionary	402
딕셔너리 vs 스트링 리스트	Dictionaries vs. String Lists	405
제네릭 인터페이스들	Generic Interfaces	407
미리 정의된 제네릭 인터페이스	Predefined Generic Interfaces	409
오브젝트 파스칼의 스마트 포인터	Smart Pointers in Object Pascal	409
레코드를 스마트 포인터용으로 사용하기	Using Records for Smart Pointers	410
제네릭 매니지드 레코드	Generic Managed Record를 사용해 스마트 포인터 구현하기	411
제네릭 레코드	Generic Record와 인터페이스Interface를 사용해 스마트 포인터 구현하기	413

암시적 변환 추가하기 Adding Implicit Conversion	414
스마트 포인터 사용법 간의 비교 Comparing Smart Pointer Solutions	416
제네릭을 사용한 공변 반환 타입 Covariant Return Types with Generics	416
Animals, Dogs, Cats에 관하여 About Animals, Dogs, and Cats	416
제네릭 결과를 가지는 메서드 A Method with a Generic Result	417
클래스가 다른 파생된 오브젝트 반환하기 Returning a Derived Object of a Different Class	418
15: 익명 메서드 Anonymous Methods	420
익명 메서드의 구문과 의미 Syntax and Semantics of Anonymous Methods	421
익명 메서드 변수 An Anonymous Method Variable	421
익명 메서드 파라미터 An Anonymous Method Parameter	422
로컬 변수 사용하기 Using Local Variables	422
로컬 변수의 수명 확장하기 Extending the Lifetime of Local Variables	423
익명 메서드의 이면 Anonymous Methods Behind the Scenes	425
(잠재적으로) 생략된 괄호 The (Potentially) Missing Parenthesis	425
익명 메서드 구현 Implementation of Anonymous Methods	426
바로 사용할 수 있는 참조 타입들 Ready-To-Use Reference Types	427
현실의 익명 메서드 Anonymous Methods in the Real World	428
익명 이벤트 핸들러 Anonymous Event Handlers	428
익명 메서드의 시간 측정하기 Timing Anonymous Methods	430
쓰레드 동기화 Threads Synchronization	432
오브젝트 파스칼에서의 AJAX AJAX in Object Pascal	434
16: 리플렉션 Reflection과 애트리뷰트 Attribute	438
확장된 RTTI Extended RTTI	439
첫 예제 A First Example	439
컴파일러가 생성하는 정보 Compiler Generated Information	440
약한 타입 링킹과 강한 타입 링킹 Weak- and Strong-Type Linking	442
RTTI 유닛 The RTTI Unit	442
Rtti 유닛 안의 RTTI 클래스들 The RTTI Classes in the Rtti Unit	445
RTTI 오브젝트 수명 관리 RTTI Objects Lifetime Management와 TRttiContext 레코드	445
클래스 정보 표시하기 Displaying Class Information	447
패키지를 위한 RTTI RTTI for Packages	448
TValue의 구조 The TValue Structure	449
TValue를 사용해 프로퍼티를 읽기 Reading a Property with TValue	451
메서드 불러내기 Invoking Methods	452
애트리뷰트 사용하기 Using Attributes	453
애트리뷰트란? What is an Attribute?	453
애트리뷰트 클래스와 애트리뷰트 선언 Attribute Classes and Attribute Declarations	454
애트리뷰트 탐색하기 Browsing Attributes	456
가상 메서드 인터셉터 Virtual Method Interceptors	458

RTTI 사례 연구 RTTI Case Studies	462
ID와 설명을 위한 애트리뷰트들 Attributes for ID and Description	462
XML 스트리밍 XML Streaming	466
다른 RTTI 기반 라이브러리들 Other RTTI-Based Libraries	473
17: TObject와 System 유닛	475
TObject 클래스 The TObject Class	475
생성과 소멸 Construction and Destruction	476
오브젝트에 대해 알아보기 Knowing About an Object	476
TObject 클래스의 더 많은 메서드들 More Methods of the TObject Class	478
TObject의 가상 메서드들 TObject's Virtual Methods	480
TObject 클래스 요약 TObject Class Summary	483
유니코드와 클래스 이름 Unicode and Class Names	484
System 유닛 The System Unit	485
선택된 시스템 타입들 Selected System Types	485
System 유닛의 인터페이스들 Interfaces in the System Unit	486
선택된 시스템 루틴들 Selected System Routines	487
미리 정의된 RTTI 애트리뷰트들 Predefined RTTI Attributes	487
18: 다른 핵심core RTL 클래스들	489
Classes 유닛 The Classes Unit	490
Classes 유닛 안의 클래스들 The Classes in the Classes Unit	490
TPersistent 클래스 The TPersistent Class	491
TComponent 클래스 The TComponent Class	492
현대적인 파일 접근 Modern File Access	494
입력/출력 유틸리티 유닛 The Input/Output Utilities Unit	495
스트림에 대한 소개 Introducing Streams	496
Reader와 Writer 사용하기 Using Readers and Writers	498
문자열과 스트링 리스트 만들기 Building Strings and String Lists	500
TStringBuilder 클래스 The TStringBuilder class	500
스트링 리스트 사용하기 Using String Lists	501
런타임 라이브러리는 꽤 방대하다 The Run-Time Library is Quite Large	502
글을 마치며	504
부록 요약	505
a: 오브젝트 파스칼의 진화	506
Wirth의 파스칼 언어	507
터보 파스칼	507
초기 델파이의 오브젝트 파스칼	508
코드기어부터 엠바카데로까지의 오브젝트 파스칼	509
모바일 환경으로	509
델파이 10.x 시기	510

델파이 11 배포	510
b: 용어집	511

파트 I 기초

오브젝트 파스칼`Object Pascal`은 매우 강력한 언어다. 깔끔한 프로그램 구조 그리고 확장 가능한 데이터 타입 지원이 이 언어의 핵심 기반이다. 부분적으로 전통적인 파스칼 언어에 뿌리를 두고 있지만, 이 언어는 심지어 핵심 특징들까지도 초기 이후에 많은 확장이 있었다.

이 책의 첫 번째 파트는 이 언어의 문법`syntax`, 코딩 스타일, 프로그램 구조`structure`, 변수`variable` 및 데이터 타입`data type` 사용, 기본적인 언어 문장`statement` (조건`condition`, 루프`loop` 등), 프로시저`procedure`와 함수`function` 사용, 핵심 타입 (배열, 레코드`record`, 문자열`string` 등) 구조들에 대해 학습한다.

두 번째와 세 번째 파트는 더 수준 높은 기능들 즉 클래스`class`부터 제네릭 타입`generic type`에 이르기까지 그 기반을 탐구한다.

언어를 배우는 것은 집을 짓는 것과 같다. 우리는 견고한 땅과 좋은 기초를 바탕으로 시작해야 한다. 그렇지 않으면, 그 위에 있는 것이 아무리 보기 좋아도 그 어느 것도 안정적이지 않다.

파트 I 요약

- 1장: 파스칼 [Pascal](#)로 코딩하기
- 2장: 변수 [Variable](#)와 데이터 타입 [Data Type](#)
- 3장: 언어의 문장 [language statement](#)
- 4장: 프로시저와 함수 [Procedures and Functions](#)
- 5장: 배열과 레코드 [Arrays and Records](#)
- 6장: 문자열 [String](#)에 관한 모든 것

01: 파스칼Pascal로 코딩하기

이 장은 오브젝트 파스칼 애플리케이션을 이루는 구성 요소 몇 가지에서 시작하여, 코드와 주석 작성 방법, 키워드, 프로그램 구조를 다룬다. 간단한 애플리케이션들을 작성하면서 각 요소의 기능을 설명하고 핵심 개념들을 소개한다. 이 장에서 소개한 개념들은 다음 장부터 보다 자세히 차근차근 다룰 것이다.

코드Code부터 시작하자

이 장에서는 이 언어의 기초를 다룬다. 작동하는 애플리케이션 전체의 세부 사항들 모두를 지금 볼 수는 없다. 이제 간단한 프로그램 두 개를 보자(서로 구조가 다름). 너무 자세히 들어가지는 않는다. 지금은 프로그램의 구조를 볼 수 있는 예제를 만들어보면서 이 언어가 구체적으로 어떻게 구성되는 지를 설명하겠다. 그 외 더 많은 요소들은 다음에 설명하기로 한다. 이 책은 독자가 책 내용을 읽고 나서 가능한 빨리 실제로 해 볼 수 있기를 바라면서 집필되었으니, 바로 예제부터 시작하자.

오브젝트 파스칼 [Object Pascal](#) 은 통합 개발 환경 [Integrated Development Environment](#), 즉 IDE 를 함께 사용한다는 전제를 바탕으로 설계되었다. 이 강력한 조합을 통해 오브젝트 파스칼은 개발 속도와 편의성을 프로그래머에게 친화적인 언어를 통해 충족하면서도, 기계 친화적인 언어가 실현하는 빠른 실행 속도까지 두 가지를 동시에 충족한다.

IDE 를 사용하면, 사용자 인터페이스 디자인, 코드 작성, 프로그램 실행 등등 많은 작업을 할 수 있다. 나 역시 이 책 전반에 걸쳐 IDE 를 사용하면서 오브젝트 파스칼 언어를 소개한다.

첫 콘솔 Console 애플리케이션

우선, 간단한 Hello, World 콘솔 [console](#) 애플리케이션 코드를 보면서, 오브젝트 파스칼 프로그램을 구성하는 요소는 어떤 것들이 있는지 살펴보자. 콘솔 애플리케이션이란 그래픽 기반의 사용자 인터페이스가 없이 텍스트만을 표시하고 키보드 입력을 받는 프로그램이다. 콘솔 프로그램은 대체로 운영체제의 콘솔 또는 명령 프롬프트에서 실행된다. 콘솔 앱들은 모바일 플랫폼에서 대부분 의미가 없지만, 윈도우 [Windows](#)에서는 여전히 사용되며(마이크로소프트는 최근 cmd.exe, PowerShell, 터미널 액세스 등을 개선하기 위해 많이 노력하고 있음), 리눅스 [Linux](#)에서는 상당히 많이 사용된다.

아래 코드에 있는 여러 요소들이 무엇인지는 아직 설명하지는 않겠다. 그 설명은 이 책 앞 부분 여러 장에서 다룰 것이다. 일단 이 HelloConsole 예제를 보자.

```
program HelloConsole;

{$APPTYPE CONSOLE}

var
  StrMessage: string;

begin
  StrMessage := 'Hello, World';
  WriteLn(StrMessage);
  // 사용자가 엔터 키를 누를 때까지 기다린다
  ReadLn;
end.
```

참고 이 책에서 다루는 모든 예제의 전체 소스 코드는 깃허브 [GitHub](#) 리포지토리에서 온라인으로 확인할 수 있다. 예제를 얻는 방법은 서문에서 자세히 설명했다. 책 내용에서 프로젝트 이름(이 경우 HelloConsole)을 언급한다. 그것은 예제의 여러 파일들을 담고 있는 폴더의 이름이기도 하다. 프로젝트 폴더는 장 별로 묶여 있으므로 이 첫 번째 예제는 01/HelloConsole 폴더에 있다.

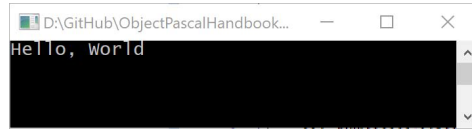
위 코드는, 첫째 줄에 프로그램 이름이 특정 선언 [declaration](#) 뒤에 있고, 그 다음 줄에는 컴파일러 지시어 [compiler directive](#)(\$ 기호로 시작하고 중괄호로 묶여 있음)가 있다. 이어서 변수 선언이 있다(이름이 주어진 문자열), 그리고 코드 세 줄과 주석 한 줄이 주요 부를 이루는 begin-end 블록 안에 있다. 코드 세 줄을 보자. 선언한 문자열에 값을 복사해 넣는다. 시스템 함수를 호출해 문자열 텍스트를 콘솔에 쓴다. 그리고 다른 시스템 함수를 호출해 사용자가 입력할 텍스트 한 줄을 읽으려고 기다린다(지금 경우는 사용자가 엔터 키를 누를 때까지). 함수를 직접 정의하는 법을 차차 배우게 된다. 하지만, 오브젝트 파스칼에는 이미 수백 가지 함수가 미리 정의되어 제공된다.

요소 하나 하나는 차근차근 배우기로 하고, 이 첫 부분에서는, 작지만 완전한 파스칼 프로그램이 어떤 모습인지만 봤다. 물론 사용자는 이 애플리케이션을 열고 실행할 수 있다. 그 결과는 다음과 같다(실제 윈도우 [Windows](#)에서 실행한 결과는 그림 1.1 참고).

| Hello, World

그림 1.1:

HelloConsole 예제를
윈도우에서 실행한 결과



첫 비주얼Visual 애플리케이션

지금 애플리케이션이 이렇게 구식인 콘솔 프로그램인 경우는 드물다. 대개 시각적 visual 요소(컨트롤 control 이라고 함)들로 구성되며 창(폼 form 이라고 함) 안에 표현된다. 이 책 역시 대부분 시각적 예제(대부분 간단한 텍스트 표시 정도이지만)를 만든다. 주로 FMX 라고도 알려진 파이어몽키 FireMonkey 라이브러리를 사용해 예제를 만들었다.

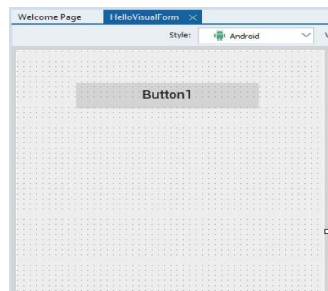
참고 델파이에서, 시각적 컨트롤은 두 종류가 있다. VCL(윈도우용 비주얼 컴포넌트 라이브러리)과 파이어몽키 FireMonkey (멀티 디바이스용 라이브러리, 데스크탑과 모바일 플랫폼을 모두 지원)이다. 어떤 경우라도 데모는 윈도우 전용 VCL 라이브러리에 적용하는 편이 보다 더 간단하다.

시각적 애플리케이션의 정확한 구조 structure 를 이해하려면 이 책의 상당 부분을 읽어야 한다. 폼 form 은 특정 클래스 class 로 만든 오브젝트 object 다. 그리고 그 안에는 메서드 method , 이벤트 핸들러 event handler , 프로퍼티 property 등을 가진다. 그 모든 특징을 살펴보려면 시간이 좀 걸린다. 그러나, 이 예제 애플리케이션 정도는 전문가가 아니어도 만들 수 있다. ‘새 데스크탑 애플리케이션 생성’ 또는 ‘새 모바일 애플리케이션 생성’ 메뉴 중 하나를 사용하면 된다. 이 책 앞부분에 있는 예제들은 대부분 파이어몽키 플랫폼이 기반이며, 폼의 문맥 context 과 버튼 클릭 동작을 사용한다. 첫 시작은, (데스크탑 또는 모바일 중) 어떤 유형이든 폼을 하나 만든다 (나는 대체로 multi-device “blank” application 을 선택한다. 이렇게 만들면 윈도우에서도 실행되기 때문이다). 그런 다음, 그 폼 위에 버튼 하나를 놓고 그 바로 밑에는 여러 줄 텍스트 컨트롤 (즉 Memo) 하나를 놓아서 결과를 보여주는 데 사용한다.

그림 1.2 는 우리가 만드는 모바일 애플리케이션을 델파이 IDE 안에 본 모습이다. (디자인 화면 위쪽 콤보 상자에서) 미리 보기 스타일을 안드로이드 Android 로 선택하고, 컨트롤 하나 즉 버튼 하나만 추가하면 이런 모습으로 보인다.

그림 1.2:

버튼 하나만 있는 간단한
모바일 애플리케이션
(HelloVisual 예제에서 발췌함)



위와 같은 애플리케이션을 직접 만들어보려면, 빈 폼에 버튼 하나를 추가하면 된다. 이제 여기에 실제 코드를 추가할 차례이다. 올려놓은 버튼을 더블-클릭해보자. 그러면 코드(또는 이와 유사한) 골격이 아래와 같이 만들어진다.

```
procedure TForm1.Button1Click(Sender: TObject)
begin

end;
```

여러분은 아직 클래스 `class`의 메서드 `method` (위 코드에 있는 `Button1Click`)가 무엇인지 모를 수 있다. 그래도, 이 코드 조각 안에(즉, `begin` 과 `end` 키워드 사이에) 코드를 써넣을 수는 있다. 여기에 써넣은 코드는 이 버튼을 누르면 실행된다는 것만 알자.

이 첫 번째 "시각적" 프로그램의 코드는 앞서 만들었던 첫 번째 콘솔 애플리케이션과 기본적으로 같다. 단지 문맥 `context` 이 다르다. 그리고, 다른 라이브러리 함수 즉 `ShowMessage` 라는 글로벌 `global/전역` 함수를 사용하기 때문에 문자열을 메시지 상자 안에 표시한다는 점이 다르다. 아래 코드는 `HelloVisual` 예제에 들어 있다. 굳이 그 예제가 없어도 이 정도는 여러분이 아주 쉽게 처음부터 만들 수 있을 것이다.

```
procedure TForm1.Button1Click(Sender: TObject)
var
  StrMessage: string;
begin
  StrMessage := 'Hello, World';
  ShowMessage(StrMessage);
end;
```

실제 코드 문장은 `begin` 뒤부터 시작하지만, `StrMessage` 변수 선언은 `begin` 문보다 앞에 있다는 점에 유의하자. 다시 말하지만, 명쾌하게 이해하지 못해도 걱정할 필요가 없다. 점차 더 자세하게 설명될 것이다.

참고 이 예제의 소스 코드는 이 장을 의미하는 01 컨테이너 안에 들어있는 폴더에 있다. 하지만 이 경우에는, 폴더 안에 예제와 이름이 같은 프로젝트 파일이 있고 아울러 프로젝트 이름 뒤에 "Form"이라는 단어가 붙은 유닛 파일도 하나가 있다. 이 책의 예제들은 이런 방식을 표준으로 작성되었다. 프로젝트 하나의 구조가 어떻게 구성되는지는 이 장의 마지막 부분에서 다룬다.

그림 1.3 은 이 간단한 프로그램의 결과다. FMX MobilePreview 모드를 활성화한 상태에서 윈도우에서 실행한 모습이다 (안드로이드, iOS, 맥 OS 에서도 실행할 수 있다. 하지만, 그러려면 IDE 에 몇 가지 추가 구성이 되어 있어야 한다).

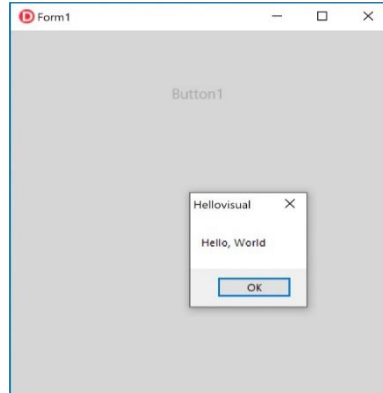
참고 FMX MobilePreview 모드는 윈도우 애플리케이션이 모바일 앱처럼 보이도록 한다. 나는 이 책의 예제 대부분에서 이 모드를 활성화했다. MobilePreview 유닛 `unit`을 프로젝트 소스 코드 안에 있는 `uses` 문에 추가하면 된다.

이제 델파이 프로그램을 만드는 방법과 테스트하는 방법을 알았으니, 다시 원점으로 돌아가서, 약속한 대로, 애플리케이션의 초기 구성 요소 몇 가지에 대한 세부 사항을

모두 다뤄보자. 가장 먼저 알아야 할 것은 프로그램을 어떻게 읽는지, 다양한 요소를 어떻게 작성하는지, 그리고 방금 우리가 만든 (PAS 파일 하나와 DPR 파일 하나) 애플리케이션은 그 구조가 어떻게 되어 있는지에 대한 것들이다.

그림 1.3:

버튼 하나만 있는 간단한
모바일 애플리케이션
(HelloVisual 예제에서 발췌함)



구문 Syntax 및 코딩 스타일 Coding Style

오브젝트 파스칼 언어 문장을 실제로 작성하는 단계로 넘어가기 전에, 오브젝트 파스칼 코딩 스타일 중 몇 가지 요소를 꼭 강조하고 싶다. 다음 질문에 대한 대답이기도 하다. 문장 규칙 [syntax rule](#) (아직 살펴보지 않았다)은 별개로 하고, 코드를 어떤 식으로 작성하는 것이 좋을까? 여기에 정답은 없다. 개인 취향에 따라 스타일이 다를 수 있기 때문이다. 하지만 몇 가지 스타일 원칙들을 알아야 한다. 주석 [comment](#), 대문자, 공백 [space](#) 등과 관련된 그리고 [예쁘게 인쇄하기 pretty-printing](#) (컴퓨터가 아니라, 사람이 보기에 예쁘다는 의미다. 이 용어는 오래 전에 사용되던 것이다)에 대한 원칙들이다.

대체로 모든 코딩 스타일의 목표는 명확성이다. 개발자가 결정하는 코딩 스타일과 코드 서식은 일종의 속기 [shorthand](#) 형식이 되어, 주어지는 코드 조각의 목적을 알려준다. 명확성은 일관성이라는 필수 도구가 있어야 실현된다 - 어떤 스타일을 선택하든 프로젝트 전체에서 그리고 프로젝트들 사이에서 일관성이 지켜져야 한다.

팁 IDE(통합 개발 환경)에는 코드 서식 자동 반영 기능이 있다(유닛 또는 프로젝트 수준에서 반영 됨). 예를 들어, Ctrl+D 키를 쓰면 코드에 다시 서식을 반영하도록 에디터에게 요청할 수 있다. 나만의 서식 규칙들 세트를 지정하려면 40 가지 서식 요소(IDE 옵션에 있음)를 조정하면 된다. 게다가 이 설정은 팀의 다른 개발자와 공유하고 서식을 일관성 있게 유지할 수도 있다. 하지만, 이 서식 자동 반영은 최신 언어 특징 중 몇 가지에서는 작동하지 않는다.

주석Comment

코드는 그 자체가 스스로 내용을 설명하는 경우가 많다. 하지만, 프로그램의 소스 코드 안에 주석을 상당히 많이 넣어서 다른 사람에게 (그리고 나중에 그 코드를 보게 될 자기 자신에게) 코드가 왜 그런 식으로 작성되었는지, 무슨 가정이 깔려 있는지를 설명하는 것이 좋다.

기존 파스칼에서, 주석은 중괄호 또는 괄호와 별표의 조합을 사용해서 주석을 감싸는 방식밖에 없었다. 하지만 최신 버전에서는 C/C++에서 쓰는 스타일인 한 줄 주석 즉 사선 두 개를 붙이는 형태도 사용할 수 있다. 이 방식은 그 한 줄을 모두 주석으로 처리한다. 따라서 뒤에 닫는 기호를 붙일 필요가 없다.

```
// 이 줄 끝까지 모두 주석이다
{ 이 여러 줄이
  하나의 주석이다 }
(* 이것도 마찬가지로
  여러 줄이 하나의 주석이다 *)
```

첫 번째 주석 형태가 가장 흔하게 쓰인다. 원래 파스칼에는 없던 것이다. C/C++에서 가져왔다. C/C++ 계열 즉 C#, Objective-C, Java, JavaScript 에서는 여러 줄 주석에 `/* 주석 */`을 사용한다.

두 번째 주석 형태는 세 번째 형태보다 더 흔하게 사용된다. 세 번째 주석 형태는 유럽에서 선호하는 경향이 있다. 키를 조합하는 방식임에도 선호하는 이유는 많은 유럽 키보드에는 중괄호 기호가 없거나 사용하기 불편하게 되어 있기 때문이다. 어쨌든 이 형태는 가장 오래된 구문 [syntax](#) 이고, 유행에 좀 뒤떨어졌다.

한 줄 단위 주석 형태는 짧은 주석을 달 때 또는 코드 한 줄을 주석으로 뺄 때 매우 유용하다. 최신 오브젝트 파스칼 언어에서 가장 흔하게 쓰이는 주석 형태이기도 하다.

팁 IDE 에디터에서, 단축키를 사용해 현재 줄(또는 선택한 여러 줄)을 주석으로 만들거나 풀 수 있다. 미국 키보드 배치라면 `Ctrl+/` 키 조합을 사용한다. 만약 다른 키보드 배치라면 `/`가 있는 다른 물리적 키를 사용한다. 이런 실제 키 조합은 에디터의 팝업 메뉴에서 볼 수 있다.

위와 같이 세 가지 다른 주석 방식이 도움이 될 때가 있다. 주석 안에 또 다른 주석을 넣어야 하는 상황에서 그렇다. 소스 코드 여러 줄을 주석 처리하여 비활성화 하려는 데, 그 안에 이미 주석이 들어 있다면, 그 주석과 같은 기호를 쓸 수 없다.

```
{
  코드...
  {같은 주석 기호가 중첩된 주석, 문제가 발생한다}
  코드...
}
```


위 코드는 컴파일러 에러가 발생한다. 가장 먼저 만나는 닫힌 중괄호가 주석 영역의 끝이라고 표시하기 때문이다. 이럴 때는 보조 주석 식별자 `identifier` 를 사용해, 다음과 같이 작성하면, 올바른 코드가 된다.

```
{
  코드...
  // 이 주석은 문제가 없다
  코드...
}
```

주석을 중첩하는 또 다른 방법이 있다. 앞에서 설명한 한 줄 주석 형태를 사용하여 여러 줄을 주석으로 처리하는 것이다. 그러면 `//` 주석이 이미 있는 줄 앞에 추가로 주석 표시가 하나 더 달린다. 이렇게 하면 그 여러 줄에서 다시 주석을 푸는 것도 편하다 (원래 주석이 유지됨)

참고 중괄호 또는 괄호-별표 조합에서 달러 기호(\$)가 붙으면 주석이 아니라 컴파일러 지시어가 된다. 이미 첫 예제에 있는 `{$APPTYPE CONSOLE}` 줄에서 봤다. 컴파일러 지시어는 컴파일러가 특별한 작업을 수행하도록 지시한다. 이 내용은 이 장의 마지막에 간략히 설명한다. 실제로, 컴파일러 지시어도 여전히 주석이다. 예를 들어 `{$X+ 이것은 주석입니다}`는 문법에 맞다. 유효한 지시어이고 동시에 유효한 주석이다. 비록 정상적인 프로그래머들은 대체로 지시어와 주석을 구분해 작성하는 경향이 있지만 말이다.

주석 [Comment](#) 및 XML 문서 [XML Doc](#)

주석의 특별한 버전이 있다. 다른 프로그래밍 언어에서도 흔하게 볼 수 있으며, 컴파일러가 특별하게 취급하는 주석이다. 이 특별한 주석은 컴파일러를 거치면서 추가 문서와 XML 파일로 생성된다. 또한 IDE 헬프 인사이트 [Help Insight](#) 에서 사용된다.

참고 델파이 [Delphi](#) IDE의 헬프 인사이트는 심볼 [symbol](#)에 대한 정보(심볼 타입과 정의된 코드 위치 등)를 자동으로 표시한다. XML 문서 [XML Doc](#) 주석을 사용하면, 소스 코드 안에 여러분이 직접 적어놓은 상세한 정보들은 생성되는 문서에 그리고 헬프 인사이트에 추가로 반영된다.

XML 문서는 `///` 주석 또는 `{!}` 주석을 사용하여 활성화할 수 있다. 이 주석 안에는 일반 텍스트 또는 (더 좋게는) 특정 XML 태그를 사용한다. 그러면 그 주석을 적은 대상의 심볼 [symbol](#), 파라미터 [parameter](#), 반환값 [return value](#) 등에 대한 정보를 보여줄 수 있다. 텍스트는 매우 간단하고 자유로운 형식으로 적으면 된다.

```
public
  /// 이것은 사용자가 지정한 메서드이다
  procedure CustomMethod;
```

XML 문서 생성을 활성화해 놓았다면, 컴파일러는 이 정보를 XML 출력에 추가한다. 생성되는 문서는 다음과 같다.


```
<procedure name="CustomMethod" visibility="public">
  <devnotes>
    이것은 사용자가 지정한 메서드이다
  </devnotes>
</procedure>
```

그림 1.4 처럼, 심볼 **symbol** 위로 마우스를 가져가면 동일한 정보가 IDE 에 표시된다.

그림 1.4:

Delphi IDE의
헬프 인사이트 **Help Insight**는
/// 주석으로 작성된
XML 문서 정보를
표시한다.

```
public
  /// <summary> 이것은 사용자가 지정한 메소드이다 </summary>
  procedure CustomMethod;
end;

var
  Form1: TForm1;

implementation

procedure TForm1.Button1Click(Sender: TObject);
begin
  CustomMethod;
end;

TForm1.CustomMethod Method - Unit1.pas (18,15)
  이것은 사용자가 지정한 메소드이다
  Declared in Unit1.TForm1
```

권장 가이드라인에 맞게, XML 주석 안에 <summary> 구역을 넣으면, 그 내용 역시 헬프 인사이트 창에 그대로 똑같이 표시된다.

```
public
  /// <summary> 이것은 사용자가 지정한 메서드이다 </summary>
  procedure CustomMethod;
```

여러분이 쓸 수 있는 많은 다른 XML 태그들이 있다. 파라미터, 반환값 등에 대해 더 자세한 정보를 넣을 수 있어서 좋다. 제공되는 태그 목록은 아래 문서에 있다.

http://docwiki.embarcadero.com/RADStudio/en/XML_Documentation_Comments

심볼 식별자 **Symbolic Identifier**

프로그램 하나는 수많은 서로 다른 심볼 **Symbol**들로 구성된다. 여러분은 심볼을 만들 수 있다. 다양한 요소(데이터 타입, 변수, 함수, 오브젝트, 클래스 등)들에 이름을 붙이면 된다. 원하는 거의 모든 이름을 식별자 **identifier**로 사용할 수 있다. 하지만, 따라야 할 몇 가지 규칙이 있다.

- 식별자는 공백 **space**을 가질 수 없다 (공백은 식별자들을 기타 다른 언어 요소들과 구분하기 위해 사용되기 때문임).
- 식별자에는 문자 **letter**와 숫자 **number**를 가질 수 있다. 여기에는 유니코드 문자 집합의 모든 문자도 해당되므로, 원한다면 모국어 이름을 만들 수 있다(권장하지는 않는다. IDE 안에 있는 도구 중 몇 가지가 그 이름들을 지원하지 않을 수도 있기 때문임).

- 식별자가 가질 수 있는 기존 아스키 기호는 밑줄(_)기호뿐이다. 그 외 다른 아스키 기호는 허용되지 않는다. 식별자에 사용할 수 없는 기호에는 매치 기호(+, -, *, /, =), 괄호, 중괄호, 마침표, 특수 문자(@, #, \$, %, ^, &, \, | 등)가 있다. 하지만 ♪ ♫ 또는 ∞와 같은 유니코드 기호는 사용할 수 있다.
- 식별자의 첫 문자는 문자 또는 밑줄이어야 한다. 숫자는 식별자의 맨 앞이 아닌 위치에서 사용할 수 있다. 여기서 숫자는 0에서 9까지의 아스키 숫자를 말한다. 그런데, 숫자에 대응하는 다른 유니코드 표현들은 허용된다.

다음은 전형적인 식별자의 예다. `IdentifiersTest` 애플리케이션에 나열돼 있다.

```
MyValue
Value1
My_Value
_Value
Val123
_123
```

다음은 문법을 준수한 유니코드 식별자의 예다(마지막이 약간 극단적이긴 함).

```
Cantù (악센트가 있는 라틴 문자)
잔액 (한글)
画像 (일본어)
☼ (해를 표현하는 유니코드 기호)
```

다음은 잘못된 식별자의 예다.

```
123
1Value
My Value
My-Value
My%Value
```

팁 실행 중 `runtime`에 유효한 식별자인지를 확인해야 하는 경우(다른 개발자를 돕기 위한 도구를 작성하는 경우를 제외하고는 이런 경우가 거의 없음)에 사용할 수 있도록 런타임 라이브러리에는 `IsValidIdent`라는 함수가 있다.

대소문자 구분 및 대문자 `Upper-Case` 사용

C 구문을 기반으로 하는 언어(C++, Java, C#, and JavaScript) 등 많은 다른 언어들과 달리, 오브젝트 파스칼 컴파일러는 식별자의 대소문자를 구분하지 않는다. 따라서 식별자 `Myname`, `MyName`, `myname`, `myName`, `MYNAME`은 모두 같다. 내 개인적인 의견으로는, 대소문자-구분없음 `case-insensitivity`은 분명히 긍정적인 점이다. 대소문자를 가리는 언어는, 대소문자를 틀렸을 때, 구문 에러와 기타 미묘한 실수들이 발생하기 때문이다.

하지만, 유니코드를 식별자에 사용할 수 있다는 사실을 생각해본다면, 상황이 조금 더 복잡하다. 글자의 대문자 버전은 동일한 요소로 취급되지만, 그 글자에 악센트가 붙은 버전은 다른 요소로 취급되기 때문이다. 다시 말하면 다음과 같다.


```
cantu: Integer;
Cantu: Integer; // 예러: 식별자 중복
cantù: Integer; // 문법에 맞음: 다른 식별자임
```

경고 대소문자 구분 규칙에 한 가지 예외가 있다. 컴포넌트 패키지의 *Register* 프로시저는 대문자 *R*로 시작해야 한다. 이는 C++ 호환성 문제 때문이다. 물론 다른 언어에서 내보내진 식별자(예: 운영체제에 내장된 함수)를 참조할 때는 해당 대소문자를 맞추어 사용해야 할 수도 있다.

대소문자를 가리지 않을 때 생기는 미묘한 단점이 몇 가지 있다. 첫째, 대소문자가 달라도 모두 동일한 식별자이므로 다른 요소처럼 사용하지 않도록 주의해야 한다. 둘째, 대문자를 일관성 있게 사용해야 한다. 그래야 코드의 가독성이 높아진다.

비록 컴파일러가 강제하지는 않지만, 대소문자를 일관성 있게 사용하는 것은 좋은 습관이다. 일반적인 접근 방식은 각 식별자에서 첫 글자만 대문자로 사용하는 것이다. 식별자가 여러 개의 연속된 단어로 구성된 경우에는 각 단어의 첫 글자에 대문자를 사용하면 좋다(식별자에 공백을 넣을 수 없음).

```
MyLongIdentifier MyVeryLong
AndAlmostStupidIdentifier
```

이를 흔히 "파스칼-케이싱 [Pascal-casing](#)"이라고 부른다. 자바 등 C 구문을 기반으로 하는 다른 언어에서 사용하는 소위 "카멜-케이싱 [Camel-casing](#)" 즉, 안에 들어 있는 각 단어의 첫 글자에는 대문자를 사용하지만, 맨 앞의 글자는 소문자를 사용하는 방식과 대비된다.

myLongIdentifier

현장에서는, 오브젝트 파스칼 코드는, 로컬 [local/지역](#) 변수에는 카멜-케이싱(맨 앞 글자만 소문자)을 사용하는 경우가, 클래스 요소, 파라미터, 기타 글로벌 [global/전역](#) 요소에는 파스칼-케이싱을 사용하는 경우가 점점 더 많아지고 있다. 어쨌든, 이 책에 있는 소스 코드 조각에서는 모든 심볼에 일관성 있게 파스칼 케이싱을 사용하려고 노력했다.

공백 [Whitespace](#)

대소문자 구분 외에, 소스 코드에 있는 공간 [space](#), 새 줄 [new line](#), 탭 [tab](#) 역시 컴파일러가 완전히 무시한다. 이것들을 통칭하여 [공백 whitespace](#)이라고 한다. 공백은 코드를 더 쉽게 읽을 수 있게 해주는 역할만 할 뿐 컴파일에 아무런 영향을 미치지 않는다.

기존 베이직 [BASIC](#) 과 달리, 오브젝트 파스칼에서는 문장 [statement](#) 하나를 여러 줄에 걸쳐 줄을 바꾸면서 작성할 수 있다. 이처럼 문장이 한 줄을 넘을 수 있기 때문에 생기는 단점이 있는데, 바로 각 문장이 끝날 때 마다 세미콜론을 넣어야 한다는 점이다. 더 정확히 말하면, 프로그래밍 문장은 그 다음 문장과 명확히 구분되도록 해야 한다. 프로그래밍 문장 하나를 여러 줄에 나누어 적는 것을 수 없는 유일한 경우는 문자열 리터럴 [string literal](#) 뿐이다 (웁진이: 이제는 문자열 리터럴도 여러 줄에 나누어 적을 수 있다. 델파이 12 버전부터 가능).

이상하겠지만, 다음 줄에 있는 문장들을 컴파일 하면 모두 똑같아진다.

```
A := B + 10;
A :=
  B
  +
  10;

A
:=
// 이것은 문장 중간에 있는 주석임
B + 10;
```

다시 말하지만, 코드를 작성할 때, 공백과 여러 줄을 어떻게 사용해야 한다는 고정된 규칙은 없다. 다만, 경험 법칙 몇 가지를 정리하면 다음과 같다.

- 에디터에는 80자 정도 뒤에 세로 줄이 있다. 이 줄을 기준으로 삼아 그 경계를 넘지 않도록 노력하자. 그러면 소스 코드가 더 보기 좋아진다. 그리고 가로 스크롤을 하지 않아도 화면이 작은 컴퓨터에서 읽을 수 있다. 한 줄 당 80자까지 제한하는 원래 의도는 코드를 더 보기 좋게 인쇄하는 것이었다. 그런데, 요즘은 코드를 인쇄하는 경우가 그리 많지 않다.
- 함수나 프로시저에서 사용되는 파라미터들이 복잡하고 여러 개라면, 파라미터들을 서로 다른 줄에 배치하는 것이 일반적인 관행이다.
- 한 줄을 완전히 비우는 것도 좋다. 주석 앞에 놓거나 또는 긴 코드 사이에 빈 줄을 끼워 코드를 작은 부분으로 나누어 구분할 수 있다. 이 간단한 아이디어 하나로 코드의 가독성이 높아진다. 책에서 단락과 여백을 사용하는 것과 같은 이치이다.
- 공백을 사용하자. 함수를 호출할 때 파라미터 사이를 공백으로 구분한다. 심지어 여는 괄호 앞에도 공백을 둘 수도 있다. 나는 표현식 `expression` 안에서 피연산자 `operand` 사이에 공백을 두어 구분하는 것을 좋아한다. 하지만, 이는 선호도에 따른 문제다.

들여쓰기 Indentation

공백 사용 관련 마지막 제안이 있다. 이것은 파스칼의 전형적인 언어-서식 스타일과 들여쓰기에 관한 것이다.

참고 들여쓰기 규칙은 개인 취향에 따라 다를 수 있다. 탭과 공백 중 어느 것이 더 좋은지를 두고 다룰 생각은 없다. 지금 소개하는 규칙은 오브젝트 파스칼 세계에서 "가장 일반적인" 즉 "표준" 서식 스타일로, 델파이 라이브러리 소스 코드에 적용된 규칙이다. 파스칼 세상에서는 이 규칙 세트를 예전부터 예쁘게-인쇄하기 `pretty-printing`라고 불렀다. 이 용어는 지금 거의 사용되지 않는다.

규칙은 간단하다: 복합문 `compound statement` 을 작성한다면, 현재 문장에서 오른쪽으로 두 칸을 들여 쓴다 (탭이 아니다. 탭은 일반적으로 C 프로그래머들이 사용한다). 복합문 안에서 또 다른 복합문을 써야 할 때는, 맨 앞에서 네 칸을 들여 쓴다. 이런 식으로 들여쓰기를 해 나간다.


```

if ... then
    statement;

if ... then
begin
    statement1;
    statement2;
end;

if ... then
begin
    if ... then
        statement1;
        statement2;
    end;
end;

```

다시 말하지만, 이 일반 규칙에 대해 프로그래머들은 저마다 다르게 해석한다. 어떤 프로그래머는 `begin end` 문장을 안쪽 코드의 수준이 같게 맞춰 들여쓰기도 한다. 또 다른 프로그래머는 `begin` 을 앞 문장의 맨 뒤에 두기도 한다(C 언어 같은 방식이다). 들여쓰기는 대부분 개인 취향의 문제다.

이와 비슷한 들여쓰기 형식으로는, 변수 또는 데이터 타입을 `type` 과 `var` 키워드 뒤에 나열할 때 자주 사용된다. 아래 코드와 같다.

```

type
    Letters = ( 'A', 'B', 'C' );
    AnotherType = ...

var
    Name : string;
    I : Integer;

```

또한, 문장을 여러 줄에 걸쳐 적거나, (한 줄 당 파라미터 하나가 아니라) 함수의 파라미터를 나열하다가 줄을 바꿔야 하는 경우에도 이런 들여쓰기가 흔하게 사용된다.

```

MessageDlg( '이것은 메시지입니다',
    mtInformation, [mbOK], 0 );

```

구문Syntax 강조 표시Highlighting

오브젝트 파스칼 코드를 더 쉽게 읽고 쓸 수 있도록, IDE 에디터에는 구문 강조 표시 `syntax highlighting` 라는 기능이 있다. 입력된 단어의 언어적 의미에 따라 다양한 색상과 글꼴 스타일이 반영되어 표시된다. 기본 설정인 경우, 키워드 `keyword` 는 굵게, 문자열 `string` 과 주석 `comment` 은 컬러(종종 기울임꼴)로 표현된다.

그 최대 수혜자 3 가지는 예약어 `reserved word`, 주석, 문자열이다. 철자가 틀린 키워드, 제대로 종료되지 않은 문자열, 여러 줄에 걸친 주석의 길이 등을 한 눈에 알 수 있다.

구문 강조 표시 `syntax highlighting` 를 맞춤 지정하는 방법은 쉽다. IDE 의 Options 대화 상자에 있는 Editor Colors 페이지에서 지정하면 된다. 오브젝트 파스칼 소스 코드를

보는 사람이 그 컴퓨터에서 오직 자신 혼자라면 내가 원하는 색상을 선택하면 된다. 만약 다른 프로그래머와 긴밀하게 작업하는 경우라면, 표준 색상 구성에 모두 동의를 해야 한다. 나는 다른 컴퓨터에서 작업할 때 내가 평소에 쓰는 구문 색상 [syntax coloring](#) 구성이 아니어서 혼란스러웠던 적이 많았다.

에러 인사이트 [Error Insight](#) 및 코드 인사이트 [Code Insight](#)

IDE 에디터에는 올바른 코드를 작성하도록 도움을 주는 기능이 많다. 가장 눈에 띄는 기능은 에러 인사이트 [Error Insight](#) 다. 컴파일러가 이해하지 못한 소스 코드 요소 아래 빨간색 구불구불한 밑줄을 표시한다. 워드 프로세서의 맞춤법 실수 표시와 방식이 같다.

참고 완벽하게 정상 코드인데 오류 인사이트가 표시되는 경우를 피하기 위해, 가끔 프로그램을 먼저 컴파일해야 할 때가 있다. 또한 현재 컴포넌트에게 필요한 적절한 유닛을 강제로 포함시키도록 `form` 파일 등을 먼저 저장해야 오류 인사이트에 잘못된 표시가 나오는 문제가 해결되는 경우가 있다. 이러한 문제는 델파이 10.4에 처음 도입된 새 LSP(언어 서버 프로토콜 [Language Server Protocol](#)) 기반 코드 인사이트에 의해 대부분 해결되었다.

또 다른 기능, 즉, 코드 완성 같은 기능은 현재 작성하고 있는 곳에 들어가기 알맞은 심볼 [symbol](#) 을 목록으로 제시하여 코드를 빠르고 정확하게 작성할 수 있도록 돕는다. 함수나 메서드에 파라미터가 있는 경우에는 타이핑을 할 때 해당 파라미터가 목록으로 표시된다. 또한 심볼 [symbol](#) 위에 마우스를 놓으면 해당 심볼의 정의 [definition](#) 가 표시된다. 그러나, 이런 기능은 IDE 에디터 전용 기능이므로, 더 자세히 들어가지 않겠다. (오브젝트 파스칼 코드를 작성할 때 단연코 가장 많이 쓰이는 도구들인 것은 분명하지만) 이 책에서는 언어에 집중하기로 하자.

이 언어의 키워드들 [Language Keywords](#)

키워드는 언어가 미리 예약하고 사용하는 식별자 [identifier](#) 모두를 말한다. 이 심볼들의 의미와 역할은 이미 정해져 있다. 따라서, 우리는 그 이외의 문맥 [context](#) 에서 쓸 수 없다. 예약어 [reserved word](#) 와 지시어 [directive](#) 사이에는 공식적인 차이가 있다. 예약어는 식별자로 사용될 수 없다. 반면, 지시어는 특별한 의미를 가지고 있지만 문맥 다른 곳에서는 식별자로 사용될 수 있다 (하지만, 그렇게 하지 않기를 권장한다). 실제 현장에서는, 어떤 키워드도 식별자로 사용해서는 안 된다.

아래 코드를 실행하면 (여기에서 `property` 는 키워드다):

```
var
  property: string
```

다음과 같은 에러 메시지가 표시된다.

```
E2029 Identifier expected but 'PROPERTY' found
```


대체로, 키워드를 잘못 사용하면, 상황에 따라 오류 메시지가 다르게 표시된다. 이는 컴파일러가 그 키워드를 인식하고 있지만, 코드 상의 위치 또는 키워드 뒤에 나오는 요소들 때문에 코드의 명령을 명확히 알지 못하고 있다는 의미다.

전체 키워드를 여기에 나열하지는 않는다. 키워드 중에는 다소 모호하고 거의 쓰이지 않는 것들이 있기 때문이다. 여기에는 역할 별로 묶어서 몇 가지를 나열한다. 나열할 것과 생략된 키워드까지 모두 나열된 목록은 꽤 길다. 아래 공식 참고 자료에 있다.

[http://docwiki.embarcadero.com/RADStudio/en/Fundamental Syntactic Elements \(Delphi\)#Reserved Words](http://docwiki.embarcadero.com/RADStudio/en/Fundamental_Syntactic_Elements_(Delphi)#Reserved_Words)

참고 지금 여기에서는 일반적으로 가장 일반적인 문맥만을 언급하지만 (키워드 몇 개는 두 번 나열하기도 했음). 다른 문맥^{context}에서 사용될 수 있는 키워드들이 몇 개 있다는 점을 알아 두자. 그 이유는 기존 애플리케이션을 망가뜨릴 수 있는 새로운 키워드를 도입하는 것을 컴파일러 팀이 수년 동안 꺼리면서 기존 키워드 중 일부를 *재활용*^{recycled}했기 때문이다.

이제, 앞에서 예제를 통해서 보았던 키워드부터 파악해보자. 다음은 **애플리케이션 프로젝트의 구조**를 정의하기 위해 사용되는 키워드다.

program	애플리케이션 프로젝트의 이름을 표시한다
library	라이브러리 프로젝트의 이름을 표시한다
package	패키지 라이브러리 프로젝트의 이름을 표시한다
unit	소스 코드 파일인 유닛의 이름을 표시한다
uses	코드가 의존하고 있는 다른 유닛들의 이름을 참조 ^{refer} 한다
interface	유닛에서 인터페이스 선언이 구현되는 부분
implementation	유닛에서 실제 코드가 구현되는 부분
initialization	프로그램이 시작되는 시점에 실행되는 코드 부분
finalization	프로그램이 종료되는 시점에 실행되는 코드 부분
begin	코드 블록의 시작
end	코드 블록의 끝

선언 ^{declaration} 중 여러 기본 **데이터 타입**과 (그 데이터 타입의) **변수 선언** 관련 키워드

type	타입 ^{type} 을 선언하는 블록을 시작한다
var	변수 ^{variable} 를 선언하는 블록을 시작한다
const	상수 ^{constant} 를 선언하는 블록을 시작한다
set	<i>파워 세트</i> ^{power set} 데이터 타입을 정의한다
string	문자열 ^{string} 변수 또는 사용자 지정 문자열 타입을 정의한다
array	배열 ^{array} 타입을 정의한다
record	레코드 ^{record} 타입을 정의한다
integer	정수 ^{integer} 변수를 정의한다
real, single, double	부동 소수점 ^{floating point} 변수를 정의한다
file	파일 ^{file} 을 정의한다

참고 오브젝트 파스칼에는 다른 데이터 타입들도 많이 있다. 그것들은 뒤에서 다루겠다.

기본 언어 문장 `statement` 즉, 조건문, 반복문, 함수, 프로시저 등에서 사용하는 키워드

If	조건문 <code>conditional statement</code> 을 시작한다
then	조건문과 실행되는 코드를 분리한다
else	조건에 맞지 않을 경우에 실행되는 대체 코드를 표시한다
case	여러 옵션이 있는 조건문을 시작한다
of	조건과 해당 옵션들을 분리한다
for	순환 반복 <code>repetitive cycle</code> 주기를 시작한다
to	for 순환을 마지막으로 실행하게 하는 가장 큰 값을 표시한다
downto	for 순환을 마지막으로 실행하게 하는 가장 작은 값
in	순환을 반복할 모듬 <code>collection</code> 을 표시한다
while	조건부 <code>conditional</code> 순환 반복을 시작한다
do	순환 조건과 실행되는 코드를 분리한다
repeat	종료 조건이 있는 순환 반복 <code>repetitive cycle</code> 을 시작한다
until	순환 반복의 종료 조건을 표시한다
with	작업에서 사용하는 데이터 구조를 표시한다
function	결과 <code>result</code> 를 반환하는 하위-루틴 <code>sub-routine</code> 또는 문장들의 집합
procedure	결과를 반환하지 않는 하위-루틴 또는 문장들의 집합
inline	함수 호출하는 대신, 그 함수의 실제 코드 자체를 안에 넣으라고 컴파일러에게 요청한다. 더 빠르게 실행하기 위함
overload	함수 또는 프로시저의 이름을 재사용 <code>reuse</code> 할 수 있도록 허용한다

클래스 및 오브젝트와 관련된 많은 키워드

class	클래스 타입을 표시한다
object	예전에 클래스 타입을 나타내는 데 사용 (현재는 사용되지 않음)
abstract	정의 <code>definition</code> 가 완전하게 되지 않는 클래스
sealed	상속할 수 없어서 자식 클래스를 가질 수 없는 클래스
interface	인터페이스 타입을 표시한다(위 첫 번째 그룹에도 나열됨)
constructor	오브젝트 또는 클래스를 초기화 <code>initialization</code> 하는 메서드
destructor	오브젝트 또는 클래스를 비우고 정리하는 <code>cleanup</code> 메서드
virtual	가상 <code>virtual</code> 메서드
override	가상 <code>virtual</code> 메서드가 수정된 버전
inherited	기반 <code>base</code> 클래스의 메서드를 참조한다
private	클래스 또는 레코드 내용 중 외부 접근이 안되는 부분
protected	클래스 내용 중 외부에서 접근이 제한적으로 허용되는 부분
public	클래스 또는 레코드 내용 중 외부 접근이 완전히 허용되는 부분
published	클래스 내용 중 특별하게 사용자 접근이 허용되는 부분
strict	<code>private</code> 과 <code>protected</code> 구역을 보다 강력하게 제한
property	값 <code>value</code> 또는 메서드 <code>method</code> 를 매핑 <code>mapping</code> 하는 심볼 <code>symbol</code>
read	프로퍼티 <code>property</code> 의 값 <code>value</code> 을 가져오기 위한 매퍼 <code>mapper</code>
write	프로퍼티 <code>property</code> 의 값을 지정하기 위한 매퍼 <code>mapper</code>
nil	오브젝트 없음 <code>zero</code> 을 의미하는 값(다른 엔티티에도 사용됨)

예외 처리 [exceptions handling](#) (11 장 참조)를 위해 사용되는 키워드

try	예외 처리 exceptions handling 블록의 시작
finally	예외와 관계없이 실행될 코드를 시작한다
except	예외 발생 시 실행될 코드를 시작한다
raise	예외를 발동 trigger 시키는 데 사용된다

연산자 [operator](#) 에 사용되는 키워드. 이 장 뒷부분에 있는 "표현식 [Expression](#) 및 연산자"에서 다루게 된다. (일부 수준 높은 연산자라서 이 장 이후에 다룰 키워드들은 생략함).

as	and	div
is	in	mod
not	or	shl
shr	xor	

마지막으로, **덜 자주 사용되는 키워드**. 사용하면 결코 안 되는 일부 오래된 키워드도 포함되어 있다. 다시 말하지만, 지금은 키워드가 어디에 사용하는지만 간략히 알려줄 뿐이다. 내용은 이 책의 뒷부분에서 차차 살펴보게 될 것이다.

default	프로퍼티 property 의 기본값 default value 을 표시한다
dynamic	가상 virtual 메서드인데 조금 다르게 구현되어 있다
export	예전에 legacy 내보내기에 사용되던 키워드. 아래 키워드로 대체됨
exports	DLL 프로젝트 안에서, 내보내지는 함수를 나열한다
external	바인딩하려는 외부 DLL 함수를 참조한다
file	예전에 legacy 파일 file 타입에 사용됨. 요즘에는 거의 사용되지 않음
forward	함수가 사전 선언 forward declaration 되고 있다고 표시한다
goto	코드의 특정 위치에 붙여 놓은 레이블로 이동. 권장하지 않음.
index	프로퍼티 property 에 인덱스 index 를 지정할 때 사용. 그리고 함수를 가져오거나 내보낼 때 (드물게) 사용.
label	goto 문이 찾아올 수 있도록 레이블을 정의. 사용하지 말 것
message	플랫폼 메시지와 연계되는 가상 virtual 함수용 대체 키워드
name	외부 함수와 매핑하는 데 사용된다
nodefault	프로퍼티 property 에 기본값 default value 이 없음을 표시한다
on	예외를 발동 trigger 하는 데 사용된다
out	var의 대안, 참조로 전달 passed by reference 된다는 점은 동일하지만, 초기화가 되지 않은 상태로 전달되는 파라미터를 표시한다
packed	레코드 record 또는 데이터 구조의 메모리 배치 layout 를 변경한다
reintroduce	가상 virtual 함수의 이름을 재사용 reuse 할 수 있도록 한다
requires	패키지 package 에서 종속 dependent 패키지를 표시한다

위 오브젝트 파스칼 언어 키워드 중, 최근 몇 년 동안 추가된 것이 거의 없다는 점을 눈여겨보자. 키워드가 새로 추가되면 컴파일 에러가 생길 가능성이 있다는 점을 고려하기 때문이다. 즉, 키워드를 새로 추가하면 이미 그 이름으로 된 심볼 [symbol](#) 을 사용하고 있는 기존 프로그램들에게 영향을 줄 수 있다. 최근에 추가된 언어 기능

대부분은 새로운 키워드를 도입하지 않았다. 제네릭 `generic`, 익명 메서드 `anonymous method`도 그렇다.

프로그램의 구조 `Structure`

맨 앞에서는, 파일 하나만으로 간단한 콘솔 애플리케이션을 만들었다. 하지만 실제로 파일 하나 안에 모든 코드를 작성하는 경우는 거의 없다. 시각적 `visual` 애플리케이션을 만드는 경우에는, 프로젝트 `project` 파일 외에 적어도 보조 `secondary` 소스 코드 파일이 하나 이상 생성된다. 이런 보조 파일을 유닛 `unit` 이라고 한다. 확장자는 PAS(Pascal 소스 유닛)이다. 이와 달리, 중심 파일인 프로젝트 파일은 확장자가 DPR (Delphi PProject 파일)이다. 두 파일 모두 그 안에는 오브젝트 파스칼 소스 코드가 들어 간다.

오브젝트 파스칼은 유닛 `unit` 즉 프로그램 모듈들을 폭넓게 활용한다. 사실, 유닛들을 가지고도 모듈화 `modularity`와 캡슐화 `encapsulation`를 제공할 수 있다. 오브젝트를 사용하지 않고도 말이다. 유닛은 네임스페이스 `namespace`처럼 작용한다. 대체로 오브젝트 파스칼 애플리케이션은 여러 개의 유닛들로 구성된다. 폼 `form`을 담은 유닛과 데이터 모듈 `data module`을 담은 유닛도 여기에 해당된다. 실제로 프로젝트 안에 새 폼을 하나 추가하면, IDE는 실제로 새 유닛을 추가한다. 그리고 새 폼의 코드가 이 유닛 안에 정의된다.

유닛은 폼을 정의할 때만 사용되는 것이 아니다. 유닛 안에 그저 루틴 `routine`들만 모아 두거나 또는 (클래스 등) 데이터 타입을 하나 이상 담아서 정의해 두고 사용할 수도 있다. 프로젝트에서 빈 유닛을 새로 추가하면, 그 유닛 안에는 아래와 같이 구역을 나누는 키워드들만 들어 있을 것이다.

```
unit Unit1;

interface

implementation

end.
```

간단한 유닛 구조는 위와 같다. 다음 요소들이 이 안에 있다.

- 첫째, 유닛에는 고유한 이름이 있어야 한다. 이 이름은 파일 이름과 일치해야 한다 (즉, 위의 예제 유닛은 `Unit1.pas` 파일에 저장되어야 한다).
- 둘째, 유닛에는 `interface` 구역이 있다. 다른 유닛에게 보이는 것들을 선언한다.
- 셋째, 유닛에는 `implementation` 구역이 있다. 여기에는 구현 `implementation` 세부 사항, 실제 코드가 들어간다. 또한 로컬/지역 `local/지역` 선언들이 들어가는데, 이것들은 이 유닛 바깥에서 보이지 않는다.

유닛과 프로그램의 이름 Unit and Program Names

앞서 언급했듯이, 유닛 이름은 그 유닛의 파일 이름과 일치해야 한다. 프로그램 역시 마찬가지다. 유닛 이름을 바꾸려면 프로젝트 매니저에서 Rename 이름 바꾸기 옵션을 써야 한다. 그러면 그 두 개가 동기화된다 (IDE 에서 Save As 다른 이름으로 저장 를 써서 유닛 이름과 그 파일 이름을 맞출 수 있지만, 이전 파일까지 디스크에 남는다). 물론 파일 시스템에서 파일 이름을 바꿔도 되지만, 그러면 이름을 바꾼 그 유닛 파일을 열어서 맨 앞의 선언을 직접 수정해 맞춰야 한다. 안 그러면, 유닛이 컴파일 될 때(또는 IDE 가 그 파일을 적재할 때) 에러가 표시된다. 다음은 유닛 선언에서 이름을 변경했지만, 파일 이름을 그에 맞게 업데이트 하지 않은 경우에 표시되는 에러 메시지의 예시다.

[DCC Error] E1038 Unit identifier 'Unit3' does not match file name

유닛 이름과 그 유닛의 파일 이름이 일치해야 한다는 메시지다. 즉 유닛과 프로그램은 그 이름이 반드시 유효한 파일 이름이면서, 동시에 유효한 파스칼 식별자 identifier 이어야 한다는 것이다. 예를 들어, 이름에 공백이 들어갈 수 없다. 또한 밑줄(_) 말고 어떠한 특수 문자도 들어갈 수 없다. 그 점은 이미 앞에서 식별자 identifier 를 다룰 때 설명했다. 유닛과 프로그램의 이름은 반드시 오브젝트 파스칼 식별자이어야 하기 때문에, 파일 이름으로도 자동으로 유효하게 된다. 따라서 크게 걱정할 필요가 없다. 물론 유니코드 기호를 써서 이름을 만들면 파일 이름으로 유효하지 않을 것이다.

점으로 연결되는 유닛 이름

유닛 식별자를 만드는 기본 규칙에서 확장된 규칙이 있다. 유닛 이름은 점 표기법 dotted notation 을 사용할 수 있다. 따라서 아래에 있는 유닛 이름은 모두 유효하다.

```
unit1
myproject.unit1
mycompany.myproject.unit1
```

일반 규칙에 따라, 유닛 이름이 점으로 구분되어 있으면, 파일 이름도 그래야 한다 (MyProject.Unit1 유닛은 *MyProject.Unit1.pas* 파일에 저장).

이렇게 확장된 이유는, 유닛 이름은 반드시 고유 unique 해야 하는데, 계속해서 더 많은 유닛들이 엠바카데로와 써드-파티 공급사에서 의해 제공되었기 때문이다. 그만큼 더 점점 더 복잡해졌다. 델파이 라이브러리에 들어있는 모든 RTL 유닛들과 다양한 기타 유닛들은 각 영역을 표기하는 접두사와 점을 사용하는 규칙을 따른다.

- System: 핵심 core RTL용
- Data: 데이터베이스 액세스 및 기타 관련
- FMX: 파이어몽키 FireMonkey 플랫폼용. 파이어몽키는 단일-소스 멀티-디바이스 아키텍처이다. 데스크탑과 모바일용으로 사용된다.
- VCL: 윈도우 Windows에서 사용되는 비주얼 컴포넌트 라이브러리용

참고 라이브러리 유닛 등, 점으로 구분된 유닛 이름을 참조할 때에는 대체로 전체 이름을 적지만, 전체 이름 중 마지막 부분만 적어도 된다 (이전 코드와 호환성을 지키기 위함). 방법은 프로젝트 옵션 중 "Unit scope names" 설정에 세미콜론으로 구분하여 항목들을 넣는 것이다. 하지만 이 경우, 유닛의 완전한 [fully qualified](#) 이름을 사용할 때보다 컴파일 속도가 느려진다는 점에 유의하자.

유닛의 구조를 자세히 알아보기 [More on the Structure of a Unit](#)

interface, implementation 구역 외에, 원할 때만 넣을 수 있는 [optional](#) 구역들이 있다. initialization 구역에 시작 코드를 넣을 수 있다. 그러면 그 프로그램이 메모리에 맨 처음 적재될 때 실행된다. 만약 initialization 구역이 있다면, finalization 구역을 추가할 수 있다. 그 구역에는 프로그램 종료 시 실행할 코드를 넣는다.

참고 여러분은 초기화 [initialization](#) 코드를 클래스 생성자 [constructor](#) 안에 넣어도 된다. 최근에 추가된 언어 기능이며 12 장에서 설명한다. 클래스 생성자를 사용하면 링커 [linker](#)가 불필요한 코드를 제거하는데 도움이 된다. 따라서 initialization 구역과 finalization 구역보다는 클래스 생성자 [constructor](#)와 클래스 소멸자 [destructor](#)를 사용하는 것을 권장한다. 역사적 일화 정도로 말하자면, initialization 키워드 대신 begin 키워드를 사용하는 것도 컴파일러가 여전히 지원한다. 프로젝트 소스 코드 안에서는 begin을 그렇게 사용하는 것이 여전히 표준이다.

가능한 모든 구역과 샘플 요소를 넣어서 유닛의 일반적인 구조를 보면 다음과 같다.

```
unit UnitName;

interface

// interface 구역에서 참조하는 다른 유닛들
uses
    UnitA, UnitB, UnitC;

// 내보내기 되는 타입들 정의
type
    NewType = TypeDefinition;

// 내보내기 되는 상수(constant)들
const
    Zero = 0;

// 글로벌 global/전역 변수(Global variable)들
var
    Total: Integer;

// 내보내기 되는 함수(function)들과 프로시저(procedure)들
procedure MyProc;

implementation

// implementation 구역에서 참조하는 다른 유닛들
uses
    UnitD, UnitE;
```



```

// 로컬local/지역 유닛 타입 (Local unit type)들
type
  NewType2 = TypeDefinition;

// 로컬local/지역 유닛 상수 (Local unit constant)들
const
  One = 1;

// 로컬local/지역 유닛 변수 (Local unit variable)들
var
  PartialTotal: Integer;

// 모든 함수와 프로시저를 위한 코드 (로컬local과 및 내보내기export 모두 해당됨)
procedure MyProc;
begin
  // ... MyProc 프로시저를 위한 코드
end;

initialization
  // 초기화 코드 (Optional)

finalization
  // 정리 코드 (Optional)

end.

```

유닛 안에 있는 인터페이스 [interface](#) 구역의 목적은 그 유닛을 사용하고자 하는 메인 프로그램과 다른 유닛들에게 그 유닛에 포함된 것이 무엇인지 세부 사항을 제공하는 것이다. 반면에, 구현 [implementation](#) 구역에는 그 유닛의 바깥에서는 보이지 않는 볼트와 너트들이 들어간다. 이것은 오브젝트 파스칼이 클래스와 오브젝트를 사용하지 않고도 소위 캡슐화 [encapsulation](#)를 제공하는 방법이다.

보다시피, 유닛의 [interface](#) 구역에는 프로시저, 함수, 글로벌^{전역} 변수, 데이터 타입 등 다양한 요소가 선언될 수 있다. 그 중 가장 많이 선언되는 것은 데이터 타입이다. 우리가 시각적 폼 [form](#)을 생성할 때마다, IDE는 자동으로 새 클래스 데이터 타입이 들어간 새 유닛을 만든다. 오브젝트 파스칼에서 유닛은 폼 [form](#)을 정의할 때만 쓰는 것이 아니다. 함수와 프로시저를 담은(전통적인 방식) 유닛, 폼이나 기타 시각적 요소들을 참조하지 않고 클래스들만 담은 유닛 등 코드만 있는 유닛도 만들 수 있다.

[interface](#) 구역 또는 [implementation](#) 구역 안에서, [type](#) ^{타입}, [var](#) ^{변수}, [const](#) ^{상수} 등을 선언할 때는 순서에 관계없이 배치할 수 있으며, 여러 번 반복해도 된다. 상수 몇 개, 타입 몇 개, 상수 몇 개 더, 변수 몇 개, 그리고 다시 타입 몇 개를 배치해도 된다. 규칙 하나다. 심볼 [symbol](#)을 참조하려면, 그 곳보다 더 앞쪽에 그 심볼이 선언되어 있어야 한다. 그래서 같은 구역이 여러 개까지 필요한 경우가 종종 생긴다.

Uses 절 [Clause](#)

interface 구역의 맨 앞에는 Uses 절 [Clause](#) 이 있다. 이 문장은 그 유닛의 인터페이스 구역에서 액세스해야 하는 다른 유닛들이 무엇인지 나열한다. 여기에는 그 유닛에 속한 데이터 타입을 정의하는 코드에서 참조하고 있는 유닛도 포함된다. 예를 들어, 폼 [form](#) 을 정의하는 코드에서 사용되는 컴포넌트 [component](#) 들도 여기에 나열한다.

Uses 절은 implementation 구역의 시작 부분에 한 번 더 나온다. 여기에는 그 유닛의 구현 [implementation](#) 부에 있는 코드에서만 액세스하는 유닛들을 추가로 나열한다. 루틴 [routine](#) 및 메서드 [method](#) 코드 안에서 다른 유닛을 참조해야 하는 경우에는, 첫 번째 uses 절이 아니라 두 번째 uses 절에 유닛을 적어야 한다. 그래야 종속성 [dependency](#) 을 낮추고 컴파일 시간을 줄일 수 있다. 참조 [refer](#) 하려는 모든 유닛은 프로젝트가 위치한 디렉토리 안에 있거나 또는 검색 경로 [Search Path](#) 에 정의된 디렉토리 안에 있어야 한다.

팁 프로젝트에서 사용하는 검색 경로 [Search Path](#) 지정은 Project Options에서 한다. 그런데, 시스템은 라이브러리 경로 [Library Path](#)에서도 유닛들을 찾는다. 이것은 IDE 수준의 전역 설정이다.

C++ 프로그래머가 유의할 점이 있다. uses 문은 include 지시어와 같은 것이 아니다. uses 에 유닛을 나열하면, (미리 컴파일 된) 유닛의 interface 부분만 받아오는 [import](#) 효과가 있다. implementation 부분은 그 유닛이 컴파일 될 때 고려하게 된다. 참조되는 파일은 소스 코드 형식(PAS)과 컴파일 된 형식(DCU) 파일 모두 가능하다.

거의 사용되지는 않지만. 오브젝트 파스칼에는 \$INCLUDE 라는 컴파일러 지시어 [compiler directive](#) 도 있다. 그 동작은 C/C++ 인클루드 [INCLUDE](#) 와 비슷하다. 하지만 그 대상은 소스 코드 보다는 라이브러리들이 사용하는 특별한 인클루드 [include](#) 파일인 경우가 많다 (파일 확장자는 대체로 INC 다). 예를 들어, 일부 라이브러리들은 이 파일에 컴파일러 지시어들이나 기타 설정들을 담아 놓고, 여러 유닛들에 공유할 때 사용한다. 컴파일러 지시어에 대해서는 이 장의 마지막에 가서 간단히 다룬다.

경고 오브젝트 파스칼에서 컴파일 된 유닛을 사용하려면, 반드시 버전이 동일한 컴파일러와 시스템 라이브러리로 빌드 된 유닛들만 호환된다는 점에 유의하자. 이전 버전에서 컴파일 된 유닛은 일반적으로 이후 버전의 컴파일러와 호환되지 않는다. 하지만 릴리스가 같은 업데이트끼리는 호환성을 유지한다. 즉, 10.3.1 버전에서 빌드 된 유닛은 10.3.x 버전과 호환되지만, 10.2 또는 10.4 버전과는 호환되지 않는다. 델파이 11 버전은 버전 관리 방식이 변경되었다. 이제 11.x 업데이트는 모두 버전 11과 호환성을 유지한다. 하지만 버전 12와는 호환되지 않을 것이다.

유닛Unit 과 범위Scope

오브젝트 파스칼에서 유닛 [Unit](#) 은 캡슐화 [encapsulation](#) 와 가시성 [visibility](#) 의 핵심이다. 그런 의미에서, 클래스에 있는 private 또는 public 키워드보다 훨씬 더 중요할 수 있다. 식별자 [identifier](#) (예: 변수, 프로시저, 함수, 데이터 타입)의 범위 [scope](#) 란 코드 안에서 그 식별자를 볼 수 있거나 액세스할 수 있는 영역을 말한다.

식별자 **identifier**는 오직 해당 범위 **scope**, 즉 자신이 선언된 유닛, 함수, 프로시저 안에서만 의미가 있다. 이 기본 규칙에 따라, 식별자는 해당 범위 밖에서는 사용되지 못한다.

참고 최근까지, 오브젝트 파스칼에는, 일반 코드 블록 안에서 선언 **declaration**을 하고 그 범위를 그 블록 안으로만 국한하는 개념이 없었다. 델파이 10.3부터는 인라인 변수 **inline variable**를 **begin-end** 블록 안에 선언할 수 있다. 그 변수는 범위가 해당 블록 안으로 제한된다. 이는 C나 C++의 방식과 같다. 자세한 내용은 2장에 있는 "변수의 수명 및 가시성" 부분에서 설명한다.

일반적으로, 식별자는 정의된 **defined** 위치 이후에만 보인다. 이 언어에는 식별자를 그 완전한 정의 **definition** 보다 앞쪽에 사전 선언 **declaration** 할 수 있는 기술이 있다. 그래도, 이 일반 규칙은 여전히 적용된다. 정의와 선언을 둘 다 고려한다면 그렇다.

전체 프로그램을 파일 하나 안에 작성하는 것이 거의 말도 안 된다는 점을 감안할 때, 유닛 여러 개를 사용하게 될 텐데, 이 경우 위 규칙은 어떻게 달라질까? 간단히 말해, **uses** 문에 유닛들을 나열해 참조하는 경우, 나열된 유닛들의 인터페이스 구역에 있는 식별자들이 모두 보이게 된다.

뒤집어 말하면, 식별자(타입, 함수, 클래스, 변수 등)를 유닛의 **interface** 구역에 선언하면, 그 유닛을 참조하는 다른 모든 모듈들이 그 식별자를 볼 수 있다. 한편, 식별자를 **implementation** 구역에 선언하면 그 식별자는 오직 그 유닛 안에서만 사용할 수 있다(일반적으로 **로컬 식별자** **local identifier**라고 부른다).

유닛 **Unit** 을 네임스페이스 **Namespace** 처럼 사용하기

uses 문이라는 표준 기법을 통해, 다른 유닛의 범위 안에 선언된 식별자에게 접근할 수 있다는 것을 알게 되었으니, 우리는 다른 유닛의 정의 **definition** 에 접근할 수 있다. 그런데, 참조하는 두 유닛 안에 있는 식별자들이 똑 같은 이름을 쓰고 있을 수도 있다. 즉 이름이 같은 클래스 또는 루틴이 두 개인 경우가 생길 수 있다.

이런 경우에는, 유닛의 이름을 접두사로 사용하면 된다. 그래서 유닛에 정의된 타입이나 루틴의 이름 앞에 붙이면 된다. 예를 들어 **ComputeTotal** 프로시저가 **Calc** 유닛에 정의되어 있다면, **Calc.ComputeTotal** 로 참조하면 된다. 이렇게 해야 하는 경우는 많지 않다. 왜냐하면, 여러분은 프로그램을 작성할 때 동일한 식별자 이름을 서로 다른 모듈에서 사용하지 말고 가능한 피하라고 배웠을 것이기 때문이다.

하지만 델파이 시스템 또는 써드-파티 라이브러리를 보다 보면 이름이 같은 함수와 클래스들이 있을 것이다. 예를 들면, 사용자 인터페이스 프레임워크들 안에 있는 시각적 컨트롤들의 이름이 그와 비슷한 다른 프레임워크에서도 사용되는 경우가 있다. 이런 경우, **TForm** 또는 **TControl** 을 참조하는 코드는, 그것이 현재 실제로 무슨 유닛을 참조하고 있는지에 따라 가리키는 클래스가 달라질 것이다.

동일한 식별자를 노출하는 서로 다른 두 유닛이 **uses** 문 안에 나열되면, 맨 뒤에 적힌 유닛 안의 심볼 **symbol** 이 덮어쓰고 컴파일러는 그 심볼을 사용한다. 즉, 목록에서

맨 뒤에 적힌 유닛 안에 정의된 심볼이 우선이다. 이런 상황을 여러분이 피할 수 없다면, 심볼 앞에 유닛 이름을 접두사로 붙여서 코드를 작성하는 것이 좋다. 그러면, 유닛 나열 순서에 따라 다르게 작동하는 일이 없다.

참고 델파이 개발자들은 인터포저 클래스 `interposer class`라는 기법을 통해서, 이름이 같은 클래스 두 개를 가질 수 있다. 이 기술은 책의 뒷부분에서 설명한다.

프로그램 파일

앞에서 본 것처럼, 델파이 애플리케이션을 구성하는 소스 코드 파일은 두 종류다. 유닛 파일들(하나 이상)과 프로그램 파일(오직 하나다. DPR 파일에 저장된다)이다. 유닛들은 보조 `secondary` 파일로 간주된다. 즉 애플리케이션의 메인 `main` 부분인 프로그램 파일이 유닛 파일들을 참조하는 형태다. 이론으로 그렇다. 하지만, 실제로는 대체로 프로그램 파일은 자동 생성된다. 그리고 역할도 제한적이다. 프로그램 파일은 그저 프로그램 시작하기 정도만 담당한다. 시각적 애플리케이션에서는 메인 폼을 생성하고 실행하는 것만 하는 것이 일반적이다. 여러분은 프로그램 파일의 코드를 직접 편집할 수 있다. 하지만, IDE의 Project Options에서 옵션(예: 애플리케이션 오브젝트와 폼들에 관련된 설정들)을 적용하면 프로그램 파일의 해당 코드에 자동으로 반영된다.

일반적으로, 프로그램 파일의 구조는 유닛의 구조보다 훨씬 더 간단하다. 아래에 있는 샘플 프로그램 파일에서 소스 코드를 보자(옵션 사항인 표준 유닛들은 생략). 여러분을 위해 IDE가 자동으로 만드는 코드다.

```
program Project1;

uses
    FMX.Forms,
    Unit1 in 'Unit1.PAS' {Form1};

begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

보다시피 단순하다. 애플리케이션의 `uses` 구역과 메인 코드(`begin` 과 `end` 키워드로 둘러싸여 있음)만 있다. 프로그램에 있는 `uses` 문은 특히 중요하다. 애플리케이션의 컴파일 작업 `compilation`과 링크 작업 `linking`를 관리하는 데 사용되기 때문이다.

팁 프로그램 파일의 코드에 나열되는 유닛들은 IDE 프로젝트 매니저 안에서 그 프로젝트 아래에 나열된 유닛들과 대응한다. IDE에서 여러분이 유닛을 새로 프로젝트에 추가하면 프로그램 파일 소스의 목록에 그 유닛이 자동으로 추가된다. 마찬가지로 프로젝트에서 유닛을 제거하면 소스에서도 제거된다. 어떤 경우이든, 프로그램 파일의 소스 코드를 여러분이 수작업으로 편집하면 프로젝트 매니저 안에 있는 유닛 목록 역시 그에 맞추어 업데이트된다.

컴파일러 지시어 Compiler Directives

프로그램 구조의 또 다른 (실제 코드가 아닌) 특수 요소는 컴파일러 지시어 [Compiler Directive](#) 다. 이는 컴파일러를 위한 특수 명령어다. 그 형식은 다음과 같다.

| `{ $X+ }`

컴파일러 지시어 중 몇 가지는 위와 같이 단일 문자 [character](#) 로 되어 있다. 그 뒤에는 더하기 또는 빼기 기호를 붙여서 그 지시어가 활성화인지 비활성인지를 표시한다. 컴파일러 지시어 대부분은 더 길고 더 읽기 쉬운 버전도 가지고 있다. 그리고 그 뒤에 ON 과 OFF 를 사용하여 활성화 여부를 표시한다. 오직 더 길고 더 잘 이해되는 형식만 가진 지시어들도 일부 있다.

컴파일러 지시어는 컴파일 된 코드 안에 직접 들어가지 않는다. 하지만, 컴파일러 지시어가 있는 지점 뒤부터 컴파일러가 어떻게 코드를 생성하는지를 지시한다. 컴파일러 지시어를 코드에 적어 넣는 경우의 대부분은 IDE 에서 Project Options 를 열고 컴파일러 설정을 변경해도 같은 효과를 낼 수 있다. 하지만, 컴파일러 설정을 특정 유닛에만 또는 일부 코드 조각 안에만 적용하려는 상황에서는 코드가 필요하다.

특정 컴파일러 지시어들은 그 지시어가 영향을 줄 수 있는 언어 기능을 설명하는 곳에서 필요에 맞게 설명하겠다. 지금은 프로그램 코드 흐름과 관련된 몇 가지 지시어, 즉 조건부 정의 [conditional define](#) 와 인클루드 [Include](#) 에 대해서만 보자.

조건부 정의 Conditional Define

조건부 정의 [Conditional Define](#) 는 소스 코드의 일부를 포함하거나 무시하도록 컴파일러에게 지시한다. 델파이에 있는 조건부 정의는 두 종류다. `$IFDEF` 와 `$IFNDEF` 라는 전통적인 정의 그리고 보다 더 새롭고 유연한 `$IF` 정의가 있다.

조건부 정의는 정의된 심볼 [defined symbol](#) 에 의해 결정된다. `$IF` 버전에서는 상수 값 [constant value](#) 에 의해 결정된다. 정의된 심볼들은 (컴파일러 심볼, 플랫폼 심볼 등) 시스템에서 미리 정의해 놓은 것들이 있다. 여러분은 특정 프로젝트 옵션 안에서 정의할 수 있다. 또는 코드 안에서 컴파일러 지시어인 `$DEFINE` 을 사용해 정의할 수도 있다.

기존의 `$IFDEF` 와 `$IFNDEF` 의 형식은 다음과 같다.

```
{ $IFDEF TEST}
  // 이 부분은 TEST가 정의되어 있는 경우에만 컴파일 된다.
{ $ENDIF}

{ $IFNDEF TEST}
  // 이 부분은 TEST가 정의되어 있지 않은 경우에만 컴파일 된다.
{ $ENDIF}
```

여러 조건이 있는 경우 `$ELSEIF` 지시어를 사용할 수도 있다.

컴파일러 버전들

델파이 컴파일러에는 각 버전 별 정의 `define` 가 지정되어 있다. 개발자는 이 정의를 이용하여 컴파일러의 델파이 버전을 확인할 수 있다. 만약 최근에 도입된 새 기능을 사용하면서 이전 버전에서 작성한 코드 역시 여전히 컴파일 되도록 하고 싶을 때에도 이 정의를 활용할 수 있다.

델파이 최근 버전들에 맞춰진 특정 코드를 사용해야 한다면, `$IFDEF` 문을 작성하면 된다. 아래의 정의를 바탕으로 한다.

Delphi 2007	VER180
Delphi XE	VER220
Delphi XE2	VER230
Delphi XE4	VER250
Delphi XE5	VER260
Delphi XE6	VER270
Delphi XE7	VER280
Delphi XE8	VER290
Delphi 10 Seattle	VER300
Delphi 10.1 Berlin	VER310
Delphi 10.2 Tokyo	VER320
Delphi 10.3 Rio	VER330
Delphi 10.4 Sydney	VER340
Delphi 11 Alexandria	VER350

버전에서 숫자 부분은 실제 컴파일러 버전(예: 26 은 델파이 XE5 용)을 나타낸다. 이 숫자들의 순서는 델파이에만 국한된 것은 아니며, 볼랜드에서 발표한 첫 터보 파스칼 컴파일러까지 거슬러 올라간다(자세한 내용은 부록 A 참조).

또한, 내부 버전 관리 상수 `constant` 를 `$IF` 문에서 사용할 수도 있다. 이 방식은 비교 연산자(`>=`)를 사용할 수 있기 때문에 특정 버전 일치 여부만 확인하는 방식보다 장점이 있다. 버전 관리 상수의 이름은 `CompilerVersion` 이다. 다음 예제는 델파이 XE5에 해당하는 상수 값이 부동 소수점 값 26.0으로 지정되어 있다는 점을 활용하고 있다.

```
{ $IF CompilerVersion >= 26 }  
  // 이 부분에 있는 코드는 Delphi XE5 또는 그 이후 버전에서만 컴파일 된다  
{ $IFEND }
```

마찬가지로, 각 플랫폼별 시스템 정의 `system define` 를 사용하면, 특정 플랫폼을 위해 맞춤 작성한 코드 부분이 상황에 맞게 컴파일되도록 할 수 있다 (오브젝트 파스칼에서는 이렇게 해야 하는 경우가 거의 예외적으로 있을 뿐, 일반적인 관행은 아니다).

윈도우Windows

MSWINDOWS

맥OS ^{macOS}	MACOS
iOS	IOS
안드로이드 ^{Android}	ANDROID
리눅스 ^{Linux}	LINUX

위에 있는 플랫폼 정의 `platform define` 를 기반으로 조건 테스트를 하는 예제 코드 조각은 다음과 같다. 이 코드는 HelloPlatform 프로젝트 안에 있다.

```
{ $IFDEF IOS}
  ShowMessage( ' 실행되고 있는 플랫폼은 ios입니다' );
{ $ENDIF}

{ $IFDEF ANDROID}
  ShowMessage( ' 실행되고 있는 플랫폼은 안드로이드입니다' );
{ $ENDIF}
```

인클루드 파일들 Include Files

지시어 `directive` 를 하나 더 보자. `$INCLUDE` 다. 앞에서 `uses` 문을 설명하면서 이미 언급했었다. `$INCLUDE` 지시어를 사용하면 소스 코드 한 뭉치를 소스 코드 파일 안 원하는 위치에 참조하고 넣을 수 있다. 가끔은 동일한 코드 조각을 여러 유닛들 안에 포함시키고 싶을 때가 있을 것이다. 예를 들면, 컴파일러 지시어와 기타 요소들을 정의해 놓은 코드 뭉치 하나를 여러 유닛들 안에 넣어서 컴파일러가 그것들을 직접 사용하도록 할 수 있다.

`uses` 문에 나열된 유닛은 오직 한 번만 컴파일 된다. 하지만, `include` 를 사용하면 해당 코드는 자신을 인클루드 하는 모든 유닛마다 포함되어 컴파일 된다 (따라서 인클루드 파일 안에서는 새 식별자를 선언하지 않는 것이 좋다. 같은 프로젝트에서 여러 유닛들 안에 심을 `embedded` 생각이라면 말이다). 그렇다면 인클루드 파일은 어떻게 사용할까? 인클루드 파일로 활용하기 좋은 예는 유닛들 대부분에서 활성화하고 싶은 컴파일러 지시어 세트 또는 부가적인 특수한 정의 `define` 들을 담은 파일이다.

대형 라이브러리들은 인클루드 파일을 종종 이런 목적으로 사용한다. 한 가지 예로 FireDAC 라이브러리를 들 수 있다. 데이터베이스 라이브러리인데, 현재 델파이 시스템 라이브러리들 중 하나다. 또 다른 예로 델파이의 RTL 시스템 유닛들이 있다. 이 유닛들은 그 안에 `$IFDEF` 와 해당 플랫폼에 맞는 인클루드 파일들을 나열해 놓았다. 그래서 컴파일러는 그 파일들 중에서 해당 조건에 맞는 파일 하나만 포함시킨다.

02: 변수variable와 데이터 타입data type

오브젝트 파스칼 Object Pascal 은 타입을 강하게 지정하는 strongly-typed 언어로 알려져 있다. 오브젝트 파스칼에서, 변수 variable 는 데이터 타입 data type (또는 사용자-정의 데이터 타입) 하나를 담을 수 있도록 선언된다. 변수의 타입은 그 변수에 어떤 값을 담을 수 있고 어떤 연산을 수행할 수 있는지를 결정한다. 이를 통해 컴파일러는 코드의 오류를 식별하고 더 빠르게 작동하는 프로그램을 생성할 수 있다.

이것이 바로 C 또는 C++ 같은 언어보다 파스칼의 타입 개념이 더 강력한 이유다. 그 후에 나온 언어들은 C 와 동일한 문법 syntax 을 기반으로 하지만 C 와의 호환성을 켜다. C#, Java 같은 언어는 C 에서 벗어나 파스칼의 강력한 데이터 타입 개념을 수용했다. 예를 들어, C 에서는 산술 arithmetic 데이터 타입끼리 거의 상호 교환이 가능하다. 이와 대조적으로 BASIC 의 원래 버전에는 이와 비슷한 개념이 없었다. 그리고 오늘날의 많은 스크립트 언어(JavaScript 가 대표적인 예)들은 데이터 타입 표기가 매우 다르다.

참고 실제로, 타입 안전성을 우회하는 몇 가지 트릭이 있다(배리언트 레코드 variant record 타입 등). 이 트릭들은 절대로 사용하지 않도록 권장되어 왔다. 그리고 오늘날에는 잘 사용되지도 않는다.

앞서 언급했듯이 자바스크립트 JavaScript 이후의 모든 동적 언어들은 데이터 타입에 대한 개념이 같지 않거나 (적어도) 타입에 대한 개념이 매우 느슨하다. 이러한 언어 중 몇 가지는 변수에 할당되는 값 value 으로부터 타입을 유추 infer 한다. 그리고, 변수는 타입이 계속해서 변할 수 있다. 중요한 점이 있다. 데이터 타입은 대규모 애플리케이션의 정확성을 컴파일-시간에 강제할 수 있는 핵심 요소다. 실행-시간 검사에 의존하지 않는다. 데이터 타입이 강력하면 더 많은 질서 order 와 구조 structure , 작성하려는 코드에 대한 사전 계획이 필요하다. 그 장단점은 분명히 존재한다.

참고 말할 필요도 없지만, 나는 타입이 강력하게 지정되는 언어를 선호한다. 하지만, 이 책의 목표는

이 언어가 어떻게 작동하는지를 설명하는 것이지, 왜 이것이 훌륭한 프로그래밍 언어라고 생각하는지를 주장하는 것이 아니다.

변수Variable와 할당Assignment

타입을 강하게 지정하는 다른 언어들이 그렇듯, 오브젝트 파스칼에서도 모든 변수는 사용하기 전에 선언해야 `declare` 한다. 변수를 선언할 때는 데이터 타입을 명시해야 한다. 아래에서 몇 가지 변수 선언을 보자.

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

`var` 키워드는 프로그램 안 여러 위치에서 사용된다. 함수 `function` 나 프로시저 `procedure` 의 시작 부분에서 선언하면 그 변수 `variable` 는 그 함수나 프로시저의 코드 영역 안에 있는 로컬 `local/지역` 변수가 된다. 유닛 `unit` 안에 선언하면 글로벌 `global/전역` 변수가 된다.

참고 잠시 후에 살펴보겠지만, 델파이 최신 버전에는 변수를 "인라인`inline`"으로 선언할 수 있는 기능이 추가되었다. 인라인 변수는 프로그래밍 문장`statement` 안에 끼어 들어갈 수 있다. 이는 고전적인 파스칼에서 크게 벗어난 것이다.

`var` 키워드 뒤에는 변수 이름들을 나열한다. 그 뒤에는 콜론과 데이터 타입 이름을 적는다. 한 줄에 변수 이름을 하나 이상 써도 된다. 위 예문 마지막 문장은 A 와 B 가 한 줄에 적혀 있다 (이런 코딩 스타일은 오늘날 그다지 일반적이지 않다. 요즘에는 변수 이름이 두 개면 두 줄에 나누어 적는 것이 일반적이다. 줄을 나누어 적으면 가독성 `readability` 이 좋고, 버전 `version` 을 비교 `comparison` 또는 병합 `merging` 할 때 도움이 된다).

타입을 명시하여 변수를 정의하고 나면, 그 변수의 데이터 타입에서 지원하는 연산만 수행할 수 있다. 예를 들어, 불리언 `Boolean` 변수는 테스트 안에, 인티저 `Integer/정수` 변수는 숫자 표현식 `numerical expression` 안에서 사용할 수 있다. 불리언 변수와 인티저 변수를 섞어 쓰려면 반드시 알맞은 변환 `conversion` 함수를 호출해야 한다. 즉 호환되지 않는 데이터 타입을 함께 섞어 쓰지 못한다 (불리언 변수와 인티저 변수의 경우처럼 내부 표현이 물리적으로 호환이 가능한 경우라도 그렇다)

가장 간단하게 할당 `assignment` 을 하는 방법은 실제 값을 넣는 것이다. 위 예제에서라면 `Value` 변수에 숫자 값 10 을 담고 싶을 수 있다. 이 리터럴 `literal` 값을 개발자는 어떻게 표현할까? 그 개념 자체는 명확하다. 하지만, 세부 사항을 알면 좋다.

리터럴 값 Literal Value

리터럴 `Literal` 값은 개발자가 프로그램 소스 코드에 직접 입력하는 값이다. 값이 20 인 숫자가 필요하다면 그냥 적으면 된다.

```
| 20
```

동일한 숫자를 10 진수가 아니라 16 진수 `hexadecimal` 값으로 표현할 수도 있다.

```
| $14
```

델파이 `Delphi` 11 부터는 이 숫자를 2 진수 `binary` 값으로 표현할 수도 있다.

```
| %10100;
```

값이 매우 크면 밑줄을 숫자 구분 기호로 사용할 수 있다. 그러면 그 상수 `constant` 값을 더 읽기가 더 쉽다. 컴파일러는 이 구분 기호를 무시한다. 델파이 11 부터 추가된 기능이다. 예를 들어 값이 2 천만이라면 다음과 같이 쓸 수 있다.

```
| 20_000_000
```

이것들은 정수 (또는 기타 다양한 정수 타입)를 적는 리터럴이다. 만약 동일한 숫자가 필요하지만, 부동 소수점 타입으로 리터럴을 적고 싶다면, 소수점 0 을 추가한다.

```
| 20.0
```

리터럴 값은 숫자만 쓸 수 있는 것이 아니다. 문자 `character` 와 문자열 `string` 도 쓸 수 있다. 두 타입 모두 작은따옴표로 에워싼다. (다른 프로그래밍 언어에서는 두 타입 모두 큰따옴표로 에워싸거나, 또는 문자는 작은따옴표, 문자열은 큰따옴표를 사용한다).

```
// 리터럴 문자
'K'
#55
```

```
// 리터럴 문자열
'Marco'
```

위에서 볼 수 있듯, 문자를 표현할 때 그에 해당하는 숫자 값(원래는 ASCII 숫자 값, 현재는 유니코드 `Unicode` 코드 포인트 `code point` 값)을 사용할 수도 있다. 이때는 숫자 값 앞에 # 기호를 붙인다. 예를 들어, #32 는 공백 문자 `space character` 다. 이처럼 소스 코드에서 텍스트로 표현할 수 없는 제어 문자 `control character`, 즉 백스페이스 `backspace` 나 탭 `tab` 같은 문자를 적을 때 유용하다.

문자열 `string` 안에 작은따옴표를 넣어야 하는 경우에는, 작은따옴표가 이스케이프 `escape` 처리되어 문자로 인식될 수 있게 작은 따옴표를 덧붙여야 한다 (작은따옴표 두 번 적기). 즉, 내 이름 맨 뒤에 작은 따옴표를 넣으려면 아래와 같이 쓴다.

```
| 'My name is 'Marco Cantu'''
```


맨 뒤에 있는 작은따옴표 세 개 중 앞 두 개는 문자열 작은따옴표 하나를 의미한다. 세번째 작은따옴표는 이 문자열이 끝났다는 표시다. 또한 문자열 리터럴 [string literal](#) 은 한 줄에 작성해야 하지만 + 기호를 써서 문자열 리터럴 여러 개를 연결할 수 있다(굵긴이: 델파이 12부터는 + 기호로 연결하지 않아도 리터럴을 여러 줄에 걸쳐서 작성할 수 있음). 문자열 안에 새 줄 즉 줄 바꿈을 넣으려면, 두 줄에 작성하는 것이 아니다. 앞 뒤 문자열 사이에 시스템 상수 [system constant](#) 인 `sLineBreak` 를 넣어서 연결한다 (플랫폼에 따라 다름).

```
'내이름은' + sLineBreak + '''마르코 칸투'''
```

할당 문 Assignment Statement

오브젝트 파스칼에서 할당 [assignment](#) 은 콜론 등호 연산자(:=)를 사용한다. 이것은 다른 언어에 익숙한 프로그래머에게 낯선 표기법이다. = 연산자는, 다른 언어에서는 할당 연산자로 사용되지만, 오브젝트 파스칼에서는 동일성 [equality](#) 테스트에 사용된다.

역사 := 연산자는 파스칼의 전신인 Algol에서 유래했다. 오늘날 개발자들은 거의 들어본 적도 없는 (사용해보지도 않은) 언어일 것이다. 오늘날 언어 대부분은 := 할당을 피하고, = 할당 표기법을 선호한다.

할당 [assignment](#) 과 동일성 테스트 [equality test](#) 에 사용되는 기호가 서로 다르기 때문에, 파스칼 컴파일러는, C 컴파일러처럼, 소스 코드를 더 빠르게 번역할 수 있다. 연산자가 어떤 의미로 사용되는 지를 문맥 [context](#) 을 조사하지 않아도 파악할 수 있기 때문이다. 또한 연산자가 구분되면 사람이 코드를 더 쉽게 읽을 수 있다. 파스칼과 달리, C 언어와 그 구문에서 파생된 언어(예: Java, C#, JavaScript)는 = 기호를 사용해 할당을 하고, 동일성을 테스트할 때는 == 기호를 사용한다.

참고 보다 완전하게 설명하자면, 자바스크립트에는 === 연산자 (타입과 값에 대해 엄격하게 동일성 테스트)도 있다. 하지만, 이 연산자는 자바스크립트 프로그래머들도 대부분 혼동하는 부분이다.

할당에서 양쪽 요소는 *lvalue* 와 *rvalue* 라고 부른다. 왼쪽-값은 할당을 받아들이는 변수 또는 메모리의 위치다. 오른쪽-값은 할당되어 들어가는 표현식 [expression](#) 의 값이다. *rvalue* 는 표현식일 수 있지만, *lvalue* 는 반드시 수정이 허용되는 메모리의 위치를 (직접 또는 간접적으로) 참조 [refer](#) 해야 한다. 그러나 데이터 타입 몇 가지는 할당 동작 [behavior](#) 이 독특하다. 이에 대해서는 뒤에서 다룰 것이다.

다른 규칙이 더 있다. *lvalue* 의 타입과 *rvalue* 의 타입은 반드시 일치하거나 또는 이 둘 사이에 반드시 자동 변환이 될 수 있어야 한다. 이것은 바로 이어서 설명한다.

할당Assignment과 변환Conversion

간단한 할당을 사용하여, 다음과 같은 코드를 작성할 수 있다(이 부분에 나오는 많은 코드 조각들은 VariablesTest 프로젝트에 있음).

```
Value := 10;
Value := Value + 10;
IsCorrect := True;
```

위 변수들을 선언된 앞 예문을 고려하면, 이 할당 세 가지는 모두 올바르다. 하지만, 아래 문장은 올바르지 않다. 두 변수의 타입이 서로 다르기 때문이다.

```
Value := IsCorrect; // 예러
```

위 코드를 입력하는 즉시, 델파이 에디터에는 오류를 알리는 빨간색 물결무늬와 함께 알맞은 설명이 나온다. 컴파일을 시도하면 컴파일러는 아래와 같은 에러를 일으킨다.

```
[dcc32 Error]: E2010 Incompatible types: 'Integer' and 'Boolean'
```

컴파일러는 잘못된 코드 즉, 데이터 타입 두 개가 서로 호환되지 않는 문제가 있다고 알려준다. 물론 변수 값을 한 타입에서 다른 타입으로 변환하기는 종종 가능하다. 어떤 때는, 자동으로 변환되기도 한다. 예를 들어, 정수 값을 부동 소수점 변수에 할당하는 경우가 그렇다 (물론 그 반대는 안됨). 하지만, 주로 여러분은 해당 시스템 함수를 호출해 그 데이터의 내부 표현을 바꿔주어야 할 것이다.

글로벌global/전역 변수를 초기화하기 Initializing Global Variables

글로벌 변수의 경우, 변수를 선언하면서 바로 초기 값을 할당할 수 있다. 상수 [constant](#) 할당 표기법(=)을 사용하면 된다. 할당 연산자(:=)를 사용하는 게 아니다. 아래에서 설명하겠지만, 예를 들어 다음과 같이 작성할 수 있다.

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

이 초기화 기법은 오직 글로벌 변수에서만 쓸 수 있다. 사실 글로벌 변수는 무조건 초기값이 지정된다. 초기 값을 따로 지정하지 않으면 기본값 [default value](#) 으로 지정된다 (예: 숫자의 경우 0 으로 초기화).

로컬Local/지역 변수를 초기화하기 Initializing Local Variables

프로시저나 함수가 시작되는 곳에 선언되는 변수는, 글로벌 변수와 달리, 초기화되지 않는다. 따라서 기본값으로 할당되지 않으며, 선언하면서 할당하는 구문도 없다. 이런 변수는 초기값을 지정하는 코드를 명시적으로 코드 시작 부분에 추가하는 것이 좋다.

```
var
  Value: Integer;
begin
  Value := 0; // 초기화(Initialize)
```


즉, 로컬 [local/지역](#) 변수를 초기화하지 않고 그냥 사용하면 그 변수에는 완전히 임의의 값이 들어 있다 (그 변수가 차지한 메모리 위치에 담긴 바이트 [byte](#) 에 따라 달라진다). 몇몇 상황에서는, 컴파일러가 잠재적 오류에 대해 경고하지만 항상 그러지는 않는다.

다시 말해, 아래 코드를 작성하면,

```
var
  Value: Integer;
begin
  ShowMessage(Value.ToString); // Value가 정의되지 않았음(undefined)
```

컴파일러는 Value 라는 Integer [정수](#) 타입 변수가 초기화되지 않았다는 경고를 표시한다. 그리고, 여러분이 프로그램을 실행하면 그 결과 값은 완전히 임의의 값이다. 실행 당시 Value 변수가 차지하고 있는 메모리 위치에 무슨 바이트 [byte](#) 들이 있든지 그냥 Integer 로 해석되어 표시되는 것이다.

인라인 변수 [Inline Variables](#)

델파이 최근 버전에서(10.3 리오부터) 추가된 개념이 있다. 초기 파스칼과 터보 파스칼 컴파일러 시절부터 사용해 오고 있는 방식과 다른 방식으로 변수 선언을 할 수 있다: 인라인 변수 선언이라는 개념이다.

인라인 변수 선언 구문 [syntax](#) 을 쓰면, 변수를 코드 블록 안에서 직접 선언할 수 있다 (기존 변수 선언과 마찬가지로 심볼 [symbol](#) 여러 개를 나열할 수도 있다).

```
procedure Test;
begin
  var I, J: Integer;
  I := 22;
  J := I + 20;
  ShowMessage(J.ToString);
end;
```

겉으로는 그리 대단한 변화가 아니다. 하지만, 이 변화로 인한 부수적인 효과가 있다. 초기화 [initialization](#), 타입 추론 [type inference](#), 변수 수명 [lifetime](#) 과 관련된 효과다. 곧이어 다루겠다.

인라인 변수를 초기화하기 [Initializing Inline Variables](#)

오래된 선언 모델에 비해 첫 번째로 가장 크게 다른 점은, 한 문장으로 변수에 대한 인라인 선언과 초기화를 모두 할 수 있다는 것이다. 그래서 예전 방식보다 가독성이 더 좋다. 앞에서 본 예전 방식 즉 함수의 시작 부분에 여러 변수들에 대한 초기화를 적어 놓은 코드보다 간결하다.

```
procedure Test;
begin
  var I: Integer := 22;
  ShowMessage(I.ToString);
end;
```


주요 장점을 보자. 여러분이 코드 블록의 뒷부분에 가서야 변수의 값을 얻을 수 있는 경우라면, 예전 방식은 앞에서 초기값(예: 0 또는 nil)을 미리 지정하고, 뒤에 가서 실제 값을 할당한다. 하지만, 인라인 방식은 알맞은 초기 값을 계산할 수 있는 위치에 이를 때까지 변수 선언을 미룰 [delay](#) 수 있다. 아래 예문에서 `k`를 눈여겨보자.

```
procedure Test1;
begin
  var I: Integer := 22;
  var J: Integer := 22 + I;
  var K: Integer := I + J;
  ShowMessage(K.ToString);
end;
```

다르게 설명하자면, 예전 방식은 모든 지역 변수가 해당 코드 블록 안 모든 곳에서 보인다. 지금 인라인 변수는 오직 그것이 선언된 위치에서부터 해당 코드 블록의 끝 사이에서만 보인다. 따라서, 위 코드 블록에서, 첫 두 줄에서는 변수 `k`를 사용하지 못한다. 결과적으로, `k`에게 알맞은 값이 할당된 다음부터 `k`를 사용할 수 있다.

인라인 변수에 대한 타입 추론 [Type Inference](#)

인라인 변수는 또 다른 큰 장점이 있다. 컴파일러는, 몇몇 상황에서, 인라인 변수의 타입을 유추 [infer](#) 할 수 있다. 그 변수에 할당된 표현식이나 값의 타입을 보고 알아낸다. 다음은 매우 간단한 예문이다.

```
procedure Test;
begin
  var I := 22;
  ShowMessage(I.ToString);
end;
```

rvalue 위치에 있는 표현식(즉, `:=` 뒤에 오는 것의)의 타입을 분석해서 그 변수의 타입을 결정한다. 문자열 [string](#) 상수 [constant](#)를 할당하면 변수는 문자열 [string](#) 타입이 된다. 이런 분석 수행은 여러분이 함수 [function](#)나 복합 표현식 [complex expression](#)의 결과를 할당하는 경우에도 마찬가지다.

알아 둘 점이 있다. 이 변수는 여전히 타입에 엄격하다. 데이터 타입 결정을 컴파일러가 하고, 할당 역시 컴파일 시점에 수행되기 때문에 새 값을 할당한다고 해도 그 타입은 변하지 않는다. 타입 추론은 코드를 더 적게 작성할 수 있도록 돕는 편의 기능일 뿐이다(복잡한 데이터 타입에서 유용하다. 예: 복잡한 제네릭 [generic](#) 타입 등). 정적 [static](#)이고 타입이 강력하게 지정하는 [strongly-typed](#)이 언어의 특성을 바꾸는 기능이 전혀 아니다. 당연히, 실행 시간 [runtime](#)에 속도 저하를 유발하지도 않는다.

참고 타입 추론을 할 때, 일부 데이터 타입은 더 큰 타입으로 "확장"된다. 위 예문에서는 숫자 값 22(`ShortInt`)가 `Integer`로 확장된다. 일반적인 규칙은, 오른쪽에 있는 표현식의 타입이 정수 타입이고 32 비트보다 작은 경우, 그 변수는 32 비트 `Integer`로 선언된다. 물론, 여러분이 더 작은 숫자 타입을 원한다면, 그 타입을 직접 명시하면 된다.

상수 Constants

오브젝트 파스칼은 상수 `constant` 선언 또한 허용한다. 상수를 쓰면, 프로그램이 실행되는 내내 변하지 않는 값에 의미를 담은 이름을 지정할 수 있다(또한 프로그램의 크기가 줄어들 수 있다. 같은 상수 값이 컴파일 된 코드 안에 중복되지 않기 때문이다).

상수를 선언할 때는 데이터 타입을 지정할 필요가 없다. 초기 값만 지정하면 된다. 컴파일러는 그 값을 보고 적절한 데이터 타입을 자동으로 추론한다. 예문을 보자 (역시 `VariablesTest` 예제에서 발췌함).

```
const
  Thousand = 1_000;
  Pi = 3.14;
  AuthorName = '마르코 칸투';
```

컴파일러는 상수의 데이터 타입을 결정할 때 그 값을 보고 판단한다. 위 예제에서 상수 `Thousand` 는 `SmallInt` 타입이 될 거라고 가정할 수 있다. 그 값을 담을 수 있는 가장 작은 정수 타입이기 때문이다. 원한다면, 특정 타입을 사용하도록 컴파일러에게 지시할 수 있다. 선언 `declaration` 안에 원하는 타입을 적어 넣으면 된다. 아래와 같다.

```
const
  Thousand: Integer = 1_000;
```

경고 프로그램에서 가정`assumption`이란 대체로 나쁘다. 또한 컴파일러는 시간이 지나면서 변할 수 있다. 그래서 개발자가 했던 가정이 더 이상 맞지 않을 수도 있다. 코드를 보다 명확하게 표현할 수 있는 방법이 있고 가정이 필요하지 않을 수 있다면, 그렇게 하라!

여러분이 상수를 선언하면, 컴파일러는 메모리 위치를 할당하여 그 상수를 저장할 것인지, 아니면, 그 상수의 실제 값을 복제하여 그 값이 사용되는 모든 곳마다 넣을 것인지를 선택할 수 있다. 두 번째 방식은 단순한 상수에 특히 적합하다.

상수가 일단 선언되고 나면, 다른 변수와 거의 비슷하게 사용할 수 있다. 하지만, 새 값을 그 상수에 할당할 수는 없다. 그런 시도를 하면, 컴파일러 오류가 발생한다.

참고 많이 이상하겠지만, 오브젝트 파스칼에서는 타입이 지정된 상수`typed constant`의 값을 실행-시간에 변경할 수 있도록 허용한다. 하지만, 오직 `$J` 컴파일러 지시어를 활성화하거나, 이에 해당하는 *Assignable typed constants* 컴파일러 옵션을 사용하는 경우에만 가능하다. 이런 선택적 동작을 할 수 있게 한 이유는 예전 컴파일러로 작성된 코드와 하위 호환성을 고려했기 때문이다. 이런 코딩 스타일은 권장하지 않는다. 나는 그저 프로그래밍 기법에 대한 역사적 일화 정도로 참고할 수 있도록 적어 놓았을 뿐이다.

인라인 상수 Inline Constant

앞서 변수에서 본 것과 같이, 상수 값 선언도 이제 인라인으로 처리할 수 있다. 이 기능은 타입이 지정된 상수 `typed constant`, 그리고 타입이 지정되지 않은 상수 `untyped constant`

즉, 타입이 추론되는 상수(오랫동안 상수에 사용되던 방식)에 적용할 수 있다. 다음은 간단한 예제다.

```
begin
  // 몇몇 코드
  const M: Integer = (L + H) div 2; // 식별자에 타입이 명시되어 있음
  // 몇몇 코드
  const M = (L + H) div 2; // 식별자에 타입이 명시되어 있지 않음
```

일반 상수 선언에는 상수 값 [constant value](#) 만 할당할 수 있지만, 인라인 상수 선언에서는 어떤 표현식 [expression](#) 이든 사용할 수 있다.

리소스 문자열 상수 [Resource String Constant](#)

이건 약간 고급 주제이지만, 문자열 상수 [string constant](#) 를 정의할 때. 표준 상수 선언 대신 `resourcestring` 이라는 특정 지시어를 사용해 작성 수 있다. 이 지시어는 문자열을 윈도우 [Windows](#) 리소스(또는 오브젝트 파스칼이 지원하는 ‘윈도우 이외의 플랫폼’에서는 그 안으로 들어가는 이와 동등한 데이터 구조)처럼 취급하도록 컴파일러 [compiler](#) 와 링커 [linker](#) 에게 지시한다.

```
const
  AuthorName = '마르코';
resourcestring
  StrAuthorName = '마르코';
begin
  ShowMessage(StrAuthorName);
```

위에서 두 경우 모두 상수, 즉 프로그래밍 실행 중에 변경하지 않는 값을 정의하고 있다. 차이점은 오직 내부 구현뿐이다. `resourcestring` 지시어를 사용하여 정의된 문자열 상수는 프로그램의 리소스인 문자열 테이블에 저장된다. 그 테이블은 현지화 [localization](#) 도구(문자열을 다른 언어로 번역할 때 쓰는 도구)와 같은 리소스 편집기로 편집할 수 있다.

리소스 사용 시 장점은, 메모리를 보다 효율적으로 처리하도록 윈도우(또는 다른 플랫폼용으로 델파이에서 구현된 이에 해당하는 기능)가 작동한다는 점이다. 또한 소스 코드를 수정하지 않고도 프로그램을 현지화 [localization](#) 할 수 있다는 것도 장점이다.

경험 법칙을 적용하자면, `resourcestring` 은 사용자에게 표현되고, 번역이 필요할 수 있는 모든 텍스트에서 사용하는 것이 좋다. 그리고 상수는 프로그램 내부의 다른 모든 기타 모든 문자열 (예: 고정된 설정 파일 이름)에서 사용하도록 하자.

팁 IDE 에디터에는 자동 *리팩토링* 기능이 있다. 이것을 사용해 여러분은 코드 안에 있는 문자열 상수를 `resourcestring` 선언으로 교체할 수 있다. 에디터에서 커서를 해당 문자열 리터럴 안에 놓고 `Ctrl+Shift+L`을 누르면 이 리팩토링 기능이 활성화된다.

변수의 수명과 가시성 Lifetime and Visibility of Variables

변수를 어떻게 정의하는가에 따라, 변수가 사용하는 메모리 위치가 달라진다. 그리고, 사용될 수 있게 살아있는 시간(일반적으로 변수의 수명 *lifetime* 이라고 함)이 달라지고, 사용될 수 있는 코드 영역(*가시성* *visibility*이라는 용어로 표현되는 기능)이 달라진다.

모든 옵션을 완벽하게 설명하기에는 아직 너무 이르다. 하지만, 우리는 가장 관련성이 높은 경우들을 확실하게 정리할 수는 있다.

- **글로벌 *global/전역* 변수:** 변수(또는 기타 식별자)를 유닛의 *interface* 구역에 선언하면, 변수의 범위는 이 변수 선언이 담긴 유닛을 사용하는 다른 모든 유닛까지 확장된다. 이 변수를 위한 메모리는 프로그램 시작과 동시에 할당되고 프로그램이 종료될 때까지 유지된다. 글로벌 변수에는 기본 값을 할당할 수 있다. 기본 값을 더 복잡한 방법으로 계산해 할당하려면 유닛의 *initialization* 구역을 사용하면 된다.
- **글로벌 숨김 *global hidden* 변수:** 변수를 유닛의 *implementation* 구역에 선언하면, 이 유닛 바깥에서는 이 변수를 사용할 수 없다. 하지만, 그 유닛 안에 정의된 모든 코드 블록과 프로시저 안에서는 사용할 수 있다. 다만 위치가 그 변수가 선언된 위치보다 뒤에 있어야 한다. 이 변수는 앞에서 본 글로벌 변수와 동일하게 글로벌 메모리를 사용하고 수명도 동일하다. 유일한 차이는 가시성 *visibility*이다. 초기화 역시 글로벌 변수와 동일하다.
- **로컬 *local/지역* 변수:** 변수를 함수, 프로시저, 메서드를 정의하는 블록 안에 선언하면, 해당 코드 블록 바깥에서는 이 변수를 사용할 수 없다. 그 식별자의 범위는 그 함수 또는 메서드의 내부 전체다. 여기에는 중첩되어 들어있는 루틴 *nested routine*의 내부도 포함된다(단, 그 중첩된 루틴 선언이 이 변수보다 더 위에 있지 않아야 한다. 또한 그 중첩된 루틴 안에서 이름이 같은 식별자를 선언해 바깥에 있는 정의를 숨기는 *hide* 경우도 아니어야 한다). 로컬 변수를 위한 메모리는 스택 *stack*에 할당된다. 할당 시점은 프로그램에서 이 변수를 정의하는 루틴을 실행하는 시점이고, 그 루틴이 종료되는 즉시 스택에 있는 그 메모리는 자동으로 해제된다.
- **로컬 인라인 *local inline* 변수:** 함수, 프로시저, 메서드를 정의하는 블록 안에서 인라인 *inline* 변수를 선언하면, 기존 로컬 변수에 비해 가시성 *visibility*이 제한된다. 즉 변수가 선언된 줄 아래부터 볼 수 있다. 그리고 그 함수나 메서드의 끝까지 이어진다.
- **블록 범위 인라인 *block-scoped inline* 변수:** 코드 블록(즉, 중첩된 *begin-end* 문 또는 *try-finally* 블록) 안에 인라인 변수를 선언하면, 그 변수의 가시성은 그 중첩 블록 안으로 제한된다. 이는 다른 프로그래밍 언어 대부분들과 똑같다. 하지만, 오브젝트 파스칼에서는 인라인 변수 선언 구문 *syntax*과 함께 최근에 도입되었다. 유의할 점이 있다. 블록 안에 선언되는 인라인 변수의 식별자는 그 코드 블록 바깥의 다른 변수에서 이미 선언하여 사용하고 있는 식별자와 같으면 안 된다.

참고 블록 범위 인라인 변수의 경우, 가시성^{visibility}뿐 아니라 변수 수명^{lifetime}도 그 블록으로 제한된다. 매니지드^{managed(관리되는)} 데이터 타입(예: 인터페이스, 문자열, 매니지드^(관리되는) 레코드 등)은 그 하위 블록의 끝에서 폐기^{dispose}된다. 그 프로시저나 메서드가 끝날 때까지 남지 않는 게 아니다. 이는 표현식 결과를 잡아 두기 위해 생성하는 임시^{temporary} 변수의 경우에도 마찬가지다.

유닛의 인터페이스 부분에 선언된 모든 것들은 프로그램 어느 곳에서도 접근할 수 있다. 단 `uses` 절 안에 그 유닛을 포함해야 한다. 폼 ^{form} 클래스의 변수도 같은 방식으로 선언된다. 따라서, 다른 폼의 코드에서 그 폼(및 그 폼의 공용 필드, 메서드, 프로퍼티, 컴포넌트)을 참조할 수 있다. 물론 모든 것을 글로벌 ^{global/전역} 선언하는 것은 좋지 않은 프로그래밍 습관이다. 글로벌 변수를 사용하면 메모리 소비 문제가 명백할 뿐만 아니라, 프로그램 유지, 관리, 업데이트가 더 어려워진다. 요컨대, 가능하면 적은 수의 글로벌 변수를 사용해야 한다. 그리고, 델파이가 생성하는 글로벌 변수(예: 폼 변수)에 접근하고 싶은 유혹을 피해야 한다.

데이터 타입들 ^{Data Types}

파스칼에는 사전 정의된 데이터 타입이 여러 가지다. 크게 세 종류로 나눌 수 있다. 바로 순서^{ordinal} 타입, 실수^{real} 타입, 문자열^{string} 타입이다. 여기서는 순서 타입과 실수 타입을 다룬다. 문자열은 6장에서 따로 집중적으로 살펴본다.

델파이에는 타입이 지정되지 않은^{non-typed} 데이터 타입(*variant* 라고 부름) 그리고 기타 "유연한" 타입들, 즉 TValue(강화된 RTTI 지원에 들어있음) 등이 있다. 이러한 수준 높은 데이터 타입들 중 몇 가지는 5장 후반부에서 설명한다.

순서^{Ordinal} 및 숫자^{Numeric} 타입

순서 타입의 기반 개념은 순서^{order} 즉 순번^{sequence}이다. 두 값을 비교해 무엇이 더 높은 값인지 확인할 수 있을 뿐 아니라 다음 순서 값 또는 이전 순서 값을 요청할 수 있다. 그리고, 그 데이터 타입으로 표현할 수 있는 최소 값과 최대 값을 계산할 수 있다.

사전 정의된 순서 타입에서 가장 중요한 세 가지는 Integer^{정수}, Boolean, Char(문자)다. 그 외, 의미는 같지만 내부 표현이나 범위가 다른 경우를 지원하는 다른 타입도 있다.

다음 표는 순서 데이터 타입들 중 숫자를 표현하는 데 사용되는 것들이다.

크기	더하기 빼기 기호 있음 (Signed)	더하기 빼기 기호 없음 (Unsigned)
8 비트	ShortInt: -128 ~ 127	Byte: 0 ~ 255
16 비트	SmallInt: -32768 ~ 32767 (-32K ~ 32K)	Word: 0 ~ 65,535 (0 ~ 64K)
32 비트	Integer: -2,147,483,648 ~ 2,147,483,647 (-2GB ~ +2GB)	Cardinal: 0 ~ 4,294,967,295 (0 ~ 4 GB)

64 비트	Int64: -9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807	UInt64: 0 ~ 18,446,744,073,709,551,615 (읽을 수 있다면!)
-------	---	--

보다시피, 이 타입들은 서로 표현하는 숫자들이 다르다. 이는 값을 표현하는 데 사용되는 비트 수 그리고 더하기 빼기 기호 유무에 의해 달라진다. 기호가 있는 [Signed](#) 타입은 양수와 음수를 모두 값으로 가질 수 있다. 그 대신, 표현하는 숫자의 범위가 더 작다 (크기가 같고 기호가 없는 값에 비해 절반임). 그 이유는 숫자를 저장하는 비트가 하나 더 적기 때문이다.

Int64 타입은 정수를 최대 18 자리까지 표현한다. 또한, 순서 타입 루틴(예: High, Low), 숫자 루틴(예: Inc, Dec), 문자열 변환 루틴 (예: IntToStr)을 런타임 라이브러리로부터 충분히 지원받는다.

별칭이 붙은 정수 타입들 [Aliased Integer Types](#)

ShortInt 와 SmallInt 의 차이(어느 것이 더 작은지 등)를 기억하기 힘들면, 실제 타입 대신 미리 정의된 별칭 [alias](#) 을 쓰면 된다. 이 별칭은 System 유닛 안에 선언되어 있다.

```
type
  Int8   = ShortInt;
  Int16  = SmallInt;
  Int32  = Integer;
  UInt8  = Byte;
  UInt16 = Word;
  UInt32 = Cardinal;
```

다시 말해서, 위 타입들은 새로 추가된 것들이 아니다. 하지만, 아마 사용하기가 더 쉬울 것이다. 실제 구현인 Int16 은 SmallInt 보다 기억하기에 더 간단하다.

이러한 타입 별칭 [alias](#) 은 타입 이름이 비슷한 C 및 기타 언어에서 넘어온 개발자들이 사용하기에도 더 쉽다.

Integer 타입, 64 비트, NativeInt, LargeInt

오브젝트 파스칼 64-비트 버전에서, Integer 타입이 여전히 32 비트라는 사실을 알고 놀랐는가? CPU 수준에서 숫자 처리를 가장 효율적으로 할 수 있는 타입이기 때문이다.

64 비트인 것은 포인터 [Pointer](#) 타입(포인터는 나중에 자세히 설명) 그리고 기타 관련 참조 타입들이다. 숫자 타입을 그 포인터의 크기 그리고 그 네이티브 CPU 플랫폼에 맞춰야 한다면, NativeInt 와 NativeUInt 라는 별칭 타입을 사용할 수 있다. 이 타입들은 해당 플랫폼의 포인터와 크기가 동일하다(즉, 32-비트 플랫폼에서는 32 비트, 64-비트 플랫폼에서는 64 비트).

살짝 다른 상황에서는 LargeInt 타입이 필요하다. 네이티브 플랫폼 API 함수에 매핑하는 데 자주 되는 것이다. 이 타입은 32-비트 플랫폼들과 윈도우 32-비트에서는

32 비트고, 64-비트 ARM 플랫폼에서는 64 비트다. `LargeInt` 타입은 멀리하는 것이 좋다. 운영체제에 맞추기 위해 특별하게 네이티브 코드가 필요한 경우가 아니라면 말이다.

정수 타입 헬퍼 Integer Type Helper

오브젝트 파스칼 언어에서 정수 `integer` 타입들은 오브젝트들과 따로 분리되어 취급된다. 그런데, 정수 타입인 변수(및 상수 값)들은 "점 표기법 `dot notation`"을 적용하는 연산을 할 수 있다(즉, 오브젝트에서 메서드를 적용하는 데 흔히 사용되는 표기법이 가능하다).

참고 기술적으로, 네이티브 데이터 타입에서 이런 방식으로 할 수 있는 연산(operation)들은 "내재된 `intrinsic` 레코드 헬퍼 `record helper`"를 사용해 정의된다. 클래스 헬퍼와 레코드 헬퍼는 12장에서 다룬다. 짧게 말하자면, 여러분은 핵심 데이터 타입들에 적용되는 연산들을 커스터마이징 할 수 있다. 개발 전문가들은 알겠지만, 타입 연산은 그 타입과 짝지어진 내재된 레코드 헬퍼 안에 정적 클래스 메서드 `static class method`로 정의되어 있다.

몇 가지 예를 다음 코드에서 보자 (`IntegersTest` 예제에서 발췌함)

```
var
  N: Integer;
begin
  N := 10;
  Show(N.ToString);

  // 상수를 표현한다
  Show(33.ToString);

  // 타입(Type) 연산(operation), 그 타입을 저장하는 데 필요한 바이트(byte) 표현
  Show(Integer.Size.ToString);
```

참고 위 코드 조각에 사용된 `Show` 함수는 문자열 출력을 메모 컨트롤에 표시하는 간단한 프로시저다. 그러면, `ShowMessage`를 통해 대화 상자를 표시하고 계속 닫는 것보다 편하다. 또한, 표시되는 결과를 복사하고 붙여넣기도 더 쉽다. 이 책은 예제 대부분에서 이 방식을 사용한다.

위 프로그램의 출력 결과는 다음과 같다.

```
10
33
4
```

이 동작들은 (RTL 안의 다른 어느 것들보다) 매우 중요하므로 아래에 나열했다.

<code>ToString</code>	숫자를 문자열로 변환 (10진수 형식으로 표현)
<code>ToBoolean</code>	불리언 <code>Boolean</code> 타입으로 변환
<code>ToHexString</code>	문자열로 변환 (16진수 형식으로 표현)
<code>ToSingle</code>	부동 소수점 데이터 타입으로 변환 (<code>single</code> 타입으로 변환)
<code>ToDouble</code>	부동 소수점 데이터 타입으로 변환 (<code>double</code> 타입으로 변환)
<code>ToExtended</code>	부동 소수점 데이터 타입으로 변환 (<code>extended</code> 타입으로 변환)

첫번째와 세번째 연산은 숫자를 문자열로 변환한다(10 진수 또는 16 진수로 표현된다). 두번째는 불리언 `Boolean` 으로 변환한다. 마지막 세 개는 부동 소수점 타입으로 변환한다. 부동 소수점 타입은 나중에 설명한다.

이외에도 아래 연산들을 정수 타입(및 다른 숫자 타입 대부분)에 적용할 수 있다.

<code>Size</code>	그 타입의 변수를 저장하는 데 필요한 바이트 수
<code>Parse</code>	문자열을 숫자 값으로 변환 (숫자를 표현하는 문자열이 아니면 예외 <code>exception</code> 발생)
<code>TryParse</code>	문자열을 숫자로 변환하려고 시도한다.

순서 타입의 표준 루틴들 Standard Ordinal Type Routines

정수 타입 헬퍼에 정의된 연산 그리고 위에 나열된 연산 외에도, 모든 순서 `Ordinal` 타입 (숫자 타입도 순서 타입에 해당된다)에서 사용할 수 있다는 표준 및 "클래식" 함수 `function` 여러 가지가 더 있다. 대표적인 예로 `SizeOf`, `High`, `Low` 함수 등 타입 자체에 대한 정보를 요청하는 것들이 있다. (모든 데이터 타입에 적용할 수 있는 시스템 함수인) `SizeOf` 의 결과는 주어진 타입의 값을 표현하는데 필요한 바이트 수를 나타내는 정수다 (위에 본 `Size` 라는 헬퍼 `helper` 함수와 같다).

시스템 루틴들 중에서 순서 타입에서 작동하는 것들은 다음과 같다.

<code>Dec</code>	파라미터로 전달된 변수를 감소시킨다. 1씩 감소시키거나 또는 두 번째 파라미터(선택 사항)에 값을 지정한 경우 그 값만큼 감소시킬 수 있다
<code>Inc</code>	파라미터로 전달된 변수를 증가시킨다. 1씩 증가시키거나 또는 두 번째 파라미터(선택 사항)에 값을 지정한 경우 그 값만큼 증가시킬 수 있다
<code>Odd</code>	파라미터가 홀수면 <code>True</code> 를 반환한다. 짝수인지를 테스트하려면 <code>not</code> 표현식을 사용하여 <code>not Odd</code> 라고 적는다
<code>Pred</code>	파라미터 값보다 순서 상 바로 앞에 있는 값 <code>predecessor</code> 을 반환한다. 그 데이터 타입에 정해져 있는 순서를 따른다
<code>Succ</code>	파라미터 값보다 순서 상 바로 뒤에 있는 값 <code>successor</code> 을 반환한다
<code>Ord</code>	파라미터 값이 그 데이터 타입의 값 집합 안에서 순서 상 몇 번째 위치인지를 숫자로 반환한다(숫자가 아닌 순서 타입에 사용)
<code>Low</code>	그 순서 타입 범위에서 가장 낮은 값을 반환한다
<code>High</code>	그 순서 타입 범위에서 가장 높은 값을 반환한다

참고	C 또는 C++ 프로그래머라면 <code>Inc</code> 프로시저의 두 가지 버전, 즉 파라미터가 1 개 또는 2 개인 것이, 연산자 <code>++</code> 과 <code>+=</code> 에 대응한다는 점에 유의해야 한다 (마찬가지로, <code>Dec</code> 프로시저는 <code>--</code> 및 <code>- =</code> 연산자에 대응한다). 오브젝트 파스칼 컴파일러는 이러한 증가 및 감소 연산을 최적화한다. 이는 C 및 C++ 컴파일러가 하는 방식과 유사하다. 하지만 C/C++와 달리, 델파이에서는 사전 증가 및 사전 감소 버전만 제공하고 사후 증가 및 사후 감소 버전은 제공하지 않는다. 또 다른 차이점으로, 델파이의 <code>Inc</code> 와 <code>Dec</code> 는 값을 반환하지 않고 파라미터 자체를 증가/감소시킨다.
-----------	---

알아 둘 점이 있다. 이 루틴들 중 몇몇은 컴파일러에 의해 자동으로 평가되어 해당 값으로 교체된다. 예를 들어, High(X)를 호출하면(X의 타입이 Integer라고 보자), 컴파일러는 그 표현식 자체를 Integer 데이터 타입에서 가능한 가장 높은 값으로 바꾼다. 순서 타입의 이런 함수들 몇 개를 예제로 보자 (IntegersTest 예제에서 발췌)

```
var
  N: UInt16;
begin
  N := Low(UInt16);
  Inc(N);
  Show(N.ToString);
  Inc(N, 10);
  Show(N.ToString);
  if Odd(N) then
    Show(N.ToString + '은(는) 홀수입니다');
```

결과는 다음과 같다.

```
1
11
11 은(는) 홀수입니다
```

여러분이 데이터 타입을 UInt16 대신 Integer 또는 다른 순서 타입으로 바꾼 다음 출력이 어떻게 달라지는지를 직접 확인해 보는 것도 좋다.

범위를 벗어나는 연산 Out-Of-Range Operations

위에 있는 N과 같은 변수는 유효한 값의 범위가 제한되어 있다. 변수 N에 할당하는 값이 음수이거나 너무 크면 오류가 발생한다. 범위를 벗어나는 연산에서 발생할 수 있는 오류에는 실제로 세 가지 유형이 있다.

첫 번째 오류 유형은 컴파일러 오류다. 이것은 범위를 벗어난 상수 값 [constant value](#) (또는 상수 표현식 [constant expression](#))을 할당할 때 발생한다. 예를 들어 위 코드에 아래 코드를 추가하면,

```
N := 100 + High(N);
```

컴파일러가 오류를 발생시킨다.

```
[dcc32 Error] E1012 Constant expression violates subrange bounds
```

두 번째 유형은 컴파일러가 오류 상황을 미리 예측할 수 없는 상황일 때 발생한다. 즉 프로그램 흐름에 의해 오류 상황이 생기는 경우다. (위와 마찬가지로) 다음 코드를 넣어 본다고 가정해보자.

```
Inc(N, High(N));
Show(N.ToString);
```


컴파일러가 오류를 발동 [trigger](#) 하지 않는다. 위 코드는 함수를 호출하는데 컴파일러는 그 효과를 미리 알 수 없기 때문이다 (오류는 N의 처음 값에 따라 달라진다). 이 경우 두 가지 가능성이 있다. 첫째, 기본 설정에서, 이 코드를 컴파일하고 실행하면 변수 N 에는 완전히 비논리적인 값이 담긴다(위 연산에서는 1 을 뺀 결과가 나온다!). 이런 상황은 최악이다. 그 프로그램이 올바르게 작동하더라도 여러분은 오류를 얻지 못하기 때문이다.

둘째, 여러분이 할 수 있는 것은 (그리고 매우 권장되는 바는) 컴파일러 옵션 중 “Overflow checking [오버플로 검사](#)” (`{Q+}` 또는 `{OVERFLOWCHECKS ON}`)를 켜는 것이다. 그러면 이런 오버플로 [Overflow](#) 연산을 방지하고, 오류 (위에서는 “Integer overflow”)를 발생시킨다.

불리언 [Boolean](#)

논리적 True [참](#)와 False [거짓](#) 값은 불리언 [Boolean](#) 타입을 사용해 표현한다. 불리언은 조건문 안에 있는 조건의 타입으로도 사용된다(다음 장에서 살펴본다). Boolean 타입에 담을 수 있는 값은 오직 True 와 False 둘 중 하나다.

경고 마이크로소프트의 COM 및 OLE 자동화^{automation}와의 호환성을 위해, ByteBool, WordBool, LongBool 데이터 타입은 True 값이 -1이다. False 값은 여전히 0이다. 다시 말하지만, 대체로 이 타입들은 사용하지 않는 것이 좋다. 또한 모든 저-수준^{low-level}에서 불리언 조작하기, 불리언을 숫자로 매핑하기 역시 꼭 필요한 경우가 아니라면 피하는 게 좋다.

C 언어와 C 에서 파생된 언어들과 달리, 오브젝트 파스칼에서 Boolean 은 열거 [enumerated](#) 타입에 속한다. Boolean 변수를 숫자 값으로 직접 변환하는 방법은 없다. 여러분은 타입 캐스트 [type cast](#) 를 남용해 Boolean 을 숫자 값으로 직접 변환 [convert](#) 하려고 시도하지 않는 게 좋다. 하지만, Boolean 타입용 헬퍼에는 ToInteger 와 ToString 이 있다. 열거 타입에 대해서는 이 장의 뒷부분에서 살펴보겠다.

ToString 은 그 불리언 변수의 숫자 값을 문자열에 담아 반환한다는 점에 유의하자. 대안으로 글로벌^{전역} 함수인 BoolToStr 를 사용할 수 있다. 이때, 두번째 파라미터에 Ture 를 명시하면, 출력할 때 불리언 문자열(‘True’, ‘False’)을 사용하게 된다. (아래 “문자 [Char](#) 타입 연산” 부분에서 예문을 통해 설명한다)

문자들 [Characters](#)

문자 [Character](#) 변수는 Char 타입을 사용해 정의한다. 예전과 달리 이 언어에서는 현재 Char 타입을 2-바이트 [double-byte](#) 유니코드 문자 [Unicode character](#) 를 표현하는 데 사용한다 (WideChar 타입 역시 2-바이트 유니코드 문자 표현에 사용한다).

참고 델파이 컴파일러는 1-바이트 ANSI 문자용 AnsiChar와 유니코드 문자용 WideChar를 여전히 구분해서 따로 제공한다. Char 타입은 WideChar의 별칭alias으로 정의되어 있다. 권장 사항은 WideChar에만 집중하기 그리고 1 바이트byte 요소에는 Byte 데이터 타입을 사용하기다. 그러나 Delphi 10.4부터는 AnsiChar 타입을 모든 컴파일러와 플랫폼에서 사용할 수 있게 되었다는 점 역시 사실이다. 이는 기존 코드와 더 잘 호환할 수 있도록 하기 위해 제공되는 것이다.

유니코드의 문자 character에 대한 소개는 6 장에 있다. 코드 포인트의 정의와 써로게이트 쌍 surrogate pairs/대리 쌍의 정의(그리고 기타 수준 높은 주제들)에 대해서도 설명되어 있다. 지금 여기에서는 Char 타입의 핵심 개념에만 집중하겠다.

앞서 리터럴 값을 다룰 때 봤듯이, 상수 문자를 표현할 때는 'k'와 같이 기호 표기 방식 symbolic notation 또는 #78 과 같이 그 문자에 해당하는 해당되는 숫자를 표기하는 방식 numeric notation 을 모두 사용할 수 있다. 숫자 표기 방식에서는 Chr(78)에서와 같이 시스템 함수인 Chr 를 쓸 수도 있다. 반대 방향으로 변환할 때는 Ord 함수를 사용한다. 대체로 문자열, 숫자, 기호를 표기할 때는 기호 표기 방식을 쓰는 것이 더 좋다.

특수 문자 즉, #32 보다 앞에 있는 제어 문자 control character 를 표현할 때는 대체로 숫자 표기 방식을 사용한다. 아래 항목들은 가장 흔하게 사용되는 특수 문자들이다.

#8	백스페이스(backspace)
#9	탭(tab)
#10	새 줄(new line)
#13	캐리지 리턴(carriage return)
#27	이스케이프(escape)

문자 타입 연산 Char Type Operations

다른 순서 ordinal 타입들과 마찬가지로, Char 타입에는 여러 연산들이 미리 정의되어 있다. 그래서 여러분은 점 표기법으로 이 타입의 변수에 그 연산들을 적용할 수 있다. 다시 말하지만, 이 연산들은 "내재된 intrinsic 레코드 헬퍼 record helper"를 통해 정의된다.

그러나, 사용 상황은 꽤 다르다. 첫째, 이 기능을 사용하려면 먼저 활성화해야 한다. uses 문에 Character 유닛을 넣어 참조하면 된다. 둘째, Char 타입용 헬퍼 helper 에는 변환 함수 몇 가지만 있는 것이 아니라 유니코드 전용 연산 수십 가지도 들어 있다. 예를 들면 IsLetter, IsNumber, IsPunctuation 등 테스트 연산, ToUpper, ToLower 등 변환 함수들도 들어 있다. 예제는 다음과 같다 (IntegersTest 예제에서 발췌함).

```
uses
  Character;
...
var
  Ch: Char;
begin
  Ch := 'a';
  Show(BoolToStr(Ch.IsLetter, True));
  Show(Ch.ToUpper);
```


이 코드는 아래 결과를 출력한다.

```
True
A
```

참고 문자 타입 헬퍼^{helper}에 있는 `ToUpper` 연산은 유니코드를 완전히 지원한다. 즉, *û*처럼 악센트가 있는 문자를 전달하면 결과는 *Û*가 된다. 기존 RTL 함수 중 일부는 이처럼 똑똑하지 않으며 일반 ASCII 문자에 대해서만 효과가 있다. 그 기존 함수들은 아직도 바뀌지 않고 그대로다. 그 이유는 해당 유니코드 연산들은 처리 속도가 훨씬 더 느리기 때문이다.

문자는 순서 타입이다 Char as an Ordinal Type

Char 데이터 타입은 상당히 크다. 하지만, 여전히 순서 ^{ordinal} 타입이다. 따라서 `Inc`, `Dec` 함수를 사용할 수 있다 (그 문자로부터 주어진 숫자만큼 앞 또는 뒤에 있는 문자를 얻을 수 있다. 이 함수들은 이미 “순서 타입의 표준 루틴들”에서 소개했다). 또한, Char 카운터를 이용해 `for` 루프 ^{loop}를 작성할 수 있다 (`for` 루프는 다음 장에서 더 설명한다).

아래 간단한 코드 조각은 시작 지점으로부터 주어진 숫자만큼 증가된 위치에 있는 문자들을 구해서 표시한다.

```
var
  Ch: Char;
  Str1: string;
begin
  Ch := 'a';
  Show(Ch);
  Inc(Ch, 100);
  Show(Ch);

  Str1 := '';
  for Ch := #32 to #1024 do
    Str1 := Str1 + Ch;
  Show(Str1);
```

이 `for` 루프는 매우 많은 텍스트를 문자열에 추가한다. 따라서 결과가 상당히 길다 (`CharsTest` 예제에서 발췌함). 결과 텍스트의 시작 부분만 보면 아래와 같다.

```
 a
 A
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcde
fghijklmnopqrstuvwxyz{|}~
// ...몇 줄이 더 생략됨
```

Chr 을 사용하여 변환하기 Converting with Chr

이미 앞에서 `Ord` 함수가 문자에 해당하는 숫자 값(보다 구체적으로는 유니코드의 코드 포인트 ^{codepoint})을 반환한다는 것을 살펴보았다. 이와 정반대 함수 즉 코드 포인트를 가지고 해당 문자를 가져오는 함수가 있다. `Chr` 라는 특수 함수다.

32-비트 문자 32-bit Characters

비록 기본인 Char 타입은 WideChar 타입에 매핑되지만, 델파이에서는 4-바이트 문자 타입인 UCS4Char 도 정의되어 있다는 점을 알아 둘 필요가 있다. 이것은 유니버설 문자 세트의 4-바이트 표현을 위한 용도이며, System 유닛에 정의되어 있다.

```
type
    UCS4Char = type LongWord;
```

위 타입 정의 그리고 이에 상응하는 UCS4String(UCS4Char 의 배열로 정의됨)은 거의 사용되지 않지만 이것들은 언어 런타임에 들어있으며, Character 유닛에 있는 일부 함수에서 사용되고 있다.

부동 소수점 타입 Floating Point Types

정수는 그 종류가 다양하긴 해도, 순서가 있는 ordinal 값들의 집합 set 으로 표현된다. 이와 달리, 부동 소수점 숫자는 순서 ordinal 타입이 아니다(순서 order 라는 개념은 있지만, 요소들이 이어지는 시퀀스 sequence/순번 라는 개념은 없다). 부동 소수점 숫자의 값은 근사치(대략적인 값)로 표현된다. 그래서 약간의 오차가 있다.

부동 소수점 숫자는 형식 format 이 다양하다. 이는 숫자를 표현하기 위해 사용되는 바이트 수 그리고 근사치의 품질에 의해 달라진다. 다음은 오브젝트 파스칼의 부동 소수점 데이터 타입들의 목록이다.

Single	저장소 크기가 가장 작은 부동 소수점 숫자는 Single 숫자다. 4 바이트 값으로 구현된다. 단일 정밀도 single precision 부동 소수점 숫자에서 나온 이름이다. 다른 언어들은 이와 동일한 타입을 float라고 부른다.
Double	8 바이트로 구현된 부동 소수점 숫자다. 배정밀도 double precision 부동 소수점 값에서 나온 이름이다. 많은 언어에서도 같은 이름을 쓰고 있다. double 타입의 정밀도는 가장 일반적으로 쓰이는 부동 소수점 데이터 타입이다. 또한 이것은 파스칼에 예전부터 있던 Real 타입에 대한 별칭 alias이기도 한다.
Extended	10 바이트 값 숫자다. 최초의 델파이 Win32 컴파일러 안에서 구현된 것이다. 하지만 이 타입은 모든 플랫폼에서 사용하지는 못한다 (Win64 등 일부 플랫폼에서는 Double로 되돌아간다. 한편, macOS에서는 16 바이트다). 다른 프로그래밍 언어에서는 이 데이터 타입을 long double이라고 부른다.

위 부동 소수점 데이터 타입들은 모두가 IEEE 의 표준 부동 소수점 표현에 해당하는 타입이다. 그리고 서로 정밀도가 다르다. CPU(정확히 말하면, 부동 소수점 장치 Unit 인 FPU)는 이 타입들을 직접 지원한다. 이는 속도를 극대화하기 위함이다.

숫자 데이터 타입이지만 비-순서 non-ordinal 타입인 것이 두 가지 더 있다. 이 타입들은 근사치이 아니라 정확한 숫자를 표현하는 용도로 사용된다.

Comp	8 바이트 즉 64 비트(십진수를 18 자리까지 표현할 수 있음)를 사용하여 매우 큰 정수를 표현한다. 부동 소수점 타입과 달리 큰 숫자를 표현할 때 값을 잃지 않는다. (참고: DocWiki 에서는 Comp 보다는 Int64를 사용하라고 권장하고 있다).
Currency	소수점 이하 숫자가 4 자리로 고정된 소수점 값이다. 64 비트를 사용하여 표현한다는 점은 Comp 타입과 같다. 이름처럼, Currency 데이터 타입은 화폐 값을 소수점 이하 4 자리까지 정밀하게 다루기 위해 추가되었다 (계산 시 값을 잃지 않는다).

참고 델파이 11에서는 Currency 데이터 타입용 새 레코드 헬퍼가 도입되었다. 그래서 반올림 [rounding](#), 문자열을 숫자로 변환 [parsing](#), 숫자를 문자열로 변환 [string conversion](#)을 할 수 있다. 이는 부동 소수점 타입에 사용할 수 있는 헬퍼(뒤에서 설명한다)와 비슷하다.

이와 같은 모든 비-순서 [non-ordinal](#) 데이터 타입들은 High, Low, Ord 함수라는 개념이 없다. 실수 [real](#) 타입은 (이론상) 무한한 숫자 집합을 나타낸다. 이와 달리, 순서 [ordinal](#) 타입은 고정된 [fixed](#) 값들의 집합을 나타낸다.

왜 부동 소수점 값들은 다를까?

더 자세히 설명하겠다. 정수 타입 23이 있으면, 우리는 그 다음 값이 24라는 걸 안다. 정수는 유한한다(범위가 정해져 있고 순서가 있다). 하지만 부동 [floating](#) 소수점 숫자는 무한하다. 매우 작은 범위 안에서도 그렇다. 또한 시퀀스 [sequence/순번](#) 라는 개념도 없다. 그렇다면 23.0 과 24.0 사이에는 값이 몇 개 있을까? 23.46 의 다음 값은 무엇일까? 23.47, 23.461, 23.4601 중 어느 것일까? 이걸 아는 건 불가능하다!

이런 이유로, Char 데이터 타입인 문자 'w'가 그 범위 안에서 몇 번째 위치에 있는지 묻는 것은 말이 되지만, 부동 소수점 데이터 타입인 7143.1562 가 그 범위 안에서 그 위치가 어디인지 똑같이 질문하는 것은 전혀 말이 되지 않는다. 어느 실수 [real](#) 가 다른 실수에 비해 값이 더 큰지는 알 수 있다. 하지만, 주어진 숫자를 기준으로 그 앞에 있는 실수가 모두 몇 개인지 물어보는 것(즉, Ord 함수 사용)은 말이 안 된다.

또 다른 핵심 개념이 부동 소수점 값의 이면에 있다. 구현 상 모든 숫자를 정확하게 표현하지 못한다. 종종 여러분이 예상한 계산 결과 숫자(때로는 정수에서도)가 실제로는 그 숫자의 근사치인 경우가 있다. 아래와 같다 (FloatTest 예제에서 발췌).

```
var
  S1: Single;
begin
  S1 := 0.5 * 0.2;
  Show(S1.ToString);
```

여러분은 결과로 0.1 을 예상하겠지만, 실제 결과는 0.100000001490116 이다. 예상했던 값에 가깝긴 하지만 정확하지는 않다. 물론 이 결과를 반올림한다면 예상한 값을 얻을 수 있다. 만약 Single 대신 Double 변수를 사용한다면 결과 출력은 0.1 이 된다(이 비교 역시 FloatTest 예제 안에 있다).

참고 컴퓨터에서 부동 소수점 수학 연산을 어떻게 하는지에 대해 심도 있게 논의할 시간이 없으므로 이 논의는 여기에서 줄이겠다. 하지만, 오브젝트 파스칼 언어 관점에서 이 주제에 관심 있다면 고인이 된 루디 벨투이스(Rudy Velthuis)의 훌륭한 글(<http://rvelthuis.de/articles/articles-floats.html>)을 추천한다.

부동 소수점 헬퍼와 Math 유닛 Floating Helpers and the Math Unit

위 코드에서 봤듯이, 부동 소수점 데이터 타입인 변수는 연산을 직접 적용할 수 있다 (오브젝트가 연산하는 방식이다). 이 타입들을 위한 레코드 헬퍼 [record helper](#) 들이 있기 때문이다. 실제로, 부동 소수점 숫자에서 직접 사용할 수 있는 연산들은 상당히 많다.

다음은 single 타입의 인스턴스 [instance](#) 에서 직접 쓸 수 있는 연산들이다 (이름만 봐도 동작을 알 수 있는 연산도 있지만, 이해하기 어려운 이름도 있다. 필요하면 도움말을 찾아볼 수 있다).

Exponent	Fraction	Mantissa
Sign	Exp	Frac
SpecialType	BuildUp	ToString
IsNan	IsInfinity	IsNegativeInfinity
IsPositiveInfinity	Bytes	Words

런타임 라이브러리에는 Math 유닛이 있다. 여기에는 고급 수학 루틴들이 정의되어 있다. 삼각 함수(예: ArcCosh 함수), 재무(예: InterestPayment 함수), 통계(예: MeanAndStdDev 프로시저) 관련 여러 루틴들이 있다. 그 중에는 이름이 매우 이상한 것도 있다. 예를 들면 MomentSkewKurtosis 함수가 그렇다(웁긴이: 도움말에서 찾아볼 수 있음).

이 System.Math 유닛 안에는 함수들이 매우 풍부하게 들어 있다. 그런데 여러분은 오브젝트 파스칼을 위한 외부 수학 함수들도 많이 찾을 수 있을 것이다.

간단한 사용자 정의 User-Defined 데이터 타입

타입이라는 개념과 함께 니클라우스 비르트 [Niklaus Wirth](#) 가 최초 파스칼 언어에서 도입한 훌륭한 아이디어 중 하나는 새 데이터 타입을 프로그램 안에서 정의할 수 있게 하는 기능이다 (지금은 당연히 여겨지지만 당시에는 그렇지 않았다). *타입* 정의를 통해서 자신만의 데이터타입을 정의하면 되기 때문에 우리는 하위범위 [subrange](#) 타입, 배열 [array](#) 타입, 레코드 [record](#) 타입, 열거 [enumerated](#) 타입, 포인터 [pointer](#) 타입, 세트 [set](#) 타입 등을 직접 정의할 수 있다. 가장 중요한 사용자 정의 데이터 타입은 클래스다. 클래스는 이 언어의 오브젝트 지향 능력의 한 축이다. 객체 지향은 2 부에서 다룰 주제다.

타입 생성자 [type constructor](#) 는 많은 프로그래밍 언어들이 공통적으로 가진 개념이라고 생각할 텐데 사실이다. 그리고 최초로 이런 개념을 공식적으로 그리고 매우 정밀한 방식으로 도입한 언어가 바로 파스칼이다. 여전히 오브젝트 파스칼에는 다소 독특한

특징들이 남아 있다. 예를 들면 하위범위 `subrange`, 열거 `enumeration`, 세트 `set` 타입을 정의할 수 있다. 이것들은 다음 장에서 살펴본다. 또한 배열과 레코드와 같은 보다 복잡한 데이터 타입 생성자는 5장에서 다룬다.

명명된 `Named` 타입과 명명되지 않은 `Unnamed` 타입

사용자 정의 `user-defined` 데이터 타입에 이름을 지정할 수 있어서 나중에 사용할 수 있다. 또는 이름을 지정하지 않고 변수에 바로 적용하는 방법도 있다. 오브젝트 파스칼의 관례 `convention` 상 (클래스는 물론이고) 모든 데이터 타입은 그 이름을 지정할 때 접두사 `T`를 사용한다. 이 관례는 꼭 지키는 것이 좋다. Java 나 C#을 사용하던 사람이라면 처음에는 자연스럽게 안겠지만 그렇게 하는 것이 좋다.

타입에 이름을 지정하려면, 프로그램의 “type” 구역에서 지정해야 한다 (유닛 안에는 얼마든지 많은 타입을 추가할 수 있음). 다음은 타입 선언의 간단한 예문이다.

type

```
// 하위범위(Subrange) 정의
TUppercase = 'A'..'Z';

// 열거 타입(Enumerated type) 정의
TMyColor = (Red, Yellow, Green, Cyan, Blue, Violet);

// 세트(Set) 정의
TColorPalette = set of TMyColor
```

위 타입을 사용해 변수들을 정의할 수 있다. 다음과 같다.

var

```
UpSet: TUpperLetters;
Color1: TMyColor;
```

위 코드는 명명된 `Named` 타입을 사용하고 있다. 하지만, 다른 방식도 있다. 타입 이름을 명시적으로 지정하지 않고, 타입 정의를 직접 사용하여 변수를 정의하는 방식이다. 예를 들면, 다음 코드와 같다.

var

```
Palette: set of TMyColor;
```

일반적으로, 여러분은 위 코드와 같은 *이름 없는* 타입 사용을 피하는 것이 좋다. 그 이유가 있다. 루틴 `routine` 에 파라미터로 전달할 수 없고, 다른 변수를 동일한 타입으로 선언할 수도 없기 때문이다. 오브젝트 파스칼 언어는 궁극적으로 타입 이름 등가성 `type name equivalence` 을 따른다. 구조적 타입 등가성 `structural type equivalence` 을 따르지 않는다. 따라서 각 타입 별로 정의가 오직 하나만 있도록 하는 것이 정말로 중요하다. 또한, 유닛의 `interface` 구역에 정의된 타입들은 그 유닛을 `uses` 문에 넣어 참조하는 다른 유닛이 있는 경우, 그 외부 유닛에서도 볼 수 있다는 점을 기억하자.

위 예문에 있는 타입 정의의 의미를 알고 있는가? 전통적인 파스칼의 타입 구조에 익숙하지 않은 개발자를 위해 이제부터 몇 가지 설명을 하겠다. 다른 프로그래밍 언어에 있는 구조와 차이점도 함께 설명할 것이므로, 읽다 보면 흥미로울 것이다.

타입의 별칭 [Type Alias](#)

앞에서 설명했듯이, 델파이 언어는 타입이 호환되는 지를 확인할 때 (타입의 실제 정의가 아니라) 타입 이름을 사용한다. 정의가 똑같은 타입이라도 그 이름이 다르면, 그 두 개는 서로 다른 타입이다.

이는 여러분이 타입의 별칭 [alias](#), 즉 기존 타입을 기반으로 한 새 타입 이름,을 정의할 때도 일부 사실이다. 헷갈리는 이유는 동일한 구문 [syntax](#) 의 두 가지 변형이 있고, 그 효과가 서로 살짝 다르기 때문이다. 아래와 같다 ([TypeAlias](#) 예제에서 발췌함)

```
type
  TNewInt = Integer;
  TNewInt2 = type Integer;
```

위 새 타입들은 둘 다 (자동 변환을 통해) 정수 타입을 할당할 수 있다는 점에서 호환성이 있다. 하지만, TNewInt2 타입은 정확한 일치가 아니다. 위 예에서, TNewInt2 의 실제 타입(Integer)을 파라미터로 기다리는 함수에게 TNewInt2 를 참조 파라미터로 전달할 수 없다.

```
procedure Test(var N: Integer);
begin
end;

procedure TForm40.Button1Click(Sender: TObject);
var
  I: Integer;
  NI: TNewInt;
  NI2: TNewInt2;
begin
  I := 10;
  NI := I; // 작동한다
  NI2 := I; // 작동한다

  Test(I);
  Test(NI);
  Test(NI2); // 에러 발생
```

마지막 줄에서 에러가 발생한다. 에러 메시지는 다음과 같다.

E2033 Types of actual and formal var parameters must be identical

타입 헬퍼 [helper](#) 에서도 비슷한 일이 발생한다. TNewInt 는 정수 타입 헬퍼를 사용할 수 있다. 하지만 TNewInt2 는 그렇지 않다. 이 부분은 나중에 레코드 헬퍼 [record helper](#) 에 대해 설명할 때 구체적으로 다룬다.

하위범위 타입 Subrange Types

하위범위 [subrange](#) 타입은 다른 타입의 값 범위 내에서 일부 범위를 정의한다 (그래서 하위범위라는 이름이 붙었다). 예를 들어 Integer 타입의 하위범위를 정의할 수 있다. 예를 들면, 1부터 10까지 또는 100부터 1000까지를 별도의 타입으로 정의할 수 있다. 또는 Char 타입의 하위범위를 정의하여 영어 대문자만 담을 수 있는 문자 타입을 정의할 수 있다.

```
type
  TTen = 1..10;
  TOverHundred = 100..1000;
  TUpperCase = 'A'..'Z';
```

하위범위를 정의할 때는 해당 타입의 상수 두 개만 제공하면 된다 (그 기반 타입의 이름을 지정할 필요가 없음). 기반 타입은 순서 [ordinal](#) 타입이어야 하며 결과 타입 역시 또 다른 순서 타입이 된다. 하위범위 타입으로 변수를 정의하면, 그 변수에는 해당 하위범위 안에 있는 어떤 값이든 할당할 수 있다. 따라서 아래 코드는 유효하다.

```
var
  UpLetter: TUpperCase;
begin
  UpLetter := 'F';
```

하지만, 아래 코드는 유효하지 않다.

```
var
  UpLetter: TUpperCase;
begin
  UpLetter := 'e'; // 컴파일 타임 에러
```

위의 코드는 “상수 표현식이 하위범위 타입의 범위를 위반한다.”라는 컴파일 타임 에러가 발생한다. 대신, 아래와 같이 코드를 작성하면 컴파일러가 허용한다.

```
var
  UpLetter: TUpperCase;
  Letter: Char;
begin
  Letter := 'e';
  UpLetter := Letter;
```

실행 시 [runtime](#) 에는 예상한대로 [범위 확인 에러](#) [Range check error](#) 메시지가 표시된다. 단, Project Options 대화 상자의 Compiler 페이지에서 Range Checking 컴파일러 옵션을 활성화한 경우에만 이 에러 메시지를 받을 수 있다. 이는 앞에서 설명했던 정수 타입 오버플로 에러와 비슷하다.

프로그램을 개발하는 동안에는 이 컴파일러 옵션을 켜는 것이 좋다. 프로그램이 더 견고해지고 디버깅하기도 더 쉬울 것이다. 에러 상황에서, 확인할 수 없는 동작이 아니라 명확한 메시지를 받기 때문이다. 결국 프로그램을 최종 빌드할 때에는 이

옵션을 비활성화할 수도 있을 것이다. 그러면 실행 속도가 약간 더 빨라진다. 하지만, 이로 인한 속도 증가는 거의 무시할 수 있는 수준이므로 프로그램을 시장에 내놓을 때에도 이러한 런타임 검사를 모두 켜 둘 것을 권한다.

열거되는 타입 Enumerated Types

열거되는 Enumerated 타입(주로 “Enum” 이라고 함)은 또 다른 사용자-정의 순서 타입이다. 기존 타입 안에서 범위를 지정하는 대신 열거 타입에는 그 안에 들어갈 수 있는 모든 값들을 사용자가 직접 나열한다. 다시 말하면, 열거 타입은 (상수 constant) 값들의 목록이다.

type

```
TColors = (Red, Yellow, Green, Cyan, Blue, Violet);
TSuit = (Club, Diamond, Heart, Spade);
```

목록의 각 값에는 해당 순서 ordinality 가 있으며, 그 순서는 0 부터 시작한다. Ord 함수를 열거 타입의 값에 적용하면 이 “0 기반” 값을 얻게 된다. 예를 들어 Ord(Diamond)는 1 을 반환한다.

열거 타입은 그 내부 표현이 다를 수 있다. 기본적으로 By default 델파이의 8 비트 표현을 사용한다. 이는 열거되는 서로 다른 값이 256 가지를 넘지 않을 경우다. 서로 다른 값이 이 보다 많은 경우에는 16 비트 표현이 사용된다. 32 비트 표현도 있는데, 이는 때때로 C 또는 C++ 라이브러리와 호환성이 필요할 때 유용하다.

참고 여러분은 열거 타입의 기본 표현을 직접 바꿀 수 있다. 열거 타입의 요소 개수에 관계없이 더 큰 타입을 요청하려면 \$Z 컴파일러 지시어를 사용하면 된다. 하지만, 이 설정은 거의 사용되지 않는다.

범위 지정 열거자들 Scoped Enumerators

열거 타입으로 지정된 상수 값들은 글로벌 global/전역 상수로 간주된다. 따라서 모든 곳에서 효과가 있다. 이로 인해 서로 다른 열거 값들 사이에 이름이 충돌할 수 있다. 그래서 이 언어는 열거 값의 범위를 명시할 수 있게 지원한다. 이 기능을 활성화하려면 컴파일러 지시어인 \$SCOPEENUMS 를 사용한다. 그러면 열거 값을 참조할 때 해당 타입 이름을 반드시 앞에 붙여서 완전한 이름을 사용하도록 강제한다.

```
// 전형적인 열거 값(enumerated value)
S1 := Club;

// 범위 지정(Scoped) 열거 값(enumerated value)
S1 := TSuit.Club;
```

이 기능이 도입되었을 당시, 기본 default 코딩 스타일은 전통적 동작을 유지했다. 기존 코드들이 깨지지 않게 하기 위해서였다. 사실, \$SCOPEENUMS, 즉 범위 지정 열거자는,

열거자의 이런 기존 동작을 바꾼다. 즉 타입 이름까지 포함해 전체 이름이 있어야만 **fully qualified** 해당 열거자를 참조할 수 있도록 강제한다.

절대 *absolute* 이름을 가지고 열거 값을 참조하면 충돌 위험이 없다. 또한 열거 값에 (해당 열거 타입의 이니셜 *initial* 로) 접두사를 만들어 붙여서 다른 열거 타입들의 값과 구분할 필요가 없다. 코드 읽기도 더 좋다. 비록 써야 할 코드가 훨씬 길지만 말이다.

예를 들어, System.IOUtils 유닛에 정의된 열거 타입을 하나 보자.

```
{ $SCOPEENUMS ON }
type
  TSearchOption = (soTopDirectoryOnly, soAllDirectories);
```

위와 같이 정의되면, 열거 타입의 두 번째 값을 참조할 때 soAllDirectories 만 적으면 참조할 수 없다. 아래와 같이 그 열거 값의 완전한 이름을 참조해야 한다.

```
TSearchOption.soAllDirectories
```

파이어몽키 *FireMonkey* 플랫폼 라이브러리는 범위 지정 열거자를 상당히 많이 사용한다. 따라서, 실제 값에 접두사로 타입을 적어 주어야 한다. 하지만, 이보다 오래된 VCL 라이브러리는 전통적 모델을 기반으로 한다. RTL 에는 이 두 가지 모델이 섞여 있다.

참고 오브젝트 파스칼 라이브러리에 있는 열거 값에는 종종 해당 타입의 이니셜 두 개 또는 세 개를 시작 부분에 붙이는데, 관례에 따라 소문자를 사용한다. 위 예제에 있는 "so"는 검색 옵션 (Search Options)에서 첫 글자들을 따서 소문자로 앞에 붙였다. 타입 이름을 접두사로 붙이는 방식에서는 이 관례가 다소 불필요한 중복처럼 보일 수 있다. 하지만, 널리 사용되고 있다는 점을 볼 때, 금방 사라질 것 같지는 않다.

세트 타입 Set Types

세트 *set* 타입은 값들을 담은 모음을 표현한다. 세트 타입은 순서 *ordinal* 타입을 바탕으로 하며, 바탕이 되는 순서 타입에 있는 값들을 담은 모음을 세트 타입에 넣을 수 있다. 밀바탕이 되는 순서 타입은 대체로 항목이 많지 않고, 대부분 열거자 *enumeration* 또는 하위범위 *subrange* 인 경우다.

만약 1 부터 3 까지를 담는 하위 타입 *subrange* 이 바탕이라면, 파스칼 표기로 1..3 이라면, 이 세트 타입의 값 하나에는 이 숫자를 0 개부터 3 개까지 넣을 수 있다. 즉 1 (1 개), 2 (1 개), 3 (1 개), 1 과 2 (2 개), 1 과 3 (2 개), 2 와 3 (2 개), 1 과 2 와 3 (3 개), 없음(0 개) 이 값으로 들어갈 수 있다.

변수 하나에는 그 타입이 허용하는 범위 안에 있는 값 하나를 담는 것이 일반적이다. 이와 달리, 세트 타입 변수에는 값을 담지 않거나, 1 개, 2 개, 3 개, 또는 개수에 상관없이 담을 수 있다. 심지어 해당 범위 내에 있는 모든 값들을 담을 수도 있다.

다음은 세트 `set` 의 예다:

```
type
  TSuit = (Club, Diamond, Heart, Spade);
  TSuits = set of TSuit;
```

위 정의를 근거로 이 타입의 변수를 하나 정의하고 그 안에 원래 타입의 값들 중 일부를 넣어보자. 세트 값을 작성하려면 대괄호 안에 항목들을 쉼표로 구분하여 나열한다. 다음 코드는 변수에 여러 값, 단일 값, 빈 값을 할당하는 방법을 보여준다:

```
var
  Cards1, Cards2, Cards3: TSuits;

begin
  Cards1 := [Club, Diamond, Heart];
  Cards2 := [Diamond];
  Cards3 := [];
```

오브젝트 파스칼에서, 세트가 주로 사용되는 경우는 서로 배타적이지 않은 여러 가지 플래그 `flag/표식`들을 표현할 때다. 예를 들어, 글꼴 스타일은 세트 타입의 값이다. 거기에는 굵게, 기울임, 밑줄, 취소선 등이 들어갈 수 있을 것이다. 물론 같은 글꼴이 굵게와 기울임 둘 다 가지거나, 아무 스타일도 없거나, 모든 스타일을 가질 수 있을 것이다. 그렇기 때문에, 글꼴 스타일은 세트 `set` 로 선언되어 있다.

프로그램 코드에서 이 세트 타입에 값을 할당하는 방법은 아래와 같다.

```
Font.Style := []; // 스타일 없음
Font.Style := [fsBold]; // 굵은 스타일만
Font.Style := [fsBold, fsItalic]; // 두 가지 스타일을 함께 활성화
```

세트 연산자 Set Operators

세트는 파스칼 언어에 있는 특유한 사용자 정의 데이터 타입이라는 것을 살펴보았다. 이제는, 세트 연산자들을 살펴보자. 합치기(+), 차집합(-), 교집합(*), 소속되어 있는지 테스트(in), 기타 관계 연산자들이 있다.

세트에 요소를 추가하려면, 추가하려는 요소들만 다른 세트에 담는다. 그리고 나서 그것을 현재 세트와 결합하면 된다. 글꼴 스타일을 예로 들면 다음과 같다.

```
// '굵게'를 추가한다
Style := Style + [fsBold];

// '굵게' 및 '기울임꼴'을 추가하되, (혹시 밑줄이 있다면) 밑줄을 제거한다
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

다른 방법으로, 표준 프로시저인 `Include` 와 `Exclude` 를 쓸 수 있다. 이 방식이 훨씬 더 효율적이다(하지만, 세트 타입이라도 컴포넌트의 프로퍼티에서는 사용하지 못한다)

```
Include(Style, fsBold);
Exclude(Style, fsItalic);
```


표현식과 연산자 Expressions and Operators

변수에 할당할 수 있는 값으로는 타입이 호환되는 리터럴 `literal` 값, 상수 `constant` 값 또는 다른 변수에 있는 값 등이 있다는 것을 이미 학습했다. 대부분의 경우, 우리는 변수에 표현식 `Expression` 의 결과를 할당한다. 표현식은 대체로 변수와 연산자 하나 이상으로 구성된다. 표현식은 이 언어의 핵심 요소 중 하나다.

연산자 사용하기 Using Operators

표현식 작성에는 일반적인 규칙이 없다. 표현식은 주로 무슨 연산자를 사용하는지에 달려있기 때문이다. 그리고, 오브젝트 파스칼에는 연산자들이 꽤 많다. 연산자에는 논리, 산술, 불리언, 관계, 세트 연산자, 그리고 그 외에도 특수 연산자들이 있다.

```
// 샘플 표현식
20 * 5 // 곱셈
30 + n // 덧셈
a < b // 비교 대상보다 작음
-4 // 음수 값
c = 10 // 동일성 테스트(C 구문의 ==와 같다)
```

표현식 `expression` 은 많은 프로그래밍 언어에서 많이 쓰인다. 연산자들 대부분도 그렇다. 표현식이란 상수, 변수, 리터럴 값, 연산자, 함수 결과 등이 유효하게 조합된 문장이다. 표현식을 사용해, 변수에 할당할 값을 결정하고, 함수 또는 프로시저의 파라미터를 계산하고, 조건을 테스트한다. (식별자 자체를 사용하는 게 아니라) 식별자의 값을 연산에서 사용한다면, 그때마다 우리는 표현식을 사용하고 있는 것이다.

참고 표현식의 결과는 일반적으로 임시 변수에 저장된다. 그 임시 변수는 사용자를 대신하여 컴파일러가 자동으로 알맞은 데이터 타입으로 생성한다. 동일한 코드 조각 안에서 동일한 표현식을 두 번 이상 계산해야 하는 경우라면 변수를 명시적으로 선언하여 결과를 넣어 두는 것이 좋다. 복잡한 표현식의 경우 중간 결과를 저장하는 임시 변수가 여러 개 필요할 수 있지만, 이 역시 컴파일러가 알아서 처리하므로 일반적으로 무시해도 된다.

표현식 결과를 표시하기

표현식으로 몇 가지 실험을 해보고 싶다면 간단한 프로그램을 작성하는 것보다 좋은 방법은 없다. 이 책의 앞에서 본 데모에서 그랬듯이, 폼 `form` 을 가진 간단한 프로그램을 만들고 `Show` 라는 사용자 정의 함수를 만들어서 원하는 내용을 화면에 표시해보자. 표시하려는 정보가 문자열 메시지가 아니라, 숫자 또는 불리언 논리 값인 경우에는 문자열로 변환해야 한다. 예를 들면, `IntToStr`, `BoolToStr` 함수를 호출하면 된다.

참고 오브젝트 파스칼에서, 파라미터는 괄호에 묶여서 함수나 프로시저에 전달된다. 일부 다른 언어(특히 `Rebol`, 더 넓게는 `Ruby`까지)에서는 파라미터를 함수 또는 프로시저 이름 뒤에 적기만 해도 전달할 수 있다. 오브젝트 파스칼로 돌아가서, 함수를 중첩해서 호출할 때는 아래 코드와 같이 괄호를 중첩해서 사용한다.

예문을 보자 (ExpressionsTest 예제에서 발췌함). 아래 코드는 클래식한 IntToStr 구문 [syntax](#) 을 사용하고 있다. 이 파라미터는 표현식이다.

```
Show(IntToStr(20 * 5));
Show(IntToStr(30 + 222));
Show(BoolToStr(3 < 30, True));
Show(BoolToStr(12 = 10, True));
```

이 코드 조각이 출력하는 결과는 뻔하다.

```
100
252
True
False
```

제공되는 데모에는 코드 골격만 있으니, 여러분이 표현식과 연산자를 직접 여러 가지 형식으로 사용해보고 그 결과를 보기 바란다.

참고 오브젝트 파스칼에서는 표현식이 작성되면, 컴파일러가 구문 분석 [parse](#) 을 한다. 그리고 어셈블리 코드를 생성한다. 작성된 표현식을 변경하고 싶으면, 소스 코드를 변경한 다음 애플리케이션을 다시 컴파일해야 한다. 하지만, 시스템 라이브러리들은 동적 표현식 [dynamic expression](#) 을 지원하므로 실행 시 [runtime](#) 에 계산될 수 있다. 이 기능은 리플렉션 [reflection](#) 과 관련이 있으므로 16장에서 다룬다.

연산자 및 우선 순위 [Operators and Precedence](#)

표현식은 값에 연산자를 적용해 만든다. 앞서 언급했듯이, 연산자 대부분은 다양한 프로그래밍 언어에서 공통적으로 사용되고 매우 직관적이다. 기본적인 일치 연산자와 비교 연산자 등이 그렇다. 지금은 오브젝트 파스칼의 고유한 연산자 요소만 강조한다.

아래의 연산자 목록은 우선순위 별로 모여 있다. 그리고, C#, Java, Objective-C (와 C 언어 구문을 기반으로 하는 대부분의 언어)의 연산자에 대한 비교 설명이 함께 있다.

관계 연산자, 비교 연산자 (우선순위 최하위)

=	동일한지 테스트 (C에서는 ==)
<>	같지 않은지 테스트 (C에서는 !=)
<	보다 작은지 테스트
>	보다 큰지 테스트
<=	보다 작거나 같은지 테스트
>=	보다 크거나 같은지 테스트
in	항목이 세트의 구성원인지 테스트
is	오브젝트가 주어진 타입과 호환되는지 테스트(8장에서 다룸), 또는 주어진 인터페이스를 구현하고 있는지 테스트(11장에서 다룸)

더하기/빼기 연산자

+	산술 덧셈, 세트 합집합union, 문자열 합치기concatenation, 포인터 오프셋offset 더하기
-	산술 빼기, 세트 차집합difference, 포인터 오프셋 빼기
or	불리언 또는 비트의bitwise OR (C에서는 또는)
xor	불리언 또는 비트의 배타적 OR (C에서 비트의 배타적 OR 심볼은 ^ 이다)

곱셈 및 비트 bitwise 연산자

*	산술 곱셈, 또는 세트의 교집합
/	부동 소수점 나눗셈
div	정수 나눗셈 (C에서는 /도 사용)
mod	모듈로Modulo/나머지 (정수 나눗셈의 나머지) (C에서는 %)
as	실행 시runtime에 타입을 확인하고 변환conversion을 허용한다 (8장에서 다룸)
and	불리언 또는 비트의bitwise AND (C에서는 && 또는 &)
shl	비트 단위로 왼쪽으로 이동Bitwise shift (C에서는 <<)
shr	비트 단위로 오른쪽으로 이동 (C에서는 >>)

단항 Unary 연산자 (최우선 순위)

@	(변수 또는 함수의) 메모리 주소 (포인터를 반환한다, C에서는 &)
not	불리언 또는 비트의bitwise NOT (C에서는 !)

다른 많은 프로그래밍 언어와 달리, (and, or, not 등) 논리 연산자가 (더 작음, 더 큼 등) 비교 연산자보다 우선 순위가 높다. 이해를 돕기 위해 아래 코드를 보자.

```
| a < b and c < d
```

컴파일러는 AND 연산을 먼저 한다. 그러므로, 보통 위 표현식은 타입 호환성 컴파일 에러가 발생한다. AND 를 중심으로 왼쪽 전체와 오른쪽 전체를 비교하는 테스트를 하려면 괄호로 묶어주어야 한다.

```
| (a < b) and (c < d)
```

이와 달리, 수학 연산에서는 일반 규칙이 적용된다. 그러므로 곱셈과 나눗셈이 덧셈과 뺄셈보다 우선한다. 아래 세 표현식 중 위쪽 두 개는 동일하지만 세 번째는 다르다.

```
| 10 + 2 * 5 // 결과는 20이다
| 10 + (2 * 5) // 결과는 20이다
| (10 + 2) * 5 // 결과는 60 이다
```


팁 괄호가 필요 없을 수도 있다. 언어의 연산자 선행 규칙을 신뢰한다면 더 그럴 것이다. 하지만, 어떤 경우이든 괄호를 추가하는 습관이 더 좋다. 연산자 선행 규칙은 프로그래밍 언어에 따라 다르며, 나중에 코드를 읽거나 수정할 사람을 위해서도 더 명확하게 표현하는 편이 항상 좋다.

연산자 중에는 함께 사용되는 데이터 타입에 따라 다른 의미를 갖는 것이 있다. 예를 들어, + 연산자는 숫자 두 개인 경우 더하기 `add`, 문자열 두 개인 경우 합치기 `concatenate`, 세트 두 개인 경우 합집합 `union` 을 의미한다. 심지어 포인터에 오프셋 `offset` 을 더하는 용도로도 사용된다 (단, 그 포인터 타입에서 포인터 연산 `pointer math` 이 활성화된 경우).

```
10 + 2 + 11
10.3 + 3.4
'Hello' + ' ' + 'world'
```

그러나 (C에서는 하는 것처럼) 문자 `character` 두 개를 합치지는 못한다.

특이한 연산자는 `div` 이다. 오브젝트 파스칼은 숫자(실수 또는 정수) 나눗셈을 할 때 / 연산자를 사용할 수 있다. 그리고 그 결과는 언제나 실수 `real` 이다. 정수 `integer` 끼리 나누고 결과도 정수로 얻으려면 `div` 연산자를 사용해야 한다. 아래 예제에서는 이 두 경우를 대입을 통해 보여준다 (이 코드는 다음 장에서 데이터 타입을 다루면서 더 명확해질 것이다).

```
RealValue := 123 / 12;
IntegerValue := 123 div 12;
```

정수 나눗셈의 결과에서 나머지 `remainder` 가 없는지를 확인하려면 `mod` 연산자를 사용해 그 결과가 0 인지를 확인하면 된다. 아래의 불리언 표현식처럼 확인한다.

```
(x mod 12) = 0
```

날짜Date와 시간Time

파스칼 언어는 초창기 버전에 날짜와 시간을 위한 내장 `native` 타입이 없었다. 하지만, 오브젝트 파스칼은 날짜와 시간에 대한 타입이 내장되어 있으며, 부동 소수점 표현을 사용하여 날짜와 시간 정보를 처리한다. 좀 더 구체적으로 설명하자면, System 유닛에 `TDateTime` 데이터 타입이 정의되어 있어서 이런 목적에 사용할 수 있다.

`TDateTime` 타입은 실제로 부동 소수점 타입이다. 충분히 크기 덕분에 변수 하나 안에 연, 월, 일, 시간, 분, 초, 밀리초 단위까지 모두 저장할 수 있기 때문이다.

- 날짜는 1899-12-30 이후 경과된 날짜의 수다 (1899년 이전 날짜는 음수 값임). 그 숫자는 `TDateTime` 값의 정수 부분에 저장된다
- 시간은 하루의 분수 `fraction` 값이다. 그 숫자는 소수점 이하 부분에서 저장된다

역사 기준이 되는 이 이상한 날짜가 어디에서 왔는지 궁금할 수 있으므로, 엑셀^{Excel}, 그리고 윈도우^{Windows} 애플리케이션의 날짜 표현과 관련된 다소 긴 이야기를 해보겠다. 원래 아이디어는 1900년 1월 1일을 날짜 1로 잡기 위해, 1899년의 마지막 날을 0으로 잡겠다는 것이었다. 그러나, 처음으로 이 날짜 표현을 고안한 개발자는 1900년이 윤년이 아니었다는 사실을 잊어버렸었다. 그래서 나중에 계산을 보정하기 위해 하루를 조정해야 했고, 그 결과, 1900년 1월 1일은 날짜 1이 아니라 날짜 2가 되었다.

앞서 언급했듯이, `DateTime`은 컴파일러가 이해하도록 미리 정의된 타입이 아니다. 이 타입은 다음과 같이 `System` 유닛에 정의되어 있다.

```
type
    DateTime = type Double;
```

참고 `System` 유닛은 거의 이 언어의 코어^{core/핵심}에 해당한다고 봐도 좋다. 컴파일 할 때마다 언제나 자동으로 포함되기 때문이다. `uses` 문에 나열하지 않아도 그렇다 (실제로 `uses` 문에 `System` 유닛을 넣으면 컴파일 에러가 발생함). 하지만 엄밀히 말하면 이 유닛은 런타임 라이브러리(RTL)의 핵심 부분이다. `System` 유닛에 대해서는 17장에서 다룰 것이다.

`DateTime` 구조체 ^{structure}의 시간 부분과 날짜 부분을 처리하는 타입인 `TDate`, `TTime`이 있다. 두 타입 모두 전체 `DateTime`의 별칭 ^{alias}이다. 하지만 사용되지 않는 부분은 시스템 함수에 의해 잘려서 처리된다.

날짜 및 시간 데이터 타입은 사용하기가 매우 쉽다. 델파이에는 날짜 및 시간 데이터 타입에서 작동하는 여러 가지 함수가 들어 있기 때문이다. 핵심 함수들 몇 가지는 `System.SysUtils` 유닛 안에, 날짜 및 시간에 특화된 함수들은 `System.DateUtils` 유닛 안에 많이 들어있다 (이름과 달리 시간을 다루는 함수도 여기에 들어 있음).

흔하게 사용되는 날짜/시간 조작 함수를 모아 놓은 짧은 목록은 아래와 같다.

<code>Now</code>	현재 날짜와 시간을 날짜/시간 값으로 반환한다
<code>Date</code>	현재 날짜만 반환한다
<code>Time</code>	현재 시간만 반환한다
<code>DateTimeToStr</code>	날짜 및 시간 값을 기본 ^{default} 형식을 사용하여 문자열로 변환한다. 변환을 더 세밀하게 제어하려면 대신 <code>FormatDateTime</code> 함수를 사용한다
<code>DateToStr</code>	날짜/시간 값의 날짜 부분을 문자열로 변환한다
<code>TimeToStr</code>	날짜/시간 값의 시간 부분을 문자열로 변환한다
<code>FormatDateTime</code>	지정된 형식을 사용해 날짜와 시간을 문자열로 변환한다. 무슨 값을 표시할 것인지 어떤 형식을 사용할 것인지를 지정할 수 있다
<code>StrToDateTime</code>	날짜/시간 정보가 담긴 문자열을 날짜/시간 값으로 변환한다. 문자열 형식에 오류가 있으면 예외 ^{exception} 를 발생시킨다. 동반 함수인 <code>StrToDateTimeDef</code> 는 오류 발생 시 예외를 발생시키지 않고 기본값 ^{default value} 을 반환한다
<code>DayOfWeek</code>	파라미터로 전달된 날짜/시간 값의 요일에 해당하는 숫자를 반환한다 (설정된 로컬이 반영된다)

DecodeDate	날짜/시간 값에서 연도, 월, 일 값을 추출한다
DecodeTime	날짜/시간 값에서 시, 분, 초, 밀리초를 추출한다
EncodeDate	연도, 월, 일 값을 날짜/시간 값으로 변환한다
EncodeTime	시, 분, 초, 밀리초 값을 날짜/시간 값으로 변환한다

이 데이터 타입 그리고 여기에 관련된 루틴을 사용하는 방법을 보여 주기 위해 간단한 예제를 만들었다. 이 예제 프로그램을 실행하면 자동으로 현재 시간과 날짜를 계산하여 표시한다. 주요 코드는 다음과 같다. (TimeNow 예제에서 발췌함)

```
var
  StartTime: TDateTime;
begin
  StartTime := Now;
  Show( '현재 시간: ' + TimeToStr(StartTime));
  Show( '현재 날짜: ' + DateToStr(StartTime));
```

첫 번째 문장은 Now 함수를 호출한다. 이 함수는 현재 날짜와 시간을 반환한다. 값은 StartTime 변수 안에 저장된다.

참고 오브젝트 파스칼에서는 파라미터가 없는 함수를 호출할 때 빈 괄호를 적지 않아도 된다. 이는 C 스타일 언어들과 다른 점이다.

다음 두 줄은 TDateTime 값이 문자열로 변환된 결과다. 시간 부분과 날짜 부분을 각각 표현하고 있다. 위 코드가 출력하는 결과인데, 시스템의 로캘 [locale](#) 설정에 따라 달라질 수 있다.

```
현재 시간: 오후 6시 33분 14초
현재 날짜: 2020-10-07
```

이 프로그램을 컴파일하려면, System.SysUtils(“system utilities”의 줄임말) 유닛에 들어 있는 함수들을 참조해야 한다. TimeToStr 및 DateToStr 을 호출하는 것 외에도 더 강력한 함수인 FormatDateTime 를 사용할 수도 있다.

시간/날짜 값은 문자열로 변환될 때 시스템의 국제 설정을 따른다는 점에 유의하자. 날짜 및 시간 형식 지정 정보는 작동 중인 시스템에서 읽어오는데, 이는 운영체제 및 로캘에 따라 다르다. 읽어온 설정 정보는 TFormatSettings 라는 데이터 구조를 채운다. 맞춤 형식을 지정하고 싶다면, 이 타입을 사용해 사용자 정의 구조를 만들고, 그 구조를 날짜-시간 형식을 적용하는 함수 대부분에게 파라미터로 전달하면 된다.

참고 TimeNow 프로젝트에 있는 두 번째 버튼은 타이머를 하나 활성화한다. Timer는 시간이 지남에 따라 이벤트 핸들러 [event handler](#)를 자동으로 실행하는 컴포넌트다(타이머의 간격은 사용자가 지정). 데모에서 버튼을 누르면 타이머가 활성화된다. 그래서 목록에 매초마다 현재 시간이 추가되는 것을 볼 수 있다. 더 유용한 UI [사용자 인터페이스](#)를 생각해 본다면, 레이블을 하나 놓고 그 레이블을 매초마다 현재 시간으로 업데이트하여 시계를 만들 수도 있을 것이다.

날짜 시간 헬퍼 Date Time Helper

TDateTime 데이터 타입을 가지고 더 쉽게 작업할 수 있도록, 델파이 11 에는 이 타입에 특화된 헬퍼가 도입됐다. 이 장 앞부분에서 본 네이티브 native 데이터 타입용 헬퍼와 비슷하다. TDateTime 용 레코드 헬퍼의 이름은 TDateTimeHelper 다. System.DateUtils 유닛에 정의되어 있다. 수행할 수 있는 작업은 월이나 연도의 첫 번째 날짜 가져오기, 유닉스 날짜 형식으로 변환, 오전/오후 확인, 윤년 여부 확인 등이다. 이 레코드 헬퍼에 있는 메서드는 150 개가 넘기 때문에 여기에 모두 소개하는 것은 의미가 없다.

TDateTime 헬퍼 타입에는 기존 RTL 에 없던 새로운 NowUTC 연산(UTC 표준시 기준 현재 시간)도 도입되었다. 다음은 샘플 코드 조각이다. 헬퍼 두 개 즉, Tomorrow 와 ToString 를 이어서 호출하는 것을 보여준다.

```
uses
    DateUtils;

procedure TForm1.Button1Click(Sender: TObject);
begin
    var MyDate: TDateTime := TDateTime.NowUTC;
    MyDate.Tomorrow.ToString;
end;
```

타입 캐스팅 Casting 및 타입 변환 Conversion

우리가 지금까지 본 것에 따르면, 여러분은 한 데이터 타입의 변수를 다른 타입의 변수에 할당할 수 없다. 그 이유는, 데이터의 실제 표현에 따라 다를 수 있겠지만, 의미 없는 결과를 얻게 될 수 있기 때문이다.

이제부터는, 모든 데이터 타입에서 다 그렇지 않는다는 것을 보자. 예를 들어, 숫자 타입은 언제나 안전하게 승격될 promoted 수 있다. "승격 promotion"이란 내부 표현이 더 큰 타입으로 항상 안전하게 값을 할당할 수 있다는 의미다. 따라서 Word 를 Integer 에 할당할 수 있고, integer 를 Int64 값에 할당할 수 있다. 반대 연산 즉 "강등 demotion"은 컴파일러가 허용하지만 경고가 표시될 수 있다. 데이터의 일부만 담길 수 있기 때문이다. 그 외 다른 자동 변환은 한 방향으로만 허용된다. 예를 들어, 정수를 부동 소수점 숫자에 할당할 수 있다. 하지만 그 반대의 연산은 허용되지 않는다.

값의 타입을 변경하고 싶은 상황이 있다. 여러분이 그 작업하는 방법은 두 가지다. 하나는 직접 타입 캐스트 cast 를 수행하는 것이다. 이 방식은 물리적으로 데이터를 복사한다. 그 결과 올바른 변환이 될 수도 그렇지 않을 수도 있다. 이는 그 타입들과 값들에 의해 결정된다. 타입 캐스트를 수행한다면, 여러분은 컴파일러에게 "나는 내가 뭘 하고 있는지 알고 있으니 그냥 내버려둬라"고 말하는 것이다. 만약 타입 캐스트를 하는데, 여러분이 사실 뭘 하고 있는지 명확히 알지 못하다면 곤경에 처할 수 있다. 컴파일러의 타입 검사라는 안전망을 벗어났기 때문이다.

타입 캐스팅 함수 표기는 간단하다. 목표 데이터 타입 이름을 함수로 사용하면 된다.

```
var
  I: Integer;
  C: Char;
  B: Boolean;

begin
  I := Integer('X');
  C := Char(I);
  B := Boolean(I);
```

데이터 타입의 크기가 같은 것(데이터를 표현하는 바이트 **byte** 개수가 같은 것 - 위의 코드 조각은 그렇지 않음!)끼리라면 안전하게 타입 캐스팅을 할 수 있다. 대체로 순서 **ordinal** 타입끼리 타입 캐스팅을 하는 것은 안전하다. 포인터 (그리고 당연히 오브젝트) 타입끼리 타입 캐스팅을 할 수도 있다. 자신이 뭘 하고 있는지 알고 있다면 말이다.

직접 타입 캐스팅은 위험한 프로그래밍 관행 **practice** 이다. 여러분이 그 값을 마치 뭔가 다른 것을 나타내는 것이라고 생각하고 접근하는 것을 허용하기 때문이다. 데이터 타입의 내부 표현은 대체로 일치하지 않는다 (그리고 심지어 타겟 플랫폼에 따라 다를 수도 있다). 따라서 추적하기 어려운 오류를 실수로 만들 수 있다는 위험이 있다. 그러니 여러분은 *타입 캐스팅을 대체로 피해야 한다*.

변수를 다른 타입에 할당하기 위한 다른 방법이 있다. 타입 변환 **conversion** 함수를 쓰면 된다. 다양한 기본 타입들을 서로 변환할 수 있는 함수를 요약한 목록은 아래와 같다 (그 중 몇 가지 함수는 이미 이 장의 데모에서 사용한 것들이다).

Chr	순서를 나타내는 숫자를 받아서 문자로 변환한다
Ord	순서 타입의 값을 받아서 그 순서를 나타내는 숫자로 변환한다
Round	실수-타입 값을 반올림하여 정수-타입 값으로 변환한다 (아래 참고를 볼 것)
Trunc	실수-타입 값의 소수점 부분을 잘라 내고 정수 타입 값으로 변환한다
Int	부동 소수점 값을 받아서 그 중 정수 부분을 반환한다
FloatToDecimal	부동 소수점 값을 받아서 레코드 record 로 변환한다. 그 레코드에는 (지수 exponent , 숫자 digit , 기호 sign 가 따로 담긴다)
FloatToStr	기본 default 형식을 사용하여 부동 소수점 값을 문자열 표현으로 변환한다
StrToFloat	문자열을 부동 소수점 값으로 변환한다

참고 반올림 함수의 구현은 CPU에서 제공하는 네이티브 **native** 구현을 기반으로 한다. 최신 프로세서는 일반적으로 중간 값(예: 5.5 또는 6.5)이 홀수인지 짝수인지에 따라 위아래로 반올림하는 소위 "뱅크어 반올림 **Banker's Rounding**"을 채택한다. 다른 반올림 함수들도 있다. **RoundTo**와 같은 함수는 실제 연산을 더 세밀하게 제어할 수 있다.

이 장의 앞부분에서 언급했듯이, 이러한 변환 함수 중 일부는 데이터 타입에서 바로 연산하는 방식으로 사용할 수 있다 (타입 헬퍼 메커니즘 덕분임). **IntToStr** 과 같은

고전적인 변환도 있지만, 숫자 타입 대부분은 문자열 표현으로 변환할 때 ToString 연산을 직접 적용할 수 있다. 타입 헬퍼를 사용하여 변환을 직접 변수에 적용할 수 있는 것들은 많다. 그리고 이런 코딩 스타일을 타입 캐스팅보다 선호하는 것이 좋다.

위 표에 있는 루틴 중 일부는 데이터 타입에서 작동한다. 이것은 뒤에서 설명한다. 알아 둘 점이 있다. 위 표에는 특수 타입(예: TDateTime 또는 Variant)에 대한 루틴들은 빠져 있다. 또한 변환 [conversion](#) 보다는 형식 지정 [formatting](#)에 특별히 중점을 두고 있는 루틴들(Format, FormatFloat 루틴 등)도 빠져 있다.

03: 언어의 문장 language statement

데이터 타입이라는 강력한 개념은 처음 파스칼 프로그래밍 언어가 발명되었을 당시 이 언어의 획기적인 특징이었다. 프로그램에는 데이터 타입 선언 `data type declaration` 들과 프로그래밍 문장 `programming statement` 들이 들어간다. 문장 `statement` 은 그 데이터 타입들을 다룬다.

파스칼 언어가 발명될 무렵, 이 두 기둥(데이터 타입과 프로그램 명령 `program instruction`)에 대한 개념에 대해 니콜라우스 비르트 `Niklaus Wirth` 는 자신의 저서에서 명확히 설명했다. 책 제목은 "알고리즘 + 데이터 구조 = 프로그램 `Algorithms + Data Structures = Programs`"이며, 1976 년 2 월 Prentice Hall 에서 발간했다(여전히 인쇄되고 있는 고전이다). 이 책은 객체-지향 `object-oriented` 프로그래밍보다 몇 년 앞서 출간되었지만, 현대 프로그래밍의 기반 중 하나라고 여겨질 수 있다. 데이터 타입이라는 강력한 신념이 그 기반이기 때문이다. 이런 방식은 객체 지향 프로그래밍 언어를 이끌어 낸 기반 개념들 중 하나다.

프로그래밍 언어에서 문장 `statement` 을 구성하는 것은 키워드들과 기타 요소들이다. 여러분은 이것들을 사용해 컴파일러에게 일련의 연산을 수행하도록 알려줄 수 있다. 문장 `t` 들은 주로 프로시저나 함수 안에 들어간다. 보다 자세한 내용은 다음 장부터 살펴보기로 하고, 지금은 프로그램 작성을 위한 기본적인 명령 `instruction` 들만 집중한다. 1 장(공백과 코드 형식을 다루는 부분)에서 본 것처럼 프로그램 코드는 실제로 매우 자유롭게 작성할 수 있다. 앞 장에서 주식과 특수 요소 몇 가지를 다루긴 했지만, 핵심 개념을 충분히 소개하지는 못했다. 이제 언어의 핵심 개념 중 프로그래밍 문장에 대해 알아보자.

단순 문장 Simple Statement과 복합 문장 Compound Statement

프로그래밍 명령은 일반적으로 **문장 statement** 이라고 부른다. 프로그램 블록 하나는 **여러** 문장들로 구성될 수 있다. 문장에는 두 가지 유형이 있다. 즉 단순 문장 Simple Statement 과 복합 문장 Compound Statement 이 있다.

단순 문장은 다른 하위 문장을 전혀 담고 있지 않은 문장이다. 할당 문, 프로시저 호출 등은 단순 문장에 해당된다. 오브젝트 파스칼에서 단순 문장은 **세미콜론**으로 구분한다.

```
X := Y + Z; // 할당
Randomize; // 프로시저 호출
...
```

복합 문장을 정의하려면, 키워드 **begin** 과 **end** 안에 여러 문장을 담으면 된다. **begin** 과 **end** 는 (여러 문장을 담는) 그릇 역할을 한다. C 에서 파생된 언어들에서 사용하는 중괄호와 비슷하다. 하지만 똑같지는 않다. 오브젝트 파스칼에서는 단순 문장이 들어갈 수 있는 곳이라면 복합 문장 역시 들어갈 수 있다. 다음은 예제다:

```
begin
  A := B;
  C := A * 2;
end;
```

위 복합 문장의 마지막 문장에 붙은 세미콜론(**end** 바로 앞에 있는 것)은 없어도 된다.

```
begin
  A := B;
  C := A * 2
end;
```

따라서, 위 두 버전 모두 옳다. 첫 번째 버전의 마지막 세미콜론은 없어도 된다 (하지만 있어도 무해하다). 이 세미콜론은 아무것도 없는 **null** 문장, 즉 빈 **empty** 문장이다. 다시 말해 문장인데 아무 코드도 없다. 이는 다른 많은 프로그래밍 언어 (예: C 구문 **syntax** 이 기반인 언어)들과 크게 다른 점이다. 다른 언어에서, 세미콜론은 문장을 **종결하는 표시**이다(문장을 구분하는 표시가 아니다), 따라서 모든 문장의 끝에 반드시 붙여야 한다.

가끔은, 아무것도 없는 문장 **null-statement** 이 사용된다는 점을 알아두자. 특히 루프 **loop/순환** 안에서 사용되거나 또는 기타 특별한 경우, 문장이 필요한 곳에서 사용된다. 다음 예문과 같다.

```
while condition_with_side_effect do
  ; // 아무것도 없는(null) 문장, 즉 빈(empty) 문장
```

비록 이런 마지막 세미콜론은 의미가 없긴 하지만, 개발자 대부분이 그렇게 사용하고 있고, 나 또한 그렇게 하기를 권한다. 가끔, 코드 몇 줄을 작성해 놓은 곳에, 문장을

더 추가하고 싶을 때가 있다. 만약 마지막 세미콜론을 적어 놓지 않았다면, 먼저 그 세미콜론부터 붙이고 나서 추가할 문장을 적어야 한다는 것을 기억하고 있어야 한다. 그러므로 아예 처음부터 붙이는 것이 더 좋다. 곧 살펴보겠지만, 세미콜론을 덧붙이는 규칙에는 한 가지 예외가 있다. 조건 안에서 다음 요소가 `else` 문인 경우다.

If 문 The If Statement

조건문 [conditional statement](#) 을 사용하면, 문장들 중에서 하나만 실행하거나, 또는 아무것도 실행하지 않을 수 있다. 특정 테스트(즉 조건)에 따라 결정된다. 기본적인 조건문은 두 가지다. 즉 `if` 문과 `case` 문이 있다.

`if` 문은 특정 조건이 충족된 경우만 문장을 실행하거나(`if-then`), 대안 두 개 중 하나를 선택(`if-then-else`)하는데 사용한다. 조건은 불리언 표현식 [Boolean expression](#) 으로 정의한다.

조건문을 작성하는 방법을 예제로 보자 (IfTest 예제임). 이 프로그램에는 체크박스를 하나 두고, 사용자가 체크 표시를 했는지 알기 위해 체크박스의 `IsChecked` 프로퍼티를 읽는다 (그리고 받은 결과를 임시 변수에 저장한다. 임시 변수가 꼭 필요하지는 않다. 조건 표현식 안에서도 그 프로퍼티의 값을 직접 확인할 수도 있기 때문이다).

```
var
  IsChecked: Boolean;
begin
  IsChecked := CheckBox1.IsChecked;
  if IsChecked then
    Show( '체크박스가 선택되어 있습니다');
```

체크박스에 체크가 되어 있다면, 프로그램은 메시지를 표시한다. 그렇지 않다면 아무 일도 일어나지 않는다. 비교를 위해, 똑같은 문장을 C 언어 구문 [syntax](#) 으로 작성하면 다음과 같다 (조건 표현식을 괄호로 감싸야 함).

```
if (IsChecked)
  Show("체크박스가 선택되어 있습니다");
```

일부 다른 언어에는 `endif` 요소라는 개념이 있어서 여러 문장을 써넣을 수 있다. 하지만, 오브젝트 파스칼 구문 [syntax](#) 상 조건문은 단일 [single](#) 문장이 기본이다. 동일 조건 안에서 실행할 문장이 두 개 이상이면 `begin-end` 블록을 사용한다.

조건에 따라 다른 연산을 수행하려면, `if-then-else` 문을 사용한다 (아래 코드는 체크박스의 상태를 표현식 안에서 직접 읽도록 했다).

```
// if-then-else 문
if CheckBox1.IsChecked then
  Show( '체크박스가 선택되어 있습니다')
else
  Show( '체크박스가 선택되지 않았습니다');
```


첫 번째 문장의 끝과 `else` 키워드 사이에 세미콜론을 넣을 수 없다는 점을 유의하자. 세미콜론을 넣으면 컴파일러가 구문 오류를 발생시킨다. `if-then-else` 문은 하나의 문장이기 때문이다. 따라서 그 중간에 세미콜론을 넣을 수 없다.

`if` 문은 매우 복잡할 수도 있다. `if` 문 안에 (불리언 연산자인 `and`, `or`, `not` 등 사용해) 여러 조건들을 나열할 수 있다. 또는 `if` 문 안에 또 다른 `if` 문을 중첩할 수도 있다. 중첩 `if` 문 말고도, 여러 조건들이 구분되는 경우에는 대체로 `if-then-else-if-then` 문을 이어가는 경우가 흔하다. `else-if` 조건은 얼마든지 원하는 만큼 연결할 수 있다.

아래 코드는 에디트 박스 `edit box` 안에 입력된 첫 글자를 확인한다. 사용자가 입력하기 때문에, 글자가 없을 수도 있다. (IfTest 예제에 있는 세 번째 버튼의 동작을 발췌함).

```
var
  AChar: Char;
begin
  // 중첩된 If 문 여러 개
  if Edit1.Text.Length > 0 then
    begin
      AChar := Edit1.Text.Chars[0];

      // 소문자 여부 확인 (조건 두 가지를 사용)
      if (AChar >= 'a') and (AChar <= 'z') then
        Show( '소문자입니다' );

      // 후속 조치 조건
      if AChar <= Char(47) then
        Show( '숫자나 영문자보다 더 앞에 있는 심볼입니다' )
      else if (AChar >= '0') and (AChar <= '9') then
        Show( '숫자입니다' )
      else
        Show( '숫자도 아니고 그보다 더 앞에 있는 심볼symbol도 아닙니다' );
    end;
```

코드를 주의 깊게 살펴보고 프로그램을 실행하여, 잘 이해했는지 확인해 보라(그리고 비슷한 프로그램을 작성해 보면 더 많은 것을 배울 수 있다). 더 많은 옵션들 그리고 더 많은 불리언 표현식 `Boolean expression` 을 생각하면서 이 작은 예제를 더 복잡하게 만들고 해보고 싶은 테스트를 할 수도 있을 것이다.

Case 문 Case Statement

`If` 문이 매우 복잡하다면, 또는 순서 `ordinal` 값을 테스트한다면, `case` 문으로 대체하는 것을 생각해 볼 수 있다. `case` 문에는 값을 선택하기 위한 표현식 그리고 해당될 수 있는 값(또는 값의 범위)들의 목록이 명시된다. 목록의 값들은 상수 `constant` 이며 고유하고 `unique` 순서 타입 `ordinal type` 이어야 한다. 맨 뒤에는, `else` 문을 적을 수 있다. `else` 문은 해당될 수 있는 값 목록 안에 있는 어느 값도 맞지 않는 경우에 실행된다.

endcase 문은 없다. case는 항상 end로 끝난다 (이 end는 블록 종결자가 아니다. 짝을 이루는 begin이 없기 때문이다).

참고 case 문을 만들려면, 순서^{ordinal} 값이어야 한다. 문자열^{string} 값을 바탕으로 하는 case 문은 현재 허용되지 않는다. 이 경우라면, 중첩된 if 문을 사용하거나, 또는 딕셔너리^{dictionary} 등 다른 데이터 구조를 사용해야 한다 (14장에서 설명한다).

예제를 보자(CaseTest 예제에서 발췌함). 입력된 숫자를 테스트한다. 입력은 숫자 입력 컨트롤인 NumberBox 컨트롤을 사용한다.

```
var
  ANumber: Integer;
  AText: string;
begin
  ANumber := Trunc(NumberBox1.Value);
  case ANumber of
    1: AText := '일';
    2: AText := '이';
    3: AText := '삼';
  end;
  if AText <> '' then
    Show(AText);
```

예제를 하나 더 보자. 앞에서 본 복잡한 if 문을, case 테스트 조건들로 바꿨다.

```
case AChar of
  '+' : AText := '더하기 기호';
  '-' : AText := '빼기 기호';
  '*', '/': AText := '곱하기 또는 나누기';
  '0'..'9': AText := '숫자';
  'a'..'z': AText := '소문자';
  'A'..'Z': AText := '대문자';
  #12032..#12255: AText := '강희자전 부수';
else
  AText := '기타 문자: ' + AChar;
end;
```

참고 위에 작성된 코드 조각을 잘 보자. 값의 범위는 하위범위^{subrange} 데이터 타입을 표현하는 구문^{syntax}과 정의하는 방식이 똑같다. 단일 값을 여러 개를 나열할 때는 쉼표로 구분한다. 강희자전 부수 구역은 실제 문자가 아니라 숫자 값을 사용했는데, 그 이유는 IDE 에디터에서 사용하는 고정 크기^{fixed-size} 글꼴이 여기에 해당하는 심볼^{symbol}을 제대로 표시하지 못하기 때문이다. (옳긴하: 일반적인 한글 글꼴에서는 대체로 잘 표현된다. 한종일 문자이기 때문).

else 구역을 넣는 것은 좋은 관행^{practice}이라고 여겨진다. 정의해놓지 않은 즉 예상치 못한 조건을 else가 잡아 주기 때문이다. 오브젝트 파스칼에서 case 문은 단일 실행 경로를 채택하고 있다. 즉, 조건에 맞는 값의 위치는 진입 지점이 아니다. 조건에 맞는 값 바로 뒤에 붙은 콜론 뒤에 나오는 문장 또는 블록을 다 실행하면 그 뒤를 건너 뛰고 case 문의 맨 끝으로 이동한다.

C 언어(및 거기에서 파생된 언어)와 매우 다른 점이니 주의하자. C 계열 언어들은 switch 문에 있는 분기들을 진입 지점으로 삼고 그 뒤의 모든 문장들을 실행한다. 빠져나오려면 break 를 적어 주어야 한다 (비록 실제 구현에서 Java 와 C#이 서로 조금 다르지만 말이다). C 언어 구문 [syntax](#) 은 다음과 같다:

```
switch (aChar) {
  case '+': aText = "더하기 기호"; break;
  case '-': aText = "빼기 기호"; break;
  ...
  default: aText = "알 수 없음"; break;
}
```

For 루프 [loop/순환](#)

대부분의 프로그래밍 언어에 있는 전형적인 반복 [repetitive](#) 또는 루프 [loop/순환](#)를 문장들이 오브젝트 파스칼 언어에도 있다. for, while, repeat 문이 있으며, 보다 현대적인 for-in (즉 *for-each*) 반복문도 있다. 이미 다른 프로그래밍 언어를 써 본 적이 있다면 루프 대부분이 익숙할 테니 간략히 다루겠다 (다른 언어와의 주요 차이점 위주로 설명한다).

오브젝트 파스칼에서, for 루프는 카운터 [counter](#) 를 기반으로 순환한다. 카운터는 루프가 실행될 때마다 증가 또는 감소한다. 아래 예제는 for 루프를 사용해 숫자를 10 까지 더한다 (ForTest 예제에서 발췌함).

```
var
  Total, I: Integer;
begin
  Total := 0;
  for I := 1 to 10 do
    Total := Total + I;
  Show(Total.ToString);
```

출력되는 결과는 55 다. 인라인 변수가 도입된 덕분에 for 루프를 또 다른 방법으로 작성할 수 있다. 즉 루프 카운터 [loop counter](#) 변수 선언을 for 루프 선언문 안에 넣을 수 있다(그 구문은 C 및 그 파생 언어의 for 루프와 비슷하다. 뒤에서 다루겠다).

```
for var I: Integer := 1 to 10 do
  Total := Total + I;
```

이 방식은, 타입 추론 [type inference](#) 을 활용할 수 있다는 장점도 활용할 수 있다. 즉 타입 명시를 생략할 수 있다. 그 결과, 위 코드 조각 전체는 아래와 같이 작성될 수 있다.

```
var
  Total: Integer;
begin
  Total := 0;
  for var I := 1 to 10 do
    Total := Total + I;
  Show(Total.ToString);
```


인라인 [inline](#) 루프 카운터를 사용할 때의 장점은 그 카운터 변수의 범위가 그 루프 안으로만 제한된다는 것이다. `for` 문 바깥에서 사용하면 오류가 발생한다.

파스칼의 `for` 루프는 다른 언어에 비해 유연성이 낮다 (증분 [increment](#) 을 오직 1 씩만 할 수 있어서 다른 숫자를 지정할 수 없음). 하지만, 간단하고 이해하기 쉽다. 비교할 수 있도록, C 언어 구문 [syntax](#) 으로 똑같은 `for` 루프를 작성해 보았다.

```
int total = 0;
for (int i = 1; i <= 10; i++) {
    total = total + i;
}
```

이런 언어에서는, 증분이 하나의 표현식이며, 모든 종류의 순번 [sequence](#) 을 지정할 수 있다. 따라서, 아래와 같은 코드가 가능하다. 아래 코드는 읽을 수 없다고 생각하는 사람들이 많을 것이다.

```
int total = 0;
for (int i = 10; i > 0; total += i--) {
    ...
}
```

오브젝트 파스칼에서는, 이와 달리, 한 단계씩만 증분 할 수 있다. 더 복잡한 조건을 테스트하고 싶거나, 사용자 지정 카운터를 쓰고 싶다면, `for` 루프 대신 `while` 문 또는 `repeat` 문을 사용해야 한다.

`for` 루프에서 단일 증분 이외에 사용할 수 있는 유일한 대안은 단일 감소 즉 `downto` 키워드를 사용한 역방향 반복이 유일하다.

```
var
    Total, I: Integer;
begin
    Total := 0;
    for I := 10 downto 1 do
        Total := Total + I;
```

참고 역방향 카운팅은 유용하다. 루프를 통해 목록 기반 데이터 구조 안에 영향을 주는 경우를 예로 들어 보자. 목록에 있는 항목 몇 개를 삭제하는 경우라면 역방향으로 순환하고 싶을 것이다. 만약 정방향으로 순환하면 작업하고 있는 순번 [sequence](#)에 영향이 갈 수 있기 때문이다 (즉, 목록에서 세 번째 요소를 삭제하면, 그 목록의 네 번째 요소는 세 번째 요소가 된다. 그런데, 현재 삭제 작업 중인 순번이 세 번째이므로 다음 요소 (네 번째 요소)로 넘어가면 실제로 다섯 번째 요소를 가지고 작업하게 된다. 즉 요소 하나를 건너뛰게 된다).

오브젝트 파스칼에서는, `for` 루프의 카운터는 반드시 숫자일 필요가 없다. 순서 [ordinal](#) 타입 값이면 뭐든지 카운터가 될 수 있다. 즉, 문자, 열거 타입 등을 사용할 수 있다. 그 결과, 더 읽기 쉬운 코드를 작성하는데 도움이 된다. `for` 루프에서 `Char` 타입을 바탕으로 사용하는 예제를 보자.


```

var
  AChar: Char;
begin
  for AChar := 'a' to 'z' do
    Show(AChar);

```

위 코드는 영어 문자 집합 안에 들어 있는 모든 문자들을 메모 컨트롤 안에서 각 줄에 하나씩 표시한다 (ForTest 프로젝트에서 발췌함).

참고 이미 비슷한 예제를 보여준 적이 있다. 그 때는 숫자 리터럴을 사용하고, 문자를 합쳐 문자열 하나를 만들었다 (2장의 CharsTest 예제).

아래 코드 조각은 사용자가 지정한 열거 값을 바탕으로 하는 for 루프이다.

```

type
  TSuit = (Club, Diamond, Heart, Spade);
var
  ASuit: TSuit;
begin
  for ASuit := Club to Spade do
    ...

```

위 코드의 마지막에 있는 루프는 TSuit 데이터 타입의 모든 요소를 하나씩 순환한다. 위와 같이 타입의 요소를 명시적으로 적어서 순환의 범위를 지정하는 편이 더 좋을 수도 있다 (정의 안에서 수정할 수 있기 때문에 더 유연하다.). 하지만 아래와 같이 그 타입의 첫 번째 요소와 마지막 요소를 가리키도록 코드를 쓰는 것도 가능하다.

```

for ASuit := Low(TSuit) to High(TSuit) do

```

위와 비슷하게, 데이터 구조 안에 있는 모든 요소를 다루기 위해 for 루프를 작성하는 것은 매우 일반적이다. 예를 들어, 문자열 하나 안에 있는 모든 문자들을 다루고 싶다면 아래와 같이 할 수 있다 (ForTest 프로젝트에서 발췌함).

```

var
  S: string;
  I: Integer;
begin
  S := '안녕하세요';
  for I := Low(S) to High(S) do
    Show(S[I]);

```

만약, 여러분이 그 데이터 구조의 첫 번째 요소와 마지막 요소를 가리키는 방식을 원하지 않는다면, 대신 for-in 루프 즉, 특별한 목적을 가진 for 루프를 사용할 수 있다. 다음 소단원에서 설명한다.

참고 컴파일러가 문자열 [string](#) 데이터 직접 읽기를 [] 연산자를 사용해 어떻게 다루는지 그리고 문자열 경계의 하한과 상한을 어떻게 파악하는지는 오브젝트 파스칼에서 다소 복잡한 주제이다. 현재 기본 [default](#) 방식은 모든 플랫폼에서 동일하다. 이 주제는 6장에서 다루겠다.

0 기반 인덱싱을 사용하는 데이터 구조라면, 여러분은 루프를 인덱스 0 에서 시작하고, 그 데이터 구조의 크기보다 1 이 작은 곳에서 끝내고 싶을 것이다. 방법은 다음과 같다.

```
for I := 0 to Count - 1 do ...
for I := 0 to Pred(Count) do ...
```

for 루프와 관련하여 마지막으로 말해줄 것이 있다. 루프가 종료된 후 루프 카운터에 무슨 일이 생길까? 간단히 말해, 그 값은 *지정되지 않음* *unspecified* 이 된다. 그리고 만약 여러분이 루프가 종료된 후에 그 루프 카운터를 사용하려고 하면 컴파일러는 경고를 표시한다. 인라인 변수를 루프 카운터로 사용하면 그 변수는 오직 루프 안에서 정의되고 루프의 end 문 뒤에서는 접근할 수 없다는 장점이 있다. 따라서 이 경우에 컴파일러는 오류를 낸다(따라서 더 강력하게 보호할 수 있다).

```
begin
  var Total := 0;
  for var I: Integer := 1 to 10 do
    Inc(Total, I);
  Show(Total.ToString);
  Show(I.ToString); // 컴파일러 예러: Undeclared Identifier 'I'
```

for-in 루프

오브젝트 파스칼에는 리스트 *list* 또는 컬렉션 *collection* 안 모든 요소를 순환하는 루프 *loop* 구조가 있는데, 바로 for-in(다른 프로그래밍 언어에서는 *for each* 라고 함)이다. 이 for 루프는 배열 *array*, 리스트 *list*, 문자열 *string*, 기타 컨테이너 타입 안에 들어 있는 각 요소를 순환하며 작동한다. C#과 달리, 오브젝트 파스칼은 IEnumerator 인터페이스를 구현하도록 요구하지 않는다. 하지만 내부 구현은 서로 어느 정도 비슷하다.

참고 클래스가 for-in 루프를 지원하는 방법 즉 맞춤 열거 *custom enumeration* 지원 클래스 안에 추가하는 방법에 대한 기술적 세부 사항은 10장에 있다.

아주 간단한 컨테이너 *container* 부터 시작해보자. 문자열 *string* 은 문자 *character* 들이 모여 있는 컬렉션이다. 앞에서 보았던 for 루프를 사용한 문자열 안의 모든 요소에 대한 연산과 똑같은 효과를 for-in 루프로 구현할 수 있다. 아래 for-in 루프에 있는 ch 변수는 문자열 안에 담긴 각 요소를 받아낸다 (ForTest 예제에서 발췌함).

```
var
  S: string;
  Ch: Char;
begin
  S := '안녕하세요';
  for Ch in S do
    Show(Ch);
```

전통적인 for 루프를 사용하는 것에 비해 좋은 점이 있다. 문자열 맨 앞에 있는 요소가 무엇인지, 맨 뒤 요소의 위치를 어떻게 추출하는지를 기억할 필요가 없다. 이 루프는 작성하고 유지 관리하기가 더 쉬우면서도 효율성 면에서는 비슷하다.

전통적인 for 루프와 마찬가지로, for-in 루프도 인라인 변수를 선언하여 그 혜택을 활용할 수 있다. 위의 코드를 인라인 변수를 사용하여 다시 작성하면 다음과 같다.

```
var
  S: string;
begin
  S := '안녕하세요';
  for var Ch: Char in S do
    Show(Ch);
```

for-in 루프를 써서 내부 요소를 접근할 수 있는 데이터 구조들은 여러 가지다.

- 문자열 string 안에 있는 문자(위 코드 조각을 참조)
- 세트 set의 활성 값
- 정적 static 또는 동적 dynamic 배열 array의 항목 (2차원 배열도 포함됨, 5장에서 다룸)
- GetEnumerator를 지원하는 클래스들이 참조하고 있는 오브젝트들. 이 지원이 미리 정의되어 있는 클래스들이 참조하는 오브젝트들도 포함됨 (예: 스트링리스트 안에 있는 문자열들), 컨테이너 클래스 안에 담겨 있는 요소들, 폼 form이 소유 own하고 있는 컴포넌트 component들 등이다. 이를 구현하는 방법은 10장에서 설명한다.

이러한 수준 높은 사용 패턴을 지금 다루기에는 조금 어려우니, 책의 뒷부분에 가서 이 루프의 여러 예제들을 다시 살펴보겠다.

참고 for-in 루프는 일부 언어(예: 자바스크립트)에서 실행 속도가 매우 느리다고 혹평을 받는다. 하지만, 오브젝트 파스칼은 그렇지 않다. 오브젝트 파스칼의 for-in 루프는 표준 for 루프와 실행 속도가 같다. 이를 증명하기 위해, LoopsTest 예제를 만들었다. 이 예제에서는 3천만자를 담은 문자열 string을 하나 생성한 후에, 이 두 종류 루프로 스캔(각 반복마다 매우 간단한 연산을 수행)을 하고 타이머 timer로 시간을 잰다. 결과는 약 10% 정도 for 루프가 빠르다 (내 윈도우 컴퓨터에서는 62밀리초 대 68밀리초였다).

While 문 및 Repeat 문

while-do와 repeat-until 루프 loop 둘 모두 그 기본 개념은 같다. 즉 코드 블록 하나를 계속 반복 수행하다가 주어진 조건에 맞으면 멈추는 것이다. 이 두 루프가 다른 점은 조건을 확인하는 위치이다. while 문은 조건 확인을 루프를 시작하는 곳에서 한다. 이와 달리, repeat 문은 끝에서 조건을 확인하므로 적어도 한 번은 실행된다.

참고 대부분의 다른 프로그래밍 언어에는 개방 루프 open loop 문장이 오직 한 가지밖에 없다. 대체로 (이름과 동작 모두) while 루프이다. C 언어 구문에는 파스칼 구문과 마찬가지로 두 가지 옵션 즉 while과 do-while 순환이 있다. 이 두 옵션 모두 똑 같은 논리 조건을 사용한다는 점을 알아야 한다. 이와 달리, repeat-until 루프는 역조건을 가진다.

repeat 루프가 왜 언제나 한 번 이상 실행되는지 그 이유를 쉽게 알 수 있도록 아래 예제를 보자.

```
while (I <= 100) and (J <= 100) do
begin
  // I와 J를 사용하여 뭔가를 계산하기...
  I := I + 1;
  J := J + 1;
end;

repeat
  // I와 J를 사용하여 뭔가를 계산하기...
  I := I + 1;
  J := J + 1;
until (I > 100) or (J > 100);
```

참고 위 예문에서 while과 repeat 둘 다 조건 안에 있는 하위 조건 두 개를 괄호로 에워싸고 있다. 위 상황에서 괄호는 반드시 필요하다. 괄호가 없으면 컴파일러는 하위 조건 비교하기 전에 or 연산을 먼저 실행한다 (2장의 연산자operator 부분에서 이미 다룬 내용임).

I 와 J 중 어느 것이라도 첫 값이 100 보다 큰 경우, while 루프는 완전히 건너뛰는다. 하지만, repeat 루프에서는 그 안에 있는 문장이 적어도 한번은 실행된다.

이 두 루프의 주요 차이가 하나 더 있다. repeat-until 루프는 역조건 reverse condition 이다. repeat 루프는 조건이 충족되지 않는 동안 계속된다. 즉 조건이 충족되었을 때 루프가 종료된다. 이는 조건이 참일 때 실행되는 while-do 루프와 정반대다. 따라서 비슷한 효과를 얻기 위해 위 코드에서는 조건을 while 과 반대로 만들었다.

참고 “역조건”은 공식적으로 “드 모르간De Morgan의 법칙”으로 알려져 있다 (예: 위키백과에 설명되어 있음: https://en.wikipedia.org/wiki/De_Morgan%27s_laws).

루프 예문들 Examples of Loops

실제 예제를 통해 루프를 좀 더 자세히 보자. 이 예제는 고정 fixed 카운터 counter 가 있는 루프와 개방 open 카운터가 있는 루프의 차이점을 강조한다 (LoopsTest 프로그램에서 발췌). 고정 카운터 루프의 첫 번째 예문에서는 for 루프를 사용해 숫자를 순서대로 표시한다.

```
var
  I: Integer;
begin
  for I := 1 to 20 do
    Show( '숫자 ' + IntToStr(I));
  end;
```


while 루프로 같은 결과를 얻을 수 있다. 루프 안에서 카운터를 1 씩 증분 하면 된다 (현재 값을 사용한 후에 카운터를 증분 한다는 점을 유의하자). 또한, while 루프를 쓰면 자유롭게 사용자 정의 증분을 지정할 수 있다. 예를 들어 2 씩 증분할 수 있다.

```
var
  I: Integer;
begin
  I := 1;
  while I <= 20 do
  begin
    Show( '숫자 ' + IntToStr(I));
    Inc(I, 2)
  end;
end;
```

위 코드는 1 부터 19 까지의 모든 홀수를 표시한다.

증분값이 정해진 루프는 논리적으로 동일하며 미리 정의된 횟수만큼 실행된다. 하지만 루프가 항상 이렇지만은 않다. 실행을 미리 확정하기 힘든 루프들이 있을 것이다. 예를 들어 외부 조건에 따라 달라지는 루프들이 있다.

참고 while 루프를 작성할 때는, 조건이 결코 충족되지 못하는 경우를 고려해야 한다. 예를 들어, 위 루프를 작성하면서 루프 카운터를 증가시키는 것을 잊어버리면 무한 루프에 빠지게 된다. 그 결과 그 프로그램은 과부하로 영원히 교착상태에 빠질 것이다. 만약 사용자 또는 운영체제가 그 프로세스 [process](#)를 죽이지 않는다면 CPU를 100% 소모할 가능성이 크다.

미리 확정하기 힘든 루프의 예를 보자. 아래 while 루프는 카운터를 기반으로 한다. 하지만, 그 카운터 증분은 무작위로 정해진다. 그러기 위해, Random 함수를 사용했으며 그 범위 값을 100 으로 지정했다. 따라서, 함수의 결과는 0 에서 99 사이에서 무작위로 선택되는 숫자다. 그 무작위 숫자들이 모여서 이 while 루프의 실행 횟수를 결정한다.

```
var
  I: Integer;
begin
  Randomize;
  I := 1;
  while I < 500 do
  begin
    Show( '무작위 숫자: ' + IntToStr(I));
    I := I + Random(100);
  end;
end;
```

Randomize 프로시저 호출을 추가하는 것을 잊지 말아야 한다. Randomize 는 프로그램 실행마다 무작위 숫자 생성기가 시작되는 지점이 달라지도록 한다. 그 결과, 생성되는 숫자가 달라진다. 아래는 이 프로그램을 두 번 실행한 결과를 나란히 표시한 것이다.

```
무작위 숫자: 1
무작위 숫자: 40
```

```
무작위 숫자: 1
무작위 숫자: 47
```


무작위 숫자: 60	무작위 숫자: 104
무작위 숫자: 89	무작위 숫자: 201
무작위 숫자: 146	무작위 숫자: 223
무작위 숫자: 198	무작위 숫자: 258
무작위 숫자: 223	무작위 숫자: 322
무작위 숫자: 251	무작위 숫자: 349
무작위 숫자: 263	무작위 숫자: 444
무작위 숫자: 303	무작위 숫자: 466
무작위 숫자: 349	
무작위 숫자: 366	
무작위 숫자: 443	
무작위 숫자: 489	

위와 같이, 생성되는 숫자가 매번 다를 뿐만 아니라 출력된 항목의 개수도 달라진다. 이는 while 루프가 실행되는 횟수가 다르기 때문이다. 이 프로그램을 계속해서 여러 번 실행하면, 출력되는 줄의 개수가 계속 달라지는 것을 볼 수 있을 것이다.

Break와 Continue를 사용해 흐름 끊기

서로 차이가 있지만, 각 루프는 한 블록 안에 있는 문장들이 여러 번 실행되게 한다. 몇 가지 규칙을 바탕으로 말이다. 그런데, 동작을 더 추가하고 싶을 수 있다. 그 예로, 주어진 문자가 있는지 확인하는 for 루프를 가정해 보자 (FlowTest 예제에서 발췌함).

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello World';
  Found := False;
  for I := Low(S) to High(S) do
    if S[I] = 'o' then
      Found := True;
```

이 루프가 끝나고, Found의 값을 확인하면 주어진 문자가 문자열의 일부인지를 파악할 수 있다. 그런데 문제가 있다. 프로그램은 주어진 문자를 찾은 후에도 여전히 루프를 반복하면서 해당 문자를 계속 찾는다 (문자열이 매우 길다면 문제가 될 수 있다).

전형적인 대안은, 이것을 while 루프로 바꾸고 두 가지 조건(루프 카운터와 Found 값)을 같이 확인하는 것이다.

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := '안녕하세요';
  Found := False;
  I := Low(S);
  while not Found and (I <= High(S)) do
    begin
```



```

    if S[I] = '하' then
        Found := True;
        Inc(I);
    end;

```

위 코드는 논리적이고 읽기 쉽다. 하지만, 코드를 더 많이 작성해야 한다. 게다가, 조건이 여러 개가 되고 더 복잡해지면, 그 다양한 조건을 모두 묶어서 확인해야 하는데 그러면 코드가 매우 복잡해진다.

이런 이유로, 이 언어(의 런타임 지원)에는 루프가 실행되는 표준 흐름을 바꿀 수 있는 시스템 프로시저 `system procedure` 가 있다.

- Break 프로시저는 루프를 끝내고, 루프 바깥 맨 앞에 있는 문장으로 건너 뛴다. 즉, break 이후에는 그 루프가 더 이상 실행되지 않는다.
- Continue 프로시저는 루프의 조건 테스트 또는 카운터 증분 위치로 건너뛴다. 즉, 루프에서 그 다음 반복부터 계속한다(단, 조건이 더 이상 참이 아니거나 또는 카운터가 최고 값에 도달한다면 루프를 빠져나온다).

일치하는 문자를 찾는 위 루프를 Break 연산을 사용해 다음과 같이 바꿀 수 있다.

```

var
    S: string;
    I: Integer;
    Found: Boolean;
begin
    S := '안녕하세요';
    Found := False;
    for I := Low(S) to High(S) do
        if S[I] = '하' then
            begin
                Found := True;
                Break; // 루프 밖으로 빠져나온다
            end;
    end;

```

시스템 프로시저 두 개를 더 보자. Exit 와 Halt 를 사용하면 수행하고 있는 함수나 프로시저로부터 즉시 나오거나 `return` 프로그램을 종료할 수 있다. Exit 는 다음 장에서 다룰 것이다. 그런데 Halt 는 호출할 이유는 전혀 없다. Halt 는 프로그램을 갑자기 종료하라는 명령이기 때문이다 (따라서 이 책에서는 다루지 않겠다).

Goto 가 온다고? 말도 안 된다!

흐름을 끊는 방법은 앞에서 본 시스템 프로시저 네 가지 이외에도 실제로 더 있다. 최초의 파스칼 언어에는 *악명* 높은 goto 문이 있었다. 그래서 개발자는 소스 코드 안에서 원하는 줄에 이 표식을 달아서, 그 표식이 있는 곳으로 바로 건너가도록 할 수 있었다. 이 방식은 조건문이나 루프와 다르다. 거기에는 코드의 흐름이 순서를 벗어나는 이유가 담겨 있다. 하지만, goto 문은 느닷없이 건너 뛰는 것처럼 보인다. 그러니 결코 사용하지 말자. 오브젝트 파스칼이 goto 문을 지원한다고 내가 말했나?

아니 그렇게 말하지 않았다. 그랬다면 예제 코드를 제시했을 것이다. 나에게 goto 는 이미 오래전에 떠났다.

참고 지금까지 다룬 것 외에도 기타 문장들이 이 언어에 정의되어 있다. 그 중 하나는 with 문이다. with 문은 레코드 [record](#)와 관련이 있으므로 5 장에서 다룬다. with는 '논쟁의 여지가 있는' 특징 중 하나이지만, 여전히 흔하게 사용되고 있기도 하다.

04: 프로시저와 함수Procedures and Functions

오브젝트 파스칼 언어가 강조하는 또 중요한 아이디어는 루틴 [routine](#) 이라는 개념이다 (C 언어도 이와 비슷한 특징이 있다). 기본적으로, 루틴이란 이어진 여러 문장들을 하나로 묶고 거기에 고유한 [unique](#) 이름을 붙인 것이다. 그래서 몇 번이든 작동시킬 수 있다. 루틴(또는 함수 [function](#))을 부를 때는 그 이름을 사용한다. 그러면 같은 코드를 여러 번 작성하지 않아도 되기 때문에, 코드 조각을 단일 버전으로 유지할 수 있다. 그리고, 그것을 프로그램 전반에 걸쳐 많은 곳에서 사용할 수 있다. 이러한 관점에서 보면, 루틴은 기본적인 코드 캡슐화 [encapsulation](#) 메커니즘이라 생각할 수도 있다.

프로시저와 함수Procedures and Functions

오브젝트 파스칼에서, 루틴 [routine](#) 의 형태는 프로시저 [Procedure](#) 와 함수 [Function](#) 두 가지로 볼 수 있다. 이론 상, 프로시저는 컴퓨터에게 수행을 요청하고, 함수는 값을 반환하도록 요청하는 연산이다. 다른 점이라면, 함수에는 결과 [result](#) (반환 값 [return value](#) 즉 반환 타입 [return type](#))이 있지만, 프로시저는 없다는 것이다. C 언어 구문 [syntax](#) 에서는 이 메커니즘이 오직 하나다. 즉 함수만 있다. 그리고 C 에서 `void`(또는 `null`) 결과를 반환하는 함수가 곧 프로시저이다.

루틴은 (이 두 유형 모두) 파라미터를 여러 개 가질 수 있으며, 파라미터에는 타입이 지정된다. 나중에 살펴보겠지만, 프로시저와 함수는 클래스가 가지는 메서드 [method](#) 의 기반이기도 하다. 메서드에서도 이 두 개의 구분은 같다. 실제로 루틴 또는 메서드를 선언할 때 여러분은 키워드로 `procedure` 또는 `function` 사용해야 한다. 이는 C, C++, Java, C#, JavaScript 와 다른 점이다.

비록 키워드가 달라서 명확히 구별되지만, 활용할 때는 프로시저 `procedure` 와 함수 `function` 사이에 차이가 거의 없다. 예를 들어, 함수를 호출하여 작업 몇 개를 수행하고 나서, 함수가 반환한 결과를 (없어도 되는 오류 코드 등 이와 비슷한 것이라면) 무시할 수 있다. 또는 프로시저를 호출하고 파라미터 중 하나를 사용해 그 결과를 받아낼 수 있다 (참조 파라미터 `reference parameter` 에 대해서는 이 장의 뒷부분에서 자세히 설명한다).

다음은 오브젝트 파스칼 언어 구문 `syntax` 으로 작성된 프로시저의 정의다. `procedure` 키워드를 사용한다. (FunctionTest 예제에서 발췌함)

```
procedure Hello;
begin
    Show( '안녕하세요!' );
end;
```

똑같은 함수를 C 언어 구문 `syntax` 으로 작성하면 아래와 같다. 비교해 보자. 키워드가 없고 괄호가 있다(파라미터가 없는 경우에도 괄호는 필요함). 그리고 `void` 가 있다 (빈 반환 값 즉 결과가 없다는 표시임).

```
void Hello()
{
    Show( "안녕하세요!" );
};
```

실제로, C 언어 구문에는 프로시저와 함수 사이에 차이가 없다. 그러나, 파스칼 언어 구문에는 차이가 있다. 함수는 `function` 이라는 키워드를 사용해야 하고 반환 값(또는 반환 타입)이 있어야 한다.

참고 오브젝트 파스칼과 다른 언어가 구문 상 매우 명확하게 다른 점이 또 있다. 바로 정의 안에서 함수나 프로시저 서명 `signature` 이 끝나는 곳, 즉 `begin` 키워드 앞에 세미콜론이 붙는다는 점이다.

호출된 함수가 결과를 표현하는 방법은 두 가지다. 결과 값을 함수 이름에 대입하기 또는 `Result` 키워드에 대입하기 중에 하나를 선택하면 된다.

```
// 고전적인 파스칼 스타일
function DoubleOld(Value: Integer): Integer;
begin
    DoubleOld := Value * 2;
end;

// 현대적인 대안
function DoubleIt(Value: Integer): Integer;
begin
    Result := Value * 2;
end;
```

참고 고전적인 파스칼 언어 구문과 달리, 현대적인 오브젝트 파스칼에는 함수에서 결과를 표현하는 방법은 사실 세 가지다. `Exit` 메커니즘이 있기 때문이다. 그 내용은 이 장의 "결과를 가지고 빠져나가기 `Exit with a Result`" 부분에서 설명한다.

Result 를 사용하는 것이, 함수 이름을 사용하는 것 보다, 함수에서 반환 값을 할당할 때 가장 일반적으로 사용된다. 게다가 코드를 더 읽기 쉽게 해주는 경향이 있다. 함수 이름에 할당하는 방식은 고전적인 파스칼 표기법에서 쓰던 것인데, 이제는 사용되는 경우가 거의 없다. 하지만 여전히 지원된다.

이번에도, 똑같은 함수를 C 언어 구문으로 작성해보면 다음과 같다.

```
int DoubleIt(int Value)
{
    return Value * 2;
};
```

참고 return 문은 C 구문 기반 언어에서 함수의 결과를 표현할 뿐 아니라, 실행을 종료한다. 따라서, 통제권이 반환 [return](#) 되어 그 함수를 호출했던 위치로 되돌아 간다. 이와 달리, 오브젝트 파스칼에서는 Result에 그 함수의 결과 값을 할당한다고 해서 그 함수를 종료하지 않는다. 그래서, Result를 일반 변수로 사용하는 경우가 꽤 있다. 예를 들어 Result에 초기 기본값을 미리 할당하거나, 심지어 알고리즘 안에서 Result [결과](#)를 변경하기도 한다. 결과를 표현하면서 동시에 실행을 중지해야 하는 경우에는 Exit라는 흐름 제어 문장을 사용해야 한다. 이 모든 내용은 뒤에 나오는 "결과를 가지고 빠져나가기 [Exit with a Result](#) " 부분에서 설명한다.

이런 방식으로 루틴을 정의하고 나면, 정의에 비해 그것을 호출하는 구문은 명확하다. 식별자 [identifier](#) 를 적고 이어서 괄호 안에 파라미터들을 나열하면 된다. 파라미터가 없는 경우에는 빈 괄호를 적어도 되고 생략해도 된다 (C 구문 기반 언어와 다른 점이다). 이제부터 코드 조각을 살펴보자 (FunctionTest 프로젝트에서 발췌한 것들이다).

```
// 프로시저 호출
Hello;

// 함수 호출
X := DoubleIt(100);
Y := DoubleIt(X);
Show(Y.ToString);
```

여기에서 캡슐화 [encapsulation](#) 라는 코드 개념을 볼 수 있다. DoubleIt 함수를 호출할 때, 그 함수에 구현되어 있는 알고리즘을 알 필요가 없다. 만약, 시간이 지나고, 숫자를 두 배로 늘리는 더 좋은 방법을 찾았다면, DoubleIt 함수의 코드를 바꾸기도 쉽다. (실행 속도를 더 향상하더라도) 이 함수를 호출하는 코드는 그대로 유지된다.

Hello 프로시저에도 똑같은 원리가 적용된다. 프로그램 출력을 바꾸고 싶다면 Hello 프로시저의 코드만 바꾸면 된다. 그러면 그 효과는 기본 프로그램 코드에 자동으로 반영된다. Hello 프로시저의 구현 부분의 코드를 아래와 같이 변경할 수 있을 것이다.

```
procedure Hello;
begin
    Show( '안녕하세요, 다시 한번! ');
end;
```


포워드 선언 Forward Declarations

(어떤 종류이든) 식별자를 사용해야 할 때는, 이미 그 식별자를 컴파일러가 본 적이 있어야 한다. 그래야 그 식별자가 가리키는 것이 무엇인지 알 수 있다. 이런 이유로, 어떤 루틴이든 사용하기 전에 전체 정의를 먼저 제공하는 것이 일반적이다. 그런데, 그렇게 하는 것이 불가능한 경우도 있다. 프로시저 A 가 프로시저 B 를 호출하고, 프로시저 B 가 프로시저 A 를 호출하는 경우라면, 어느 한 프로시저는 컴파일러가 여전히 정의를 보지 못한 프로시저를 호출하는 코드를 가지게 될 것이다.

이 경우 (그리고 다른 많은 상황에서), 이름과 파라미터 만으로 프로시저 또는 함수의 존재를 (실제 코드가 없이) 선언할 수 있다. 이처럼 함수나 프로시저를 정의하지 않고 선언하려면 이름과 파라미터 (함수 서명 [signature](#) 이라고 함)를 적고 바로 뒤에 forward 키워드를 적는다.

```
procedure NewHello; forward;
```

코드에서 forward 로 선언된 프로시저는 나중에 전체 정의를 제공해야 한다 (같은 유닛 안에 있어야 함). 어쨌든 그 프로시저는 이제 전체 정의가 있는 위치보다 앞에서도 호출할 수 있다. 아래 예제를 보자. 아이디어를 얻을 수 있을 것이다.

```
procedure DoubleHello; forward;

procedure NewHello;
begin
  if MessageDlg('Do you want a double message?',
    TMsgDlgType.mtConfirmation,
    [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo], 0) = mrYes then
    DoubleHello
  else
    ShowMessage('Hello');
end;

procedure DoubleHello;
begin
  NewHello;
  NewHello;
end;
```

참고 위 코드 조각에서 호출되는 MessageDlg 함수는 사용자에게 확인을 요청할 때 사용하는 비교적 간단한 방법이다. 이 함수는 파이어몽키 프레임워크에 들어 있다 (VCL 프레임워크에도 비슷한 함수가 있다). 이 함수에게 전달할 파라미터는 표시하고 싶은 메시지, 대화 상자의 유형, 표시할 버튼들이다. 그리고 이 함수가 반환하는 결과는 사용자가 선택한 버튼의 식별자다.

위 방식을 통해 우리는 상호 재귀 [mutual recursion](#) 를 하는 코드를 작성할 수 있게 된다. DoubleHello 가 NewHello 를 호출하고, NewHello 도 DoubleHello 를 호출하는 것이 가능하다(역시 FunctionTest 예제에 있음). 사용자가 계속 Yes 버튼을 선택하는 한, 프로그램은 계속 메시지 상자를 표시한다. 그리고 Yes 버튼이 한 번 선택될 때마다 NewHello 는 두 번 실행된다. 재귀 [recursive](#) 코드 안에는 반드시 재귀를 종료하는 조건이

있어야 한다. 그래야 스택 오버플로 [stack overflow](#)라 불리는 상태가 되는 것을 피할 수 있다. 스택 오버플로의 주된 원인은 재귀 호출 [recursive call](#)에 의한 무한 루프 [infinite loop](#)다.

참고 함수 호출은 애플리케이션 메모리의 스택 부분을 사용한다. 거기에서 파라미터, 반환값, 로컬 변수 등을 다룬다. 만약 함수가 자기 자신을 계속 호출하는 끝없는 루프에 빠지면, 스택 [stack](#)용 메모리 영역(이 영역은 미리 정의된 고정 크기다. 링커 [linker](#)에 의해 정해지는데 프로젝트 옵션에서 설정할 수 있음)은 스택 오버플로라는 에러가 발생하고 결국 종료된다. 인기 있는 개발자 지원 사이트(www.stackoverflow.com)도 이 프로그래밍 에러에서 그 이름을 따왔다.

포워드 [forward](#) 프로시저 선언은 오브젝트 파스칼에서 그리 흔하지 않다. 하지만, 이와 비슷한 경우인데 훨씬 더 빈번하게 사용되는 것이 있다. 여러분이 프로시저나 함수 선언을 유닛의 `interface` 구역에 작성하면, 그것은 자동으로 포워드 [forward](#) 선언으로 간주된다. `forward` 키워드가 없는데도 말이다. 사실 여러분은 루틴의 본문을 유닛의 `interface` 구역 안에 넣지 못한다. 알아 둘 점이 있다. 루틴의 실제 구현은 그 루틴이 선언된 바로 그 유닛 안에서 반드시 제공해야 한다.

재귀 함수 A Recursive Function

재귀 [recursion](#)에 대해 언급을 했고 재귀의 다소 특이한 예(두 개의 프로시저가 서로를 호출하는 예)를 보았다. 이제 재귀 함수가 자기 자신을 호출하는 전형적인 예를 보자. 재귀를 사용하여 루프를 대체하는 경우가 종종 있다. 다음과 같다.

전형적인 데모는 숫자의 거듭제곱이다. 거듭제곱 [power](#) 함수가 없다고 가정하면 (물론 런타임 라이브러리에는 거듭제곱 함수가 들어있음), 2^3 은 2를 세 번 곱한 것($2*2*2$)과 같다는 수학 지식을 활용할 수 있다.

이것을 코드로 표현해보자. `for` 루프를 작성해 세 번 (지수 [exponent](#)의 값만큼) 반복해서 실행한다. 이때, 루프 안에서는 현재 결과를 2(밑 [base](#)의 값)에 곱한다. 루프 카운터는 1부터 시작하자.

```
function PowerL(Base, Exp: Integer): Integer;
var
  I: Integer;
begin
  Result := 1;
  for I := 1 to Exp do
    Result := Result * Base;
end;
```

또 다른 방법이 있다. 밑 [base](#)을 곱하는 연산을 반복하는데, 지수 [exponent](#)를 감소시키면서 0이 될 때까지 반복해서 진행하는 것이다. 이때, 지수가 0이면 결과는 언제나 1이 되도록 한다. 이 방식은, 같은 함수를 반복해서 재귀 호출하는 코드로 작성할 수 있다.

```
function PowerR(Base, Exp: Integer): Integer;
var
  I: Integer;
```



```
begin
  if Exp = 0 then
    Result := 1
  else
    Result := Base * PowerR(Base, Exp - 1);
end;
```

재귀 버전으로 된 프로그램은 for 루프 기반 버전보다 빠르지 않을 가능성이 높다. 읽기도 더 어렵다. 그런데, 이런 경우들이 있다. 코드 구조(예: 트리 구조)를 분석하는데 *parcing* (처리해야 할) 요소의 수를 미리 정할 수 없는 구조라면 루프를 적용하는 것은 거의 불가능하다. 하지만, 재귀 함수는 그런 상황에서도 잘 맞춘다.

일반적으로 재귀 코드는 강력하다. 하지만 더 복잡한 경향이 있다. 지난 수년 간 재귀는 거의 잊혀져 가고 있었다. 프로그래밍 초창기에 비하면 말이다. 그런데, 지금은, Haskell, Erlang, Elixir 등 새로운 함수형 언어들이 재귀를 무척 많이 사용한다. 그리고 그 아이디어가 다시 인기를 얻도록 이끌고 있다.

어쨌든, 위 거듭제곱 함수 두 가지는 FunctionTest 예제에 들어 있다.

참고 데모에 있는 위 두 거듭제곱 함수는 지수가 음수인 경우를 다루지 못한다. 위 재귀 버전에서 음수를 지수에 넣으면, 영원히 (더 정확하게는 프로그램이 물리적 한계에 도달할 때까지) 반복 된다. 또한 정수를 사용하므로 데이터 타입의 최대 크기에 비교적 빠르게 도달하여 넘쳐흐르게 *overflow* 된다. 이런 한계가 있지만 여러분을 위해 코드를 단순하게 유지하려고 그렇게 작성했다.

메서드란 무엇인가? What Is a Method?

포워드 *forward* 선언을 작성하는 방법에는 유닛의 인터페이스 구역 안에 선언하기와 *forward* 키워드 사용하기가 있다는 것을 앞에서 살펴보았다. 그런데, 클래스 타입 안에 선언된 메서드 역시 포워드 선언으로 간주된다.

메서드 *Method* 란 정확히 무엇일까? 메서드란 함수 또는 프로시저의 특별한 유형으로서, 데이터 타입 두 가지, 즉 레코드 *record* 와 클래스 *class* 중 하나,에 연결되어 있는 것이다. 오브젝트 파스칼에서, 우리는 시각적 컴포넌트의 이벤트를 처리할 때 마다 메서드 *Method* 를 정의하게 된다. 대체로 프로시저 *procedure* 로 정의한다. 하지만, 메서드라는 용어는 함수와 프로시저 모두를 지칭한다. 클래스나 레코드에 연결된 것이라면 말이다.

폼 *form* 의 소스 코드에 자동으로 추가되는 빈 메서드를 하나 보자. (폼은 사실 클래스다. 이 책의 뒷부분에서 이 점에 대해 자세히 살펴본다).

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // 여러분의 코드를 여기에 넣으세요
end;
```


파라미터와 반환값Parameters and Return Value

함수 또는 프로시저를 호출할 때는, 그것이 기대하는 파라미터의 타입과 개수가 정확히 일치하는지 확인해야 한다. 그렇지 않으면 컴파일러는 타입 불일치 에러 메시지를 표시한다. 이는 변수에 할당하려는 값의 타입이 잘못되었을 때와 같은 메시지이다. 앞에서 본 DoubleIt 함수가 정수 Integer 파라미터를 받도록 정의되었다고 가정하고, 아래와 같이 호출해보자.

```
DoubleIt(10.0);
```

컴파일러는 아래와 같은 에러를 표시한다.

```
[dcc32 Error] E2010 Incompatible types: 'Integer' and 'Extended'
```

팁 델파이 에디터는 개발자가 함수(또는 프로시저) 이름 뒤에 여는 괄호를 타이핑하면 즉시 그것의 파라미터 목록을 힌트로 제공한다. 코드 파라미터Code Parameters 라는 기능인데 코드 인사이트Code Insight 기술(다른 IDE에서는 IntelliSense라고 함)의 일부다. 델파이의 코드 인사이트는 10.4 버전부터 LSP(언어 서버 프로토콜) 서버를 기반으로 작동한다.

제한적이지만 타입 변환 type conversion 이 허용되는 상황이 있다. 이는 할당을 시도할 때와 마찬가지다. 하지만, 지정된 타입과 일치하는 파라미터를 사용하도록 노력하는 것이 대체로 좋다 (참조 파라미터 reference parameter 인 경우에는 반드시 그래야 한다. 잠시 후에 살펴보겠다).

함수를 호출할 때, 파라미터로 값 대신 표현식을 전달할 수 있다. 그러면 그 표현식을 평가한 결과가 파라미터에 할당된다. 더 간단하게는, 변수 이름만 전달한다. 그러면 그 변수의 값이 복사되어 파라미터(대체로 변수의 이름과 파라미터의 이름은 서로 다름)에 들어간다. 파라미터와 그 파라미터에게 전달되는 변수가 절대로 같은 이름을 사용하지 않도록 하길 바란다. 그래야 혼란스럽지 않을 것이다.

경고 델파이에서는, 함수에 전달되는 파라미터의 평가evaluation 순서를 의지하면 안 된다. 평가 순서는 호출 규약calling convention에 따라 다르다. 그리고 정의되지 않은 경우도 있기 때문이다. 가장 흔한 경우는 오른쪽에서 왼쪽으로 평가하기다. 이에 대해 자세히 설명한 글은 아래와 같다:
[https://docwiki.embarcadero.com/RADStudio/en/Procedures_and_Functions_\(Delphi\)#Calling_Conventions](https://docwiki.embarcadero.com/RADStudio/en/Procedures_and_Functions_(Delphi)#Calling_Conventions)

마지막으로 알아 둘 점이 있다. 함수나 프로시저 하나를 다양한 버전으로 만들 수 있다 (오버로딩 overloading 이라고 부르는 특징이다). 그리고 여러분이 파라미터를 생략하고 호출할 수 있도록 미리 정의해 놓을 수도 있다 (기본 파라미터 default parameter 라고 한다). 이 두 핵심 특징들은 이 장의 뒷부분에서 자세히 설명한다.

결과를 가지고 빠져나가기 Exit with a Result

함수가 결과를 반환할 때 C 계열 언어와 상당히 다른 구문 [syntax](#) 을 사용한다는 점을 앞에서 보았다. 구문만 다른 것이 아니라 동작 [behavior](#) 도 다르다. 즉 Result(또는 함수 이름)에 값을 할당해도 그 함수를 종료하지 않는다. 이와 달리, C 계열에서 return 문은 그 함수를 종료한다. 오브젝트 파스칼 개발자는 종종 이 기능을 활용하여 Result 를 임시 저장소로 사용한다. 임시 저장소로 활용하지 않는 코드를 먼저 보자.

```
function ComputeValue: Integer;
var
  Value: Integer;
begin
  Value := 0;
  while ...
    Inc(Value);
  Result := Value;
end;
```

위 코드는 임시 변수를 사용하고 있다. 이것을 생략하고 대신 Result 를 쓸 수 있다. Result 안의 값은 그 함수가 종료 [terminate](#) 되면 반환 [return](#) 된다. 값이 무엇이든 말이다.

```
function ComputeValue: Integer;
begin
  Result := 0;
  while ...
    Inc(Result);
end;
```

한편, 반환 값을 할당하고 나서 바로 빠져나가고 싶은 상황도 있을 것이다. 예를 들면, if 분기에서 바로 빠져나가야 할 수 있다. 함수 안에서 결과를 할당한 후에 더 이상 그 함수를 실행하지 않고 빠져나오고 싶다면, Result 키워드를 담는 문장과 Exit 키워드를 담는 문장을 각각 따로 사용해야 한다.

앞 장(중 "Break 와 Continue 로 흐름 끊기" 부분)에서 본 FlowTest 예제의 코드가 기억난다면, 그 코드를 함수로 바꾸고 break 호출을 Exit 호출로 바꿔보자. 그렇게 바꾼 코드는 아래와 같다 (ParamsTest 예제에서 발췌함).

```
function CharInString(S: string; Ch: Char): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low(S) to High(S) do
    if S[I] = Ch then
      begin
        Result := True;
        Exit;
      end;
end;
```


위 코드에서 if 블록 안에 있는 문장은 두 개이다. 이제 아래와 같이 함수의 반환값을 가지고 나가는 Exit 문으로 바꿔보자. 이는 C 언어의 return 문과 비슷한 방식이다. 그리고 코드가 더 간결하다 (문장이 하나뿐이므로 begin-end 블록도 필요 없다).

```
function CharInString2(S: string; Ch: Char): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low(S) to High(S) do
    if S[I] = Ch then
      Exit(True);
end;
```

참고 오브젝트 파스칼에서 Exit는 함수다. 따라서 반환하려는 값을 반드시 괄호 안에 넣어야 한다. 이와 달리, C 스타일 언어에 있는 return은 컴파일러 키워드이며 괄호가 필요 없다.

참조 파라미터 Reference Parameters

오브젝트 파스칼에서, 프로시저와 함수는 파라미터를 값으로 *by value* 전달하거나 참조 *by reference* 로 전달할 수 있다. 기본 *default* 은 값으로 전달하는 것이다. 이 경우, 전달된 값은 스택 *stack* 에 복사된다. 따라서 루틴이 사용하고 조작하는 것은 데이터의 복사본이다. 원본에 있는 값이 아니다 ("파라미터와 반환값 [Parameters and Return Value](#) 부분에서 설명했음). 참조로 전달한다는 의미는 스택(상에 그 함수의 공식 파라미터들이 들어 있는 곳)에 원본의 값을 복사하지 않는다는 것이다. 프로그램은 복사 대신 그 원본의 값 자체를 참조한다 (그 함수의 코드 안에서 수행된다는 점은 마찬가지임). 따라서, 파라미터로 전달된 변수의 실제 원본의 값을 프로시저나 함수 안에서 변경할 수 있게 된다. 파라미터를 참조로 전달할 때는 **var** 키워드를 사용해 표시한다.

이 기술은 대부분의 프로그래밍 언어들이 사용한다. 복사 과정이 없어서 프로그램이 더 빠르게 실행되는 경우가 많기 때문이다. C에는 없지만(C에서는 포인터를 쓰면 됨), C++와 C 구문 기반 기타 언어들은 &(참조로 전달) 기호 *symbol* 를 사용해 쓰고 있다. 아래 예문은 **var** 키워드를 사용하고 있다. 파라미터를 참조로 전달하기 때문이다.

```
procedure DoubleIt(var Value: Integer);
begin
  Value := Value * 2;
end;
```

위 예문에 있는 파라미터는 두 가지 역할을 한다. 즉 DoubleIt 프로시저에게 값을 전달한다. 그리고, 그 프로시저에서 그 값이 변경되면 그 결과값을 다시 호출자에게 되돌려준다. 아래 예문은 호출자의 코드다.

```
var
  X: Integer;
begin
```



```
X := 10;
DoubleIt(X);
Show(X.ToString);
```

위 코드에서 변수 `x` 의 값은 20 이 된다. 왜냐하면, `DoubleIt` 프로시저는 `x` 가 원래 위치한 메모리 주소를 참조하기 때문에 당연히 원본 값에 영향을 준다.

일반 파라미터를 전달하는 규칙에 비해서, 참조 파라미터에 값을 전달할 때는 보다 엄격한 규칙을 따라야 한다. 값이 아니라 실제 변수가 전달되기 때문이다. 첫째, 참조 파라미터에 상수 `constant` 값을 넣어서 전달할 수 없다. 둘째, 타입이 조금만 달라도 (즉, 자동 변환이 필요하도록) 변수를 전달할 수 없다. 즉, 변수와 파라미터는 서로 타입이 정확히 일치해야 한다. 그렇지 않으면 아래와 같은 컴파일러 에러 메시지를 받는다.

```
[dcc32 Error] E2033 Types of actual and formal var parameters must be identical
```

예를 들어, 위 에러 메시지를 발생시킨 코드는 아래와 같다 (ParamsTest 예제에 있음. 단 주석으로 처리되어 있음)

```
var
  C: Cardinal;
begin
  C := 10;
  DoubleIt(C);
```

파라미터를 참조로 전달하기에 적합한 것은 순서 `ordinal` 타입과 레코드 `record` 타입이다 (다음 장에서 살펴보겠다). 이 타입들은 값 타입이라고 불린다. 그 이유는 의미 상, 값으로 전달 `pass-by-value` 되고 값으로 할당 `assign-by-value` 되기 때문이다.

오브젝트 파스칼에서, 오브젝트 `object` 나 문자열 `string` 인 경우에는 동작이 조금 다르다 (여기에 대해서는 나중에 더 자세히 살펴보겠다). 오브젝트를 담는 변수는 실제로 참조 `reference` 이다. 따라서 오브젝트를 파라미터로 받으면 그 오브젝트의 실제 데이터를 변경할 수도 있다. 이런 종류의 타입은 값 타입과는 다른 부류이며, 참조 타입이라고 불린다.

오브젝트 파스칼에는 파라미터의 유형으로 표준 그리고 참조(`var`) 외에, 매우 독특한 파라미터 지정자 `specifier` 인 `out` 이 있다. `out` 파라미터는 값을 반환할 목적으로 사용될 뿐, 값을 받지 않는다. 이런 점만 빼면, `out` 파라미터는 `var` 파라미터와 같다.

참고 `out` 파라미터는 윈도우 `Windows`의 컴포넌트 객체 모델(즉, COM)에 있는 해당 개념을 지원하기 위해서 도입되었다. 값을 전달받고 싶지 않다는 개발자의 의도를 표현하는 데 사용할 수 있다.

상수 파라미터 Constant Parameters

참조 파라미터 [reference parameter](#) 들의 대안으로 여러분은 `const` 상수 파라미터를 쓸 수 있다. 루틴 안에서는 상수 `const` 파라미터에 새 값을 할당할 수 없다. 따라서 컴파일러는 파라미터 전달을 최적화할 수 있다. 컴파일러가 접근하는 방식은 참조 파라미터(또는 C++ 용어로는 `const reference` 임)에 가깝다. 하지만, 함수 안에서 파라미터의 원본 값을 변경하지 못한다는 점은 값 `value` 파라미터에 가깝다고 볼 수 있다.

실제로, 다음 코드를 컴파일하려고 시도하면(ParamsTest 예제에 있음. 단 주석으로 처리되어 있음), 시스템에서 에러를 일으킨다.

```
function DoubleIt(const Value: Integer): Integer;
begin
    Value := Value * 2; // 컴파일러 에러
    Result := Value;
end;
```

표시되는 에러 메시지는 다음과 같다. 즉시 이해하기에는 직관적이지 않을 수 있다.

[dcc32 Error] E2064 Left side cannot be assigned to

상수 `constant` 파라미터가 매우 흔하게 쓰이는 경우는 문자열을 전달할 때다. 컴파일러가 참조 카운팅 메커니즘을 비활성화하므로 최적화 효과를 조금 얻을 수 있기 때문이다. 상수 파라미터를 사용하는 가장 흔한 이유는 바로 최적화다. 그런 면에서, 이 기능은 순서 `ordinal` 타입과 스칼라 `scalar` 타입에서 별 의미가 없다. 상수 파라미터는 오브젝트를 전달할 때도 별로 쓰이지 않는다. 그 이유를 보자. 오브젝트 파스칼에서, 오브젝트를 상수 파라미터로 전달하는 경우, 변경하지 못하는 것은 그 오브젝트를 가리키는 참조다. 그 오브젝트 자체가 아니다. 즉, 컴파일러는 여러분이 거기에 다른 새 오브젝트를 할당하지 못하도록 지켜주지만, 그 원본 오브젝트의 메서드 호출을 막지는 않는다. 그 메서드가 그 오브젝트의 실제 데이터를 변경하는 메서드라 할지라도 말이다.

참고 잘 알려지지 않았지만, `const` 파라미터를 전달하는 또 다른 방식이 있다. `ref` 애트리뷰트 [attribute](#) 를 덧붙이는 것이다. "`const [ref]`"라고 쓴다. 그러면, 그 상수 파라미터를 참조로 전달하라고 컴파일러에게 강제한다. 기본 설정에서는 값으로 전달과 참조로 전달 중 하나를 컴파일러가 알아서 선택한다. 그 기준은 파라미터의 크기이며, 판단 결과는 대상 CPU와 플랫폼에 따라 달라진다.

함수 오버로딩 Function Overloading

때로는, 매우 비슷한 함수인데 파라미터와 내부 구현이 다른 것들이 필요할 수 있다. 과거에는 이름을 살짝 바꿔서 각 함수에 붙여야 했지만, 현대 프로그래밍 언어에서는 심볼 [symbol](#) 하나를 오버로드 [overload](#) 하여 여러 정의 [definition](#) 를 가질 수 있도록 한다.

오버로딩 [overloading](#)이라는 아이디어는 간단하다. 함수 또는 프로시저 두 개 이상을 같은 이름으로 정의할 수 있게 컴파일러가 허용하는 것이다. 단, 파라미터가 달라야 한다. 컴파일러는 파라미터를 확인해서 함수의 어느 버전이 호출되었는지를 결정한다.

아래에 나열된 함수는 런타임 라이브러리(RTL)의 System.Math에서 발췌한 것이다.

```
function Min(A,B: Integer): Integer; overload;
function Min(A,B: Int64): Int64; overload;
function Min(A,B: Single): Single; overload;
function Min(A,B: Double): Double; overload;
function Min(A,B: Extended): Extended; overload;
```

Min(10, 20)을 호출하면, 컴파일러는 Min 함수 중 맨 위에 있는 버전이 호출되도록 한다. 왜냐하면 그 버전이 정수 두 개를 받아서 정수를 반환하기 때문이다.

오버로딩에는 기본 규칙 두 가지가 있다.

- 오버로드되는 함수(또는 프로시저)의 각 버전마다 반드시 **overload** 키워드를 뒤에 붙여야 한다(첫 번째 버전에도 붙여야 한다).
- 오버로드되는 함수끼리는 반드시 파라미터의 개수 또는/그리고 타입이 달라야 한다. 파라미터의 이름은 고려되지 않는다. 왜냐하면 파라미터 이름을 호출하는 코드에 적지 않기 때문이다. 반환 타입 [return type](#) 역시 함수를 구분하는데 사용되지 못한다.

참고 반환값으로 함수를 구분할 수 없다는 규칙에는 예외가 있다. Implicit [암시적](#) 그리고 Explicit [명시적](#) 변환 연산자 [conversion operator](#)에 관한 것이므로 5장에서 다룬다.

다음은 오버로드 버전이 세 개인 ShowMsg 프로시저다. (OverloadTest 예제에서 발췌함. 기본 파라미터 [default parameter](#) 그리고 오버로드 [overload](#) 를 시연하는 예제임).

```
procedure ShowMsg(Str: string); overload;
begin
  Show( '메시지: ' + Str);
end;

procedure ShowMsg(FormatStr: string;
  Params: array of const); overload;
begin
  Show( '메시지: ' + Format(FormatStr, Params));
end;

procedure ShowMsg(I: Integer; Str: string); overload;
begin
  ShowMsg(I.ToString + ' ' + Str);
end;
```

위 세 함수는 문자열 [string](#)을 표현한다. 그런데 각자 다른 방식으로 문자열 형식 [format](#)을 지정한다. 이 프로시저들을 호출하는 코드는 아래와 같다.

```
ShowMsg( '안녕하세요 ');
ShowMsg( '합계 = %d. ', [100]);
ShowMsg(10, 'M 바이트');
```


결과는 다음과 같다.

```
메시지: 안녕하세요
메시지: 합계 = 100.
메시지: 10 M 바이트
```

팁 IDE 통합개발환경의 코드 파라미터 기술은 오버로드된 프로시저와 함수에서 매우 훌륭하게 작동한다. 루틴 이름 뒤에 여는 괄호를 타이핑하면 사용 가능한 모든 대안이 나열된다. 파라미터를 넣으면 코드 인사이트 기술이 해당 타입을 인식하고 유효한 대안이 어느 것인지를 결정한다.

함수를 호출하려고 전달한 파라미터가 오버로드된 함수 버전들 중 무엇과도 일치하지 않으면 어떻게 될까? 에러 메시지가 표시된다. 아래와 같이 호출한다고 가정해보자.

```
ShowMsg(10.0, '안녕하세요');
```

이 경우에 표시되는 에러 메시지는 아래와 같이 매우 구체적이다.

```
[dcc32 Error] E2250 There is no overloaded version of 'ShowMsg' that
can be called with these arguments
```

(옮긴이: E2250 오버로드된 'ShowMsg' 버전 중 이 인수들과 함께 호출할 수 있는 것이 없음)

오버로드된 루틴은 모두 `overload` 키워드가 붙어야 한다는 사실은 중요하다. 같은 유닛 안에 있는 루틴 중에 `overload` 키워드가 붙어 있지 않은 것은 오버로드 할 수 없다는 의미이기 때문이다.

그런 시도를 하면 아래와 같은 에러 메시지를 받게 된다.

```
Previous declaration of '<name>' was not marked with the 'overload'
directive.
```

(옮긴이: 이전에 선언한 '<name>'에는 'overload' 지시어가 표시되어 있지 않음)

하지만, 이름이 같은 루틴을 만드는 것도 가능하다. 단, 유닛이 달라야 한다. 이는 유닛이 네임스페이스 `namespace/이름 영역` 역할을 하기 때문이다. 이 경우 여러분은 같은 함수를 오버로드해 새 버전을 만들고 있지 않다. 새 함수를 만들어 교체하고 원래 함수를 숨기고 `hide` 있다 (유닛 이름을 접두사로 붙이면 참조할 수 있다). 따라서 컴파일러는 파라미터를 기반으로 알맞은 버전을 고르지 않는다. 전달받은 파라미터를 지금 보고 있는 그 버전의 파라미터 타입들에게 맞추려고 한다. 타입이 일치하지 않으면 에러를 발생시킨다.

오버로드와 모호한 호출 `Overloading and Ambiguous Calls`

오버로드된 함수를 호출하면, 일반적으로, 컴파일러는 일치하는 버전을 찾아 올바르게 작동한다. 파라미터가 일치하는 버전이 없다면 에러를 발생시킨다 (방금 살펴보았다).

그런데, 또 다른 경우가 있다: 컴파일러는 함수의 파라미터를 위해 몇 가지 타입 변환을 할 수 있다. 그런데, 여러 가지 변환이 호출 하나에 대해 가능할 수도 있다. 컴파일러는 호출 가능한 함수 버전을 여러 개 찾은 경우, “완벽하게” 일치하는 것이 (있다면 그것을 고르지만) 없다면 함수 호출이 모호하다는 에러 메시지를 표시한다.

아래 예제는 논리적이진 않다. 흔히 만나는 상황도 아니다. 하지만, 생각해 볼 가치가 있다(현장에서 가끔 발생하는 사례다).

정수와 부동 소수점 숫자를 더하는 함수를 두 가지 오버로드된 버전으로 구현했다고 가정해보자.

```
function Add(N: Integer; S: Single): Single; overload;
begin
    Result := N + S;
end;

function Add(S: Single; N: Integer): Single; overload;
begin
    Result := N + S;
end;
```

이제 파라미터의 순서와 상관없이 이 함수를 호출할 수 있다 (OverloadTest 예제에서 발췌함).

```
Show(Add(10, 10.0).ToString);
Show(Add(10.0, 10).ToString);
```

일반적으로 함수는 변환이 가능한 파라미터라면 받아준다. 따라서 파라미터 타입으로 부동 소수점 타입을 기대하고 있는 곳에 정수를 전달하는 것이 가능하다. 그렇다면 다음과 같이 호출하면 어떻게 될까?

```
Show(Add(10, 10).ToString);
```

컴파일러는 오버로드된 함수의 첫 번째 버전을 호출할 수 있다. 하지만, 두 번째 버전 역시 호출할 수 있다. 즉, 여러분이 무엇을 요청하는지 모른다 (이 두 버전 중 어느 것을 호출해도 결과가 같다는 것도 모른다). 그래서 에러 메시지를 표시한다.

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'Add'
Related method: function Add(Integer; Single): Single;
Related method: function Add(Single; Integer): Single;
```

팁 IDE의 에러 창에 표시되는 에러 메시지는 위의 첫 번째 줄이다. 옆에는 더하기(펼치기) 기호가 나타난다. 이 기호를 누르면 상세 내용이 펼쳐진다. 위 경우에는 컴파일러가 어떤 것들 때문에 모호하다고 하는 지를 알려준다.

만약, 실제 상황이 위와 같다면, 파라미터의 타입을 직접 변환해 호출하면 해결된다. 즉, 호출하고 싶은 버전을 아래와 같이 명확하게 컴파일러에게 알려주면 된다.

```
Show(Add(10, 10.ToSingle).ToString);
```


모호한 호출은 variant [변이](#) 타입을 사용할 때도 쉽게 발생한다. variant 는 좀 특이한 타입이므로 이 책의 뒷부분에서만 다루겠다.

기본 파라미터 Default Parameters

오버로딩과 관련된 또 다른 특징을 보자. 함수의 파라미터 중 일부에 기본값 [default value](#) 을 지정할 수 있다. 그러면, 함수를 호출할 때, 여러분은 기본값이 지정된 파라미터에 값을 전달해도 되고 생략해도 된다. 생략되면 미리 지정된 기본값이 사용된다.

예제를 보자(역시 OverloadTest 예제에서 발췌함). Show 호출을 캡슐화 [encapsulation](#) 하는 프로시저를 정의하고 그 파라미터 중 두 개에 기본 파라미터 [Default Parameter](#) 를 제공했다.

```
procedure NewMessage(Msg: string; Caption: string = '메시지';
    Separator: string = ': ');
begin
    Show(Caption + Separator + Msg);
end;
```

위에서 정의한 프로시저를 호출하는 방법은 이제 여러 가지다. 아래와 같다.

```
NewMessage( '지금 뭔가 잘못되었습니다! ');
NewMessage( '지금 뭔가 잘못되었습니다! ', '주의' );
NewMessage( '안녕하세요 ', '메시지', '-- ');
```

출력된 결과는 다음과 같다.

```
메시지: 지금 뭔가 잘못되었습니다!
주의: 지금 뭔가 잘못되었습니다!
메시지--안녕하세요
```

알아 두어야 할 점이 있다. 기본 파라미터를 지원하기 위해 컴파일러가 만들어 내는 특별한 코드는 전혀 없다. 또한 그 함수나 파라미터에 대한 (오버로드된) 복사본 여러 개를 만드는 것도 아니다. 컴파일러는 그저 호출 코드안에 전달되지 않은 파라미터가 있으면 그 자리에 기본값을 넣어줄 뿐이다. 기본 파라미터를 사용할 때 주의해야 할 중요한 한 가지 제한 사항이 있다. 파라미터를 "건너뛰는" 것은 허용되지 않는다. 예를 들어, 두 번째 파라미터를 생략하고 세 번째 파라미터를 전달할 수 없다.

기본 파라미터가 있는 함수와 프로시저(와 메서드)를 정의하고 호출하는 규칙은 몇 가지가 있다.

- 호출 안에서, 파라미터 생략은 마지막 파라미터부터 시작해야 한다. 즉, 파라미터 하나를 생략하면 그 뒤에 나열된 파라미터들까지 모두 생략해야 한다.
- 정의 안에서, 기본값을 지정하는 파라미터들은 반드시 나열된 파라미터 중 뒤쪽 끝에 있는 것들이어야 한다.

- 기본값은 상수여야 한다. 따라서, 기본 파라미터에 사용할 수 있는 타입에 제한이 있다. 예를 들어, 동적 배열 `dynamic array` 타입 또는 인터페이스 `interface` 타입인 경우에는 기본 파라미터에 오직 `nil`만 넣을 수 있다. 레코드 `record` 타입인 경우에는 기본 파라미터로 넣을 수 있는 것이 전혀 없다.
- 기본값이 있는 파라미터는 값으로 `by value` 전달하거나 `const`로 전달해야 한다. 참조 (`var`) 파라미터는 기본값을 가질 수 없다.

기본 파라미터와 오버로드를 동시에 사용하면 컴파일러를 혼동시키는 상황이 발생할 가능성이 더 높다. 즉 컴파일러가 모호한 호출 `ambiguous call` 에러를 일으키게 되는 상황(앞에서 설명했음)이 되기 쉽다. 예를 들어, 위에 정의한 `NewMessage` 프로시저에 새 버전을 추가한다고 가정해보자.

```
procedure NewMessage(Str: string; I: Integer = 0); overload;
begin
    Show(Str + ': ' + IntToStr(I))
end;
```

이 정의는 합법적이므로, 컴파일러가 받아준다. 하지만, 아래와 같이 호출해보자.

```
NewMessage( '안녕하세요' );
```

컴파일러는 아래와 같은 에러 메시지를 표시한다:

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'NewMessage'
    Related method: procedure NewMessage(string; string; string);
    Related method: procedure NewMessage(string; Integer);
```

이 에러가 발생하는 위치를 잘 보자. 조금 전 `NewMessage` 함수에 새 오버로드 버전을 추가하기 전까지 문제없이 컴파일이 되던 곳이다. 그런데, 파라미터로 문자열 하나를 전달하는 호출을 컴파일러가 처리하려고 하면, 문자열 세 개가 파라미터로 정의된 버전과 문자열 하나와 정수 하나가 파라미터로 정의된 새로 추가된 버전 중 개발자가 원하는 것이 무엇인지 알 수 없다. 즉, 컴파일러는 명확히 알지 못하기 때문에 작동을 중지하고 더 명확하게 의도를 표현해 달라고 프로그래머에게 말하게 된다.

인라인 처리 `Inlining`

인라인 처리 `Inlining`, 즉 오브젝트 파스칼 함수와 메서드를 줄 안에 넣는 것은 이 언어의 저-수준 기능인데, 상당한 최적화 효과를 낼 수 있다. 일반적으로, 메서드를 호출하면, 컴파일러는 코드 몇 개를 생성해서 그 프로그램이 새 실행 지점으로 건너뛸 수 있게 한다. 즉, 스택 프레임 `stack frame` 설정과 몇 가지 연산을 수행한다. 또한 기계 명령 `machine instruction` 십여 개가 필요할 수도 있다. 그런데, 실행하려 하는 메서드 자체가 매우 짧은 경우가 있다. 심지어 프라이빗 필드 `private(전용) field` 를 읽기 또는 쓰기만 하는 간단한

것일 수도 있다. 이런 경우라면, 실제 코드를 복사해 호출 지점 안에 넣는 것이 메서드를 호출하는 것 보다 훨씬 더 합리적이다. 스택 프레임 설정과 기타 모든 명령을 수행하지 않기 때문이다. 이렇게 추가 부담 [overhead](#) 을 없애면, 프로그램이 더 빠르게 실행된다. 특히 수천 번 실행되는 뻘뻘한 루프 안에 들어 있는 호출이라면 그 효과는 더 클 것이다.

아주 작은 함수라면, 결과 코드의 크기가 더 작아질 수도 있다. 호출 지점에 그냥 끼워 들어가는 코드가 그 호출을 준비하느라 추가되는 코드보다 더 작을 수도 있기 때문이다. 하지만, 유의해야 할 점이 있다. 상당히 긴 함수를 인라인 [inline](#) 으로 구현하고, 게다가 그 함수를 많은 곳에서 호출한다면, 코드 부풀림 [code bloat](#) 현상을 겪을 수 있다. 즉 실행 파일의 크기가 불필요하게 커질 수 있다.

오브젝트 파스칼에서, 컴파일러에게 함수(또는 메서드)를 인라인 처리하도록 요청하려면 함수(또는 메서드) 선언 뒤에 `inline` 지시어를 붙이면 된다. 이 지시어를 정의에 중복해 붙일 필요는 없다. 명심할 점이 있다. 컴파일러에게는 `inline` 지시어가 힌트일 뿐이다. 함수가 인라인에 적합한 후보일지 아닐지는 컴파일러가 판단한다. 적합하지 않으면 (아무 경고 없이) 그 요청을 무시하는 것 역시 전적으로 컴파일러에게 달려있다. 또한, 컴파일러는 그 함수 호출 중 일부만 인라인으로 처리할 수도 있다. 모두를 그렇게 해야 하는 건 아니다. 이는 호출하는 코드를 분석한 후에 그리고 그곳의 `$INLINE` 지시어의 상태에 따라 결정된다. 그 지시어에는 세 가지 값이 있을 수 있다 (알아 둘 점: 이 기능은 컴파일러의 최적화 스위치와는 관계없이 반영된다).

- `{$INLINE OFF}`인 경우, 호출 대상 함수 선언에 `inline` 지시어가 붙어있든 아니든 상관없이 인라인 처리를 억제할 수 있다. 이것은 프로그램 전체, 프로그램의 일부, 특정 호출 위치 수준에서 지정할 수 있다.
- 기본 [default](#) 값인 `{$INLINE ON}`인 경우, 함수 선언에 `inline` 지시어가 붙은 함수에 대해 인라인 처리가 활성화된다.
- `{$INLINE AUTO}`인 경우, 컴파일러는 함수 선언에 `inline` 지시어가 붙은 함수를 대체로 인라인으로 처리한다. 또한 함수가 매우 짧으면 자동으로 그렇게 한다. 단, 코드 부풀림을 발생시킬 수 있으므로 주의해야 한다.

오브젝트 파스칼 런타임 라이브러리 안의 `inline` 지시어가 붙은 많은 함수들이 많다. 즉, 인라인 대상이 될 수 있는 함수들이 많다. 예를 들어, `System.Math` 유닛에 있는 `Max` 함수는 다음과 같은 정의되어 있다.

```
function Max(const A, B: Integer): Integer; overload; inline;
```

위 함수가 인라인 처리되면 효과가 어떨지 실제로 테스트해보자. 예문은 다음과 같다 (InliningTest 예제에서 발췌함).

```
var
  Sw: TStopWatch;
  I, J: Integer;
```



```

begin
  J := 0;
  Sw := TStopWatch.StartNew;
  for I := 0 to LoopCount do
    J := Max(I, J);
  Sw.Stop;
  Show( 'Max ' + J.ToString +
        ' [' + Sw.ElapsedMilliseconds.ToString + ']' );

```

위 코드는 System.Diagnostics 유닛에 있는 레코드인 TStopWatch 를 쓴다. 이 구조체는 Start(또는 StartNew)와 Stop 호출 사이에 경과한 시간(즉 시스템 틱 [system tick](#))을 추적한다.

폼 [form](#)에는 버튼이 두 개 있다. 둘 다 똑같은 코드를 호출한다. 다만, 하나는 호출하는 곳에서 인라인 처리를 비활성화했다. 둘 사이의 차이를 보려면 릴리스 구성 [Release configuration](#) 으로 설정하고 빌드해야 한다(인라인 처리는 릴리스를 위한 최적화 기능임). 내 컴퓨터에서 2 천만 번(LoopCount 상수의 값)을 반복 [iteration](#) 한 결과는 다음과 같다.

```

// 윈도우에서 (가상머신에서 실행)
Max on 20000000 [17]
Max off 20000000 [45]

// 안드로이드에서 (장비에서 실행)
Max on 20000000 [280]
Max off 20000000 [376]

```

위 데이터를 보자. 윈도우 [Windows](#) 에서 인라인을 사용하면 프로그램 실행 속도가 두 배 이상 빨라지는 반면, 안드로이드 [Android](#) 에서는 35% 정도 빨라진다. 그러나 안드로이드 장비에서 프로그램 실행 속도가 훨씬 느리기 때문에 단축된 시간만 보면, 윈도우에서 30 밀리초, 안드로이드 장비에서는 100 밀리초를 이 최적화를 통해 줄일 수 있었다.

이 예시는 Length 함수로도 비슷한 테스트를 한다. Length 함수는 특별히 인라인으로 처리되도록 바뀐 컴파일러의 마법 함수다. 역시 인라인 버전이 윈도우와 안드로이드 모두에서 훨씬 더 빠르다.

```

// 윈도우에서 (가상머신에서 실행)
Length inlined 260000013 [11]
Length not inlined 260000013 [40]

// 안드로이드에서 (장비에서 실행)
Length inlined 260000013 [401]
Length not inlined 260000013 [474]

```

위 결과를 만드는 테스트 루프의 코드는 아래와 같다.

```

var
  Sw: TStopWatch;
  I, J: Integer;
  Sample: string;
begin
  J := 0;
  Sample := 'sample string';

```



```

Sw := TStopWatch.StartNew;
for I := 0 to LoopCount do
  Inc(J, Length(Sample));
Sw.Stop;
Show( 'Length not inlined ' + IntToStr(J) +
      ' [' + IntToStr(Sw.ElapsedMilliseconds) + ' ] ');
end;

```

오브젝트 파스칼 컴파일러에는 인라인으로 처리될 수 있는 함수에 대한 크기 제한이 명확히 정해져 있지 않다. 인라인 처리를 할 수 없는 특정 구조(for 루프, while 루프, 조건문 등)들 역시 정해져 있지 않다. 그러나, 큰 함수를 인라인 처리하는 경우, 장점은 거의 없고 오히려 코드 부풀림으로 인한 단점이 발생할 위험이 실제로 있으므로 사용을 피하는 것이 좋다.

한 가지 제한 사항이 있다. 인라인 되는 메서드 또는 함수는 유닛의 implementation 구현 구역에 정의되어 있는 식별자 identifier(타입 type, 글로벌 변수 global variable, 글로벌 함수 global function 등)를 참조하지 못한다. 호출하는 위치에서는 그 유닛의 implementation 구현 구역에 접근할 수 없기 때문이다. 하지만, 여러분이 로컬 함수를 호출하면 (이것도 인라인으로 처리될 수 있다), 여러분의 인라인 처리 요청을 컴파일러가 허용한다.

인라인 처리의 단점은 유닛들을 더 자주 다시 컴파일하게 된다는 것이다. 인라인이 적용되는 함수를 변경하는 경우에는, 그 함수를 호출하는 곳들도 다시 컴파일을 해야 하기 때문이다. 같은 유닛 안에서는, 인라인 함수들은 자신이 호출되는 위치보다 앞에 작성하면 된다. 하지만, implementation 구현 구역의 시작 부분에 배치하는 것이 더 좋다.

참고 델파이에는 컴파일러는 단일 경로 컴파일러이다. 따라서, 앞에서 보지 못한 함수의 코드를 참조하지 못한다.

다른 유닛들 안에서는, 인라인 되는 함수가 정의되어 있는 유닛을 uses 문에 넣어야 한다. 그 메서드를 직접 호출하지 않더라도 말이다. 유닛 A 가 유닛 B 에 정의된 인라인 함수를 호출한다고 가정하자. 만약, 그 함수가 유닛 C 에 정의된 다른 인라인 함수를 호출한다면, 유닛 A 에서는 유닛 C 도 참조해야 한다. 그렇게 하지 않으면, 컴파일러는 경고 warning 를 통해 유닛 참조가 누락되어 호출이 인라인 되지 않았다고 알려준다. 이와 관련된 효과가 하나 있다. (implementation 구현 구역을 통해) 유닛 참조 순환이 있는 경우에 함수들은 결코 인라인 되지 않는다.

함수의 고급 기능들 Advanced Features of Functions

지금까지 다룬 내용은 함수와 관련된 핵심 기능에 해당된다. 하지만, 살펴볼 가치가 있는 고급 기능들이 몇 가지 더 있다. 소프트웨어 개발에 정말 초보자라면 지금은 이 부분을 건너뛰고 다음 장으로 넘어가는 것이 좋을 수도 있다.

오브젝트 파스칼의 호출 규약들 Object Pascal Calling Conventions

코드에서 함수를 호출하면 그때마다, 파라미터가 실제로 전달되는 방식, 즉 호출 규약 [calling convention](#)에 대해 호출자 [caller](#)와 수신자 [callee](#) 양쪽 모두 합의해야 한다. 함수 호출이 발생하면, 일반적으로, 파라미터를 전달(그리고 반환값을 기대)하는데, 주로 스택 [stack](#) 메모리 영역을 사용한다. 그런데, 스택 안에서 파라미터와 반환값이 배치되는 순서는 프로그래밍 언어와 플랫폼에 따라 달라진다. 그리고 언어 대부분은 여러 가지 호출 규약을 사용할 수 있도록 한다.

오래 전, 델파이 32 비트 버전에서는 "*fastcall* 빠른 호출"이라는 파라미터를 전달하는 새 방식을 도입했다. 이 방식은 가능하다면 파라미터를 최대 3 개까지 CPU 레지스터에 전달한다. 따라서 함수 호출이 훨씬 더 빠르다. 오브젝트 파스칼은 기본 [default](#)으로 이 빠른 호출 규칙을 사용하고 있다. 또한 직접 요청할 수도 있다. [register](#) 키워드를 사용하면 된다.

*fastcall*은 기본 [default](#) 호출 규약이고, 이 규약을 사용하는 함수는 즉 Win32에 있는 Windows API 함수 등 외부 라이브러리와 호환되지 않는다. Win32 API 함수들은 반드시 `stdcall`(표준 [standard](#) 호출) 호출 규약을 사용해 선언되어야 한다. `stdcall`은 Win16 API의 원래 `pascal` 호출 규약과 C 언어의 `cdecl` 호출 규약이 섞인 것이다. 오브젝트 파스칼은 이 모든 호출 규약들을 지원한다. 그러나, 기본 호출 규약이 아닌 다른 규약을 사용하는 일은 거의 없을 것이다. 시스템 라이브러리 등 다른 언어로 작성된 라이브러리를 불러내야 [invoke](#)하는 경우에 필요한 것이기 때문이다.

기본인 *fastcall* 규약에서 벗어나야 하는 일반적인 경우는 플랫폼의 네이티브 API를 호출해야 할 때다. 플랫폼(즉 운영체제)에 맞추어 알맞은 호출 규약을 사용해야 한다. 심지어 Win64와 Win32도 서로 다른 모델을 사용한다. 그래서 오브젝트 파스칼은 다양한 호출 규약 옵션을 지원한다. 하지만, 여기에서 자세히 설명하지는 않겠다. 모바일 운영체제들은 네이티브 함수 대신 클래스를 노출하는 경향이 있다. 그렇다고 해도, 주어진 호출 규약을 준수하는 문제를 고려해야 하는 건 마찬가지다.

프로시저 타입들 Procedural Types

오브젝트 파스칼의 또 다른 특징은 프로시저 타입 [procedural type](#)이다. 이것은 수준 높은 주제이며, 실제로 소수의 프로그래머만 사용한다. 하지만, 우리는 이와 관련된 주제를 다른 장 특히, 메서드 포인터 [method pointer](#) (개발 환경이 이벤트 핸들러를 정의할 때 많이 사용되는 기술), 그리고 익명 메서드 [anonymous method](#)에서 다루게 될 것이다. 그러니 지금 간략히 살펴보기로 하자.

오브젝트 파스칼에는 (전통적인 파스칼 언어에는 없음) 프로시저 타입이라는 개념이 있다 (이는 C 언어에 있는 함수 포인터 [function pointer](#) 개념과 비슷한데, C#, Java와 같은 언어들은 이 개념을 빼 버렸다. 글로벌 함수와 포인터와 엮인 것이기 때문이었다).

프로시저 타입 선언에는 파라미터 목록을 적어준다 (함수의 경우에는 반환 타입까지 포함됨). 예를 들어, 아래와 같이 새 프로시저 타입을 선언할 수 있다. Integer 정수 하나를 참조 파라미터로 가지는 프로시저를 표현하는 타입이다.

```
type
  TIntProc = procedure(var Num: Integer);
```

이 프로시저 타입은 파라미터(또는 C 전문 용어로는 함수 서명 [function signature](#))만 정확히 일치한다면 어떤 루틴 [routine](#) 과도 호환된다. 다음은 이 타입과 호환되는 루틴의 예이다.

```
procedure DoubleIt(var Value: Integer);
begin
  Value := Value * 2;
end;
```

프로시저 타입을 사용하는 목적은 두 가지다. 첫째, 프로시저 타입인 변수를 선언할 수 있다. 둘째, 다른 루틴에게 프로시저 타입(즉, 함수 포인터)을 파라미터로 전달할 수 있다. 위에서 타입을 선언했고 호환할 프로시저도 선언했으니, 이제 아래 코드를 작성할 수 있다.

```
var
  IP: TIntProc;
  X: Integer;
begin
  IP := DoubleIt;
  X := 5;
  IP(X);
end;
```

그런데, 위 코드와 효과가 같은 코드를 아래와 같이 더 짧게 만들 수도 있다.

```
var
  X: Integer;
begin
  X := 5;
  DoubleIt(X);
end;
```

첫 번째 버전이 더 복잡하다는 점은 분명하다. 그렇다면 왜 (그리고 언제) 사용할까? 호출할 함수 결정을 나중에 하는 것이 그리고 실제 호출도 그때 가서 하는 것이 매우 강력한 효과를 내는 상황이 있다. 이 접근 방식을 복잡한 예제로 볼 수도 있겠지만, 지금 매우 간단한 예제를 통해 살펴보자 (ProcType 예제에서 발췌).

아래 예제에서 프로시저는 두 가지를 사용한다. 하나는 이 파라미터 값을 두 배로 늘리는 것이고(이미 앞에서 사용한 DoubleIt), 다른 프로시저는 파라미터 값을 세 배로 늘린다(프로시저 이름은 TripleIt 임).

```
procedure TripleIt(var Value: Integer);
begin
  Value := Value * 3;
end;
```


이 두 함수를 직접 호출하는 대신, 둘 중 하나를 프로시저 타입 변수에 저장해 둔다. 무엇이 저장되는지는 체크박스로 사용자가 바꾼다. 버튼을 클릭하면 변수에 저장된 프로시저가 (제네릭 [generic/일반](#) 방식으로) 호출된다. 프로그램은 글로벌 변수 두 개(호출될 프로시저와 현재 값)를 초기화해 놓고 쓴다. 따라서 이 변수 값들은 계속 보존된다. 전체 코드는 아래와 같다 (단, 앞에 나온 코드는 생략)

```
var
  IntProc: TIntProc = DoubleIt;
  Value: Integer = 1;

procedure TForm1.CheckBox1Change(Sender: TObject);
begin
  if CheckBox1.IsChecked then
    IntProc := TripleIt
  else
    IntProc := DoubleIt;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  IntProc(Value);
  Show(Value.ToString);
end;
```

사용자가 체크박스 상태를 변경하면 작동할 [active](#) 함수가 지정된다. 그리고 나서 버튼을 클릭하면 지정된 그 함수가 호출된다. 따라서 버튼을 두 번 누르고, 체크박스에서 선택을 바꾸고, 다시 버튼을 두 번 누르면, 첫 두 번은 현재 값이 두 배로 늘어나고, 그 다음 두 번은 현재 값이 세 배로 늘어난다. 출력되는 결과는 다음과 같다.

```
2
4
12
36
```

프로시저 타입을 사용하는 또 다른 실용적인 예는 운영체제(윈도우 등)에게 함수를 전달해야 하는 경우다 (일반적으로 "콜백 [callback](#) 함수라고 부름"). 앞에서 언급했듯이 오브젝트 파스칼 개발자는 프로시저 타입 외에도 메서드 포인터 [method pointer](#) (10 장에서 다룸)와 익명 메서드 [anonymous method](#) (15 장에서 다룸)를 사용할 수 있다.

참고 위처럼, 나중에 바인딩되는 [late bind](#) (즉, 호출될 함수가 무엇인지 실행 중에 결정되는) 함수 호출을 구현할 때, 객체 지향 메커니즘에서 가장 일반적으로 쓰는 방식은 가상 메서드 [virtual method](#) 를 사용하는 것이다. 가상 메서드는 오브젝트 파스칼에서 매우 널리 사용되는 반면 프로시저 타입은 거의 사용되지 않는다. 하지만, 그 기술적 기반은 어느 정도 비슷하다. 가상 함수 [virtual function](#) 와 다형성 [polymorphism](#) 은 8장에서 다룬다.

외부 함수 선언 External Functions Declarations

시스템 프로그래밍에서 중요한 또 다른 요소는 외부 선언 [external declaration](#) 이다. 원래는 코드를 어셈블리 언어로 작성된 외부 함수에 연결하는데 쓰였다가 윈도우 프로그래밍에서 DLL(동적 링크 라이브러리 [dynamic link library](#))에 있는 함수를 호출하는 용도로 일반화되었다. 외부 함수 [external function](#) 선언은, 내부 컴파일러나 링커에게 충분히 제공하지 못하는 함수를 호출할 수 있는 능력, 즉 외부 동적 라이브러리를 적재하고 해당 함수 중 하나를 호출할 수 있는 능력이 필요하다는 것을 의미한다.

참고 오브젝트 파스칼 코드 안에서, 특정 플랫폼의 API를 직접 호출하는 코드가 있으면, 그 플랫폼 이외의 플랫폼용으로 애플리케이션을 컴파일할 수 없게 된다. 하지만, 그 외부 호출을 `$IFDEF` 컴파일러 지시어로 둘러싸서 특정 플랫폼에서만 반영되도록 하면 그 제한을 피할 수 있다.

예를 들어, 델파이 애플리케이션에서 Windows API 함수를 호출하는 방법은 아래와 같다. `Winapi.Windows` 유닛을 열면 다음과 같은 함수 선언과 정의를 많이 볼 수 있다.

```
// forward(사전) 선언
function GetUserName(lpBuffer: LPWSTR;
  var nSize: DWORD): BOOL; stdcall;

// external(외부) 선언 (실제 외부 코드를 대신함)
function GetUserName; external advapi32
  name 'GetUserNamew';
```

위와 같은 선언을 작성할 필요는 거의 없다. Windows 유닛과 다른 많은 시스템 유닛들 안에 이 선언들이 이미 나열되어 있기 때문이다. 외부 선언 코드를 직접 작성해야 할 이유가 있다면, 사용자가 만든 맞춤 [custom](#) DLL 안에 있는 함수 또는 플랫폼 API 들 중에서 시스템 유닛 안에 미리 번역되어 들어있지 않은 윈도우 함수를 호출할 때다.

위 선언에 따르면 `GetUserName` 함수의 실제 코드는 `advapi32` (DLL 의 전체 이름인 'advapi32.dll'에 연결된 상수 [constant](#) 임)라는 동적 라이브러리 안에 들어 있는 `GetUserNamew` 함수(이 Windows API 함수로는 ASCII 버전과 WideString 버전 두 가지가 있고, 여기서는 WideString 버전을 사용함)의 코드라는 뜻이다. 이처럼, 외부 선언을 할 때 코드에서 사용할 함수 이름과 실제 DLL 안에 있는 함수 이름은 서로 달라도 된다.

DLL 함수 적재 지연

윈도우 운영체제에서, 윈도우 SDK (또는 기타 모든 DLL) 안에 있는 API 를 불러내는 [invoke](#) 방법은 두 가지다. 외부 함수에 대한 모든 참조를 애플리케이션 로더 [loader](#) 가 해결하도록 두거나, 코드를 직접 작성해서 함수를 찾고 (찾았다면) 실행하도록 것이다.

앞의 방식 코드가 (이미 앞에서 본 코드처럼) 더 쉽다. 여러분은 그저 외부 함수 [external function](#) 선언만 적으면 된다. 하지만, 그 선언에 명시된 라이브러리가 없는 환경이라면 (심지어 호출하려는 함수 하나가 없어도), 그 프로그램은 실행을 시작하지도 못한다.

동적 적재 `dynamic loading` 방식은 더 유연하다. 하지만, 수작업으로 라이브러리를 적재하고, `GetProcAddress` API 를 사용하여 (호출하려는) 함수를 찾고, 찾아 낸 포인터를 알맞은 타입으로 캐스팅 `cast/변환` 한 후에 비로소 호출할 수 있다. 이런 종류의 코드는 상당히 번거롭고 오류가 발생하기 쉽다.

그렇기 때문에, 오브젝트 파스칼 컴파일러와 링커에는 좋은 기능이 들어 있다. 이것은 현재 윈도우 운영체제 수준에 들어 있으며, 일부 C++ 컴파일러에서는 이미 사용되고 있던 것인데, 함수가 호출되기 전까지 적재 `load` 를 지연 `delay` 하는 기능이다. 이 방식으로 선언하는 목적은 DLL 이 암묵적으로 적재되는 동작을 피하려는 게 아니라 (이 동작은 어쨌든 발생함), DLL 안에 있는 함수가 코드에 바인딩되는 동작을 지연할 수 있도록 허용하는 것이다.

기본적으로, 코드 작성에 있어서는 DLL 함수를 일반적으로 실행할 때와 비슷하다. 하지만, 함수의 주소가 결정되는 시점은 애플리케이션이 적재될 때가 아니라 함수를 맨 처음 호출할 때다. 즉, 찾는 함수가 없으면 런타임 예외인 `EExternalException` 이 발생한다. 그렇지만, 대체로 여러분은 운영체제의 현재 버전 확인 또는 호출하려는 특정 라이브러리의 버전 확인을 먼저 하고 나서, 그 호출을 할 것인지 아닌지를 결정하면 된다.

참고 예외`exception`가 아니라, 더 구체적이고 글로벌 수준에서 처리하기 쉬운 방식을 원한다면, 지연된 적재`delayed loading` 호출에 대한 예러 메커니즘에 연결할 수 있다. 설명은 Allen Bauer의 블로그 글 (https://blog.therealoracleatdelphi.com/2009/08/exceptional-procrastination_29.html)을 참조하자.

오브젝트 파스칼 언어의 관점에서 보면, 함수 적재 `load` 를 지연하는 코드의 다른 점은 외부 함수 `external function` 선언 부분뿐이다. 아래와 같이 쓰지 않고:

```
function MessageBox;  
external user32 name 'MessageBoxW';
```

이제 아래와 같이 쓰면 된다 (역시, Windows 유닛 안에 있는 실제 코드다).

```
function GetSystemMetricsForDpi(nIndex: Integer; dpi: UINT): Integer;  
stdcall; external user32 name 'GetSystemMetricsForDpi' delayed;
```

위 예문에 있는 `GetSystemMetricsForDpi` API 는 윈도우 10 에서, 보다 정확히 1607 버전에서 처음으로 추가되었다. 따라서, 이 함수를 호출하기 전에는 실행 환경을 먼저 확인하는 것이 좋다. 그러니 사용할 때에는 다음과 같이 코드를 작성하도록 하자.

```
if (TOSVersion.Major >= 10) and (TOSVersion.Build >= 14393)  
begin  
  NMetric := GetSystemMetricsForDpi(SM_CXBORDER, 96);
```

예전 방식에 비해, 위 코드는 훨씬 더 짧다. 옛 버전의 윈도우에서 동일한 프로그램을 실행하기 위해 여러분이 지연 적재 없이 작성해야 했던 코드에 비하면 그렇다.

자체 DLL 을 직접 구축하고 그 DLL 을 오브젝트 파스칼에서 호출하는 경우에도 이와 동일한 메커니즘이 사용된다는 점을 알아 두자. 즉, 실행 파일은 하나만 제공하고, 동일한 DLL 의 여러 버전을 그 실행 파일에 바인딩 할 수 있다. 그리고, 지연 적재를 사용해 새 함수에 연결할 수 있다.

05: 배열과 레코드 Arrays and Records

2장에서 데이터 타입을 소개하면서, 오브젝트 파스칼에 내장 데이터 타입 [built in data type](#)과 타입 생성자 [type constructor](#)가 모두 있다는 사실을 언급했다. 타입 생성자의 간단한 예는 열거되는 타입 [enumerated type](#)이다. 이것도 역시 2장에서 소개했다.

타입 정의가 강력한 능력을 발휘하는 곳은 배열 [array](#), 레코드 [record](#), 클래스 [class](#) 등 고급 메커니즘을 다룰 때다. 이 장에서는 배열과 레코드를 다룬다. 이 두 타입은 파스칼 초기 정의에 있던 본질을 그대로 간직하면서도, 수년에 걸쳐 많이 변경되었다 (그리고 더 강력해졌다). 타입 생성자도 옛 조상과 거의 닮지 않았다. 이름은 똑같지만 말이다.

이 장이 막바지에는, 포인터 등 오브젝트 파스칼의 고급 데이터 타입들을 간략하게 소개할 것이다. 하지만 사용자 정의 데이터 타입 [custom data type](#)의 진정한 능력은 7장에서 공개된다. 7장부터 우리는 클래스와 객체-지향 프로그래밍을 살펴보기 시작할 것이다.

배열 데이터 타입 Array Data Types

배열 타입 [array type](#)은 타입이 같은 요소들이 나열된 목록을 정의한다. 목록(즉 배열)에는 두 종류가 있다. 즉 그 안에 들어가는 요소의 개수가 고정된 것(정적 배열 [static array](#))과 그 개수가 변할 수 있는 것(동적 배열 [dynamic array](#))이 있다. 배열 요소들 중 어느 하나에 접근하려면 인덱스 [index/순번](#)을 대괄호 안에 적는 것이 일반적이다. 대괄호는 정적 배열에 들어가는 값들의 개수를 정의할 때에도 사용된다.

오브젝트 파스칼 언어가 지원하는 배열 타입은 기존의 정적 배열부터 동적 배열까지 다양하다. 특히 모바일 버전의 컴파일러를 사용할 때는 동적 배열을 쓰는 것이 좋다. 정적 배열에 대해 먼저 보고나서 동적 배열을 집중적으로 살펴보자.

정적 배열 Static Arrays

기존 파스칼 언어에서 배열은 정적 `static` 즉 크기를 고정하여 정의된다. 하루 24 시간별 온도를 나타내는 24 개의 정수를 담은 목록을 타입으로 정의하면 아래와 같다.

type

```
TDayTemperatures = array[1..24] of Integer;
```

고전적인 배열 정의는, 하위범위 타입 `subrange type` 을 사용할 수 있다. 실제로 위에서는 대괄호 그리고 순서 타입 `ordinal type` 상수 `constant` 두 개를 사용하여 새 하위범위 타입을 정의하고 있다. 이 하위범위는 그 배열에서 유효한 인덱스 `index` 를 명시하고 있다. 즉, 배열 인덱스의 상한과 하한을 직접 지정하는 방식이다. 따라서, 반드시 인덱스가 0 부터 시작하지 않아도 된다 (C, C++, Java 등 대부분의 다른 언어에서는 인덱스가 무조건 0 부터 시작함). 오브젝트 파스칼은 순서 타입이지만 하면 무엇이든 정적 배열에서 인덱스로 사용될 수 있다. 따라서 숫자 이외에도 문자, 열거 타입 등을 사용할 수도 있다. 하지만, 숫자가 아닌 것이 인덱스로 사용되는 경우는 매우 드물다.

참고

JavaScript 등 연관 배열 `associative array` 을 많이 쓰는 언어들이 있다. 오브젝트 파스칼에서 배열은 순서 `ordinal` 값만 인덱스가 될 수 있다. 즉, 문자열 `string` 을 인덱스로 직접 사용할 수 없다. 하지만, 딕셔너리 `Dictionary` 또는 이와 비슷한 데이터 구조들이 RTL 안에 있다. 연관 배열 기능 같은 것이 필요할 때 바로 쓰면 된다. 자세한 내용은 제네릭 `Generics` 을 다루는 장(이 책의 3부)에서 살펴본다.

이 배열의 인덱스는 그 기반이 하위범위 `subrange` 다. 따라서, 컴파일러는 이 인덱스의 범위를 점검할 수 있다. 하위범위를 지정하는 상수가 유효하지 않으면 컴파일-시간에 에러가 발생한다. 그리고 실행-시간에 범위를 벗어난 인덱스를 사용하면 실행-시간에 에러가 발생한다(단, 해당 컴파일러 옵션이 활성화된 경우).

참고

이 옵션은 Range checking `범위 확인` 옵션이다. IDE의 Project Options `프로젝트 옵션` 대화 상자> Compiling `컴파일` 페이지> Runtime errors `런타임 에러` 그룹 안에 있다. 이 옵션에 대해서는 이미 2장의 "하위범위 `Subrange` 타입" 부분에서 언급했다.

위에 정의된 배열을 사용하여 DayTemp1 변수를 TDayTemperatures 타입으로 지정할 수 있다 (아래 코드 참조, ArraysTest 예제에서 발췌함).

type

```
TDayTemperatures = array[1..24] of Integer;
```

var

```
DayTemp1: TDayTemperatures;
```

begin

```
DayTemp1[1] := 54;
```

```
DayTemp1[2] := 52;
```

```
...
```

```
DayTemp1[24] := 66;
```

```
// 아래 줄은 다음 에러가 발생함:
```

```
// E1012 Constant expression violates subrange bounds
```

```
// DayTemp1[25] := 67;
```


배열을 다루는 표준 방법은, 그 본질이 그러하듯, for-루프 [for-loop](#) 를 사용하는 것이다. 아래 코드는 루프를 사용하여 하루 24 시간의 온도를 표시한다.

```
var
  I: Integer;
begin
  for I := 1 to 24 do
    Show(I.ToString + ': ' + DayTemp1[I].ToString);
```

이 코드는 잘 작동한다. 하지만 배열 경계(1 과 24)를 하드-코딩하는 것은 전혀 이상적인 방법이 아니다. 시간이 지나면서 배열 정의 자체가 변경될 수 있고, 동적 배열을 사용하도록 바꾸고 싶을 수도 있기 때문이다.

배열 크기 및 경계 [Array Size and Boundaries](#)

배열로 작업할 때, 언제나 표준 함수인 Low 와 High 를 사용해 범위의 상한과 하한을 테스트할 수 있다. Low 와 High 사용은 적극 권장한다. 특히 루프 안에서라면, 배열의 현재 범위 값의 영향을 받지 않는 코드를 만들 수 있다 (범위 값은 0 부터 마지막 값은 배열의 크기보다 1 작은 정수까지, 또는 1 부터 배열의 크기를 나타내는 정수까지, 또는 하위범위에 정의된 다른 범위일 수 있음). 나중에 배열 선언에서 인덱스 범위를 변경하는 상황이 와도, Low 와 High 를 사용한 코드는 배열 정의에서 그 값을 가져오므로 자동으로 해당 범위가 반영된다. 이와 달리, 루프를 작성할 때 그 범위를 하드-코딩한다면, 배열 범위가 변경되었을 때 루프 코드를 다시 고쳐야 한다. 요컨대, Low 와 High 를 사용하면 코드를 유지 관리도 쉽지만, 안정성도 높아진다.

참고 의도한 대로, 정적 배열에서 Low와 High를 사용하는 것은 실행-시간에 추가 부담이 전혀 없다. 컴파일할 때 이미 알맞은 상수 표현식으로 해소된다. 즉 함수 호출은 실제로 발생하지 않는다. 이렇게 표현식과 함수 호출이 컴파일 시 해소되는 일은 다른 많은 시스템 함수들도 마찬가지다.

배열의 범위와 관련된 또다른 함수는 Length 다. Length 는 배열의 요소 [element](#) 개수를 반환한다. 앞에 있던 코드 조각 세 가지를 활용하는 코드를 보자. 아래 코드는 하루의 평균 온도를 계산하고 표시한다.

```
var
  I: Integer;
  Total: Integer;
begin
  Total := 0;
  for I := Low(DayTemp1) to High(DayTemp1) do
    Inc(Total, DayTemp1[I]);
  Show((Total / Length(DayTemp1)).ToString);
```

(위 코드 역시 ArraysTest 예제에서 발췌함).

다-차원 정적 배열 Multi-Dimensional Static Arrays

배열은 1차원을 넘을 수 있다. 즉, 목록 *list*(1차원) 외 행렬 *matrix*(2차원), 큐브 *cube*(3차원) 등을 표현할 수 있다. 아래 정의를 보자.

```
type
  TAllMonthTemps = array[1..24, 1..31] of Integer;
  TAllYearTemps = array[1..24, 1..31, 1..12] of Integer;
```

요소 하나에 접근하는 코드는 아래와 같다.

```
var
  AllMonth1: TAllMonthTemps;
  AllYear1: TAllYearTemps;
begin
  AllMonth1[13, 30] := 55; // 시, 일
  AllYear1[13, 30, 8] := 55; // 시, 일, 월
```

참고 정적 배열은 많은 메모리(위의 경우 스택 *stack*)를 즉시 차지한다. 그러니 피하는 것이 좋다. AllYear1 변수에는 Integer *정수*가 8,928개 들어간다. Integer가 4바이트 *byte*니까, 거의 35KB를 차지한다. (위 코드처럼) 큰 블록을 글로벌 *global* 메모리나 스택 *stack*에 할당하는 것은 정말 실수다. 이와 달리, 동적 배열은 힙 *heap* 메모리를 사용한다. 그리고 훨씬 더 유연하게 메모리를 할당하고 관리할 수 있도록 지원한다.

위에 정의한 두 배열 타입은 똑같은 핵심 타입(Integer)을 기반으로 한다. 따라서, 먼저 만든 데이터 타입을 사용하여 선언하는 것이 더 좋다. 다음 코드를 보자.

```
type
  TMonthTemps = array[1..31] of TDayTemperatures;
  TYearTemps = array[1..12] of TMonthTemps;
```

위 선언은 더 앞에서 사용한 방식과 인덱스의 순서가 반대이다. 하지만, 전체 블록을 할당할 수 있다는 점은 똑같다.

```
Month1[30][14] := 44; // 일, 시
Month1[30, 13] := 55; // 일, 시
Year1[8, 30, 13] := 55; // 월, 일, 시
```

이렇게 중간 타입 *intermediate type*을 사용하는 것을 받쳐주는 중요한 사실이 있다. 이 배열 타입들은 서로 호환될 수 있는데 그 이유는 정확히 일치하는 타입 이름 (정확히 똑같은 타입 정의)을 참조하기 때문이지, 타입 정의의 구현이 똑같기 때문이 아니라는 점이다. 이러한 타입 호환성 규칙은 몇 가지 예외만 빼고 오브젝트 파스칼의 모든 타입에 적용된다.

예를 들어, 아래 문장은 그 달의 온도를 복사해서 그 해의 세 번째 달에 넣는다.

```
Year1[3] := Month1;
```


그런데, 처음에 예를 들었던 독립형 배열 정의들(타입 호환이 서로 되지 않음)을 사용해서 비슷하게 서술해보자.

```
AllYear1[3] := AllMonth1;
```

위 문장은 (타입 호환성) 에러가 발생할 것이다.

```
Error: Incompatible types: 'array[1..31] of array[1..12] of Integer'
and 'TAllMonthTemps'
```

언급했듯이, 정적 배열은 메모리 관리 문제가 있다. 특히 파라미터로 전달하거나 큰 배열 중 일부만 할당하는 경우라면 더욱 그렇다. 게다가, 정적 배열 변수를 한 번 만들면, 없어지기 전까지 크기를 바꾸지 못한다. 그렇기 때문에, 동적 배열을 사용하는 것이 좋다. 동적 배열이 비록 메모리 할당과 같은 추가 관리가 조금 필요하더라도 말이다.

동적 배열 Dynamic Arrays

기존 파스칼에서 배열은 고정 크기이고, 데이터 타입 선언에 적힌 요소의 개수로 제한되지만, 오브젝트 파스칼은 동적 배열을 직접 네이티브로 구현하도록 지원한다.

참고 여기서 "동적 배열을 직접 구현하는 방식"과 대조되는 방식은 포인터 [pointer](#)와 동적 메모리 할당 [dynamic memory allocation](#)을 사용하여 매우 복잡하고 에러가 발생하기 쉬운 코드로 비슷한 효과를 얻는 것이다. 참고로, 동적 배열은 대부분의 최신 프로그래밍 언어에서 널리 사용되는 구조이다.

동적 배열 [dynamic array](#) 은 동적으로 할당 [dynamically allocated](#) 되고 참조 카운트 [reference counted](#) 된다 (파라미터로 전달될 때 속도가 훨씬 더 빠르다. 배열의 전체 복사본이 아니라 참조만 전달되기 때문이다). 작업이 끝나면, 동적 배열을 지울 수 있다. 해당 변수를 `nil` 로 지정하거나, 길이를 0 으로 설정하면 된다. 동적 배열은 컴파일러가 자동으로 메모리를 해제 [free](#) 한다. 참조 카운트되기 때문이다. 자동 해제는 그 배열의 항목들이 사용하는 메모리에 대해서만 수행된다는 점을 알아야 한다. 만약 그 배열의 항목에 들어있는 데이터가 참조 [reference](#) 여서 또다른 메모리 위치를 가리키고 있다면 (오브젝트 참조 등) 그 참조되는 오브젝트가 차지하던 메모리 위치도 비웠는지 확인한 다음에 그 배열을 해제해야 한다.

동적 배열에서는, 요소의 개수를 지정하지 않고 배열 타입을 선언한다. 그리고 나서 `SetLength` 프로시저를 사용하여 배열의 크기를 할당한다.

```
var
  Array1: array of Integer;
begin
  // 실행 시간에 범위-확인(range-check) 에러가 발생할 수 있다
  // Array1[0] := 100;
  SetLength(Array1, 10);
  Array1[0] := 100; // 이제 괜찮다
```


위 배열은 길이를 할당하기 전에 사용할 수 없다. 필요한 메모리를 힙 [heap](#)에 할당해야 사용할 수 있기 때문이다. 그러면, Range Check [범위 확인](#) 에러가 나거나 (컴파일러에 해당 옵션이 활성화되어 있는 경우), 또는 윈도우에서 Access Violation [접근 위반](#)이나 다른 플랫폼에서 이와 유사한 메모리 접근 에러가 발생한다. SetLength를 호출하면 배열의 모든 요소 값이 0으로 지정된다. 이 초기화 코드가 실행된 뒤에는 이 배열에 값을 읽고 쓸 수 있다. 메모리 에러를 걱정할 필요가 없다(단, 배열의 범위를 넘지 않아야 한다).

동적 배열은 메모리를 명시적으로 할당해야 하지만, 직접 해제 [free](#) 할 필요는 없다. 위 코드 조각이 끝나면 Array1 변수는 생존 범위를 벗어나게 되고, 컴파일러는 자동으로 그 메모리를 해제한다(위 경우에는 Integer [정수](#) 10개를 할당했던 메모리). 따라서 이 동적 배열 변수를 nil로 지정하거나 SetLength(0)을 호출해도 되지만, 그럴 필요는 대체로 없다 (매우 드물게 필요한 경우가 있다).

SetLength 프로시저는 배열의 크기를 조정할 때에도 사용할 수 있다. 배열을 늘리는 경우에는 들어있는 내용을 잃지 않는다. 하지만, 배열을 줄이는 경우에는 일부 요소를 잃을 수 있다. SetLength를 처음 호출할 때는 요소의 개수만 명시할 수 있다. 동적 배열의 인덱스는 무조건 0부터 시작하고 요소 개수에서 1을 뺀 숫자에서 끝난다. 다시 말해, 동적 배열은 고전적인 정적 파스칼 배열에 있는 기능 중 두 가지 즉, 하한을 0이 아닌 숫자로 시작할 수 있는 기능과 숫자가 아닌 인덱스를 사용할 수 있는 기능을 지원하지 않는다. 이와 동시에, 동적 배열은 C 구문을 기반으로 하는 대부분의 언어들의 배열과 작동하는 방식에 오히려 더 가깝다.

동적 배열의 현재 크기를 알려면 [query](#), 정적 배열과 마찬가지로 방식으로, Length, High, Low 함수를 사용한다. 그런데, 동적 배열에서 Low는 항상 0을 반환하고, High는 항상 배열의 길이 - 1을 반환한다. 따라서, 배열이 비어 [empty](#) 있는 경우에, High가 반환하는 값은 -1이다 (생각해보면 이상한 값이다. 상한 값이 하한 값보다 작으니 말이다).

이제 적응형 루프 [adaptable loop](#)를 사용하여 동적 배열에서 정보를 채우고 추출하는 코드를 보자 (DynArray 예제에서 발췌함).

```
type
  TIntegersArray = array of Integer;
var
  IntArray1: TIntegersArray;
```

위 배열을 할당하고 [allocate](#) 값을 채우는 [populate](#) 코드는 아래와 같다. 루프를 이용하여 각 배열 요소의 값에 인덱스와 똑같은 숫자를 넣는다.

```
var
  I: Integer;
begin
  SetLength(IntArray1, 20);
  for I := Low(IntArray1) to High(IntArray1) do
    IntArray1[I] := I;
end;
```


두 번째 버튼의 코드를 보자. 값을 하나씩 표시하고 합계를 구한다. 그리고 마지막에 평균을 계산한다. 위에서 본 예문과 비슷하다.

```
var
  I: Integer;
  Total: Integer;
begin
  Total := 0;
  for I := Low(IntArray1) to High(IntArray1) do
    begin
      Inc(Total, IntArray1[I]);
      Show(I.ToString + ': ' + IntArray1[I].ToString);
    end;
  Show('평균: ' + (Total / Length(IntArray1)).ToString);
end;
```

출력 결과는 당연히 아래와 같다 (대부분 생략됨).

```
0: 0
1: 1
2: 2
3: 3
...
17: 17
18: 18
19: 19
평균: 9.5
```

Length, SetLength, Low, High 외에도 배열에서 일반적으로 사용할 수 있는 프로시저들이 있다. 예를 들어 Copy 함수는 배열의 일부(또는 전체)를 복사한다. 또한 배열을 담은 변수를 다른 변수에 할당할 수 있다. 그런데 이 경우에는 전체 복사가 아니라 두 변수가 같은 메모리 즉 같은 배열을 참조 [refer](#) 한다.

조금 복잡한 코드를 보자. 이 코드는 배열을 두 가지 방법으로 복사한다(DynArray 예제의 뒷부분에서 발췌함).

- Copy 함수 사용하기: 배열 데이터를 똑같이 하나 더 만들고, 다른 메모리 영역에 있는 새 데이터 구조 안에 넣는다.
- 할당 연산자 [assignment operator](#) 사용하기: 별칭 [alias](#), 즉 새 변수를 만들어서, 같은 메모리 즉 같은 배열을 참조한다.

이 두 방법으로 복사된 새 배열을 변경하면, 복사 방법에 따라 원본에 영향을 주는 것과 그렇지 않은 것을 알 수 있다. 코드를 보자.

```
var
  IntArray2: TIntegersArray;
  IntArray3: TIntegersArray;
begin
  // 별칭(Alias)
  IntArray2 := IntArray1;
```



```
// 별도의 복사본
IntArray3 := Copy(IntArray1, Low(IntArray1), Length(IntArray1));

// 새 배열에서 항목 변경
IntArray2[1] := 100;
IntArray3[2] := 100;

// 각 배열에서 값을 확인
Show(Format('%d] %d -- %d -- %d',
  [1, IntArray1[1], IntArray2[1], IntArray3[1]]));
Show(Format('%d] %d -- %d -- %d',
  [2, IntArray1[2], IntArray2[2], IntArray3[2]]));
```

출력은 다음과 같다.

```
[1] 100 -- 100 -- 1
[2] 2 -- 2 -- 100
```

IntArray2 에서 변경한 내용은 IntArray1 에 전파된다. 그 두 참조가 물리적으로 같은 배열을 참조하고 있기 때문이다. 이와 달리, IntArray3 에서 변경한 내용은 원본과 별개이다. 별도의 데이터 복사본이기 때문이다.

동적 배열의 네이티브 연산 Native Operations on Dynamic Arrays

동적 배열은 상수 배열 [constant array](#) 할당을 지원하고 이어 붙이기 [concatenation](#) 도 지원한다.

참고 동적 배열에 이 기능이 추가된 것은 델파이 XE7부터이다.

실제로, 아래와 같이 코드를 쓸 수 있다. 앞에서 본 코드 조각에 비해 매우 단순하다.

```
var
  DI: array of Integer;
  I: Integer;
begin
  DI := [1, 2, 3]; // 초기화(Initialization)
  DI := DI + DI; // 이어 붙이기(Concatenation)
  DI := DI + [4, 5]; // 혼합식 이어 붙이기(Mixed concatenation)

  for I in DI do
    begin
      Show(I.ToString);
    end;
```

for-in 루프를 사용하여 배열 요소를 스캔하고 있다는 점을 보자(DynArrayConcat 예제에서 발췌함). 위에 있는 배열은 기반 타입이 단순한 정수 [Integer](#) 지만 레코드 [record](#), 클래스 [class](#) 등 어떤 타입이든 기반 타입이 될 수 있다.

할당, 이어 붙이기 외에 추가된 것이 더 있다. 언어라기보다 RTL 에 들어간 것인데, 문자열 [string](#) 에서 흔히 쓰이는 함수들(Insert, Delete 등)을 동적 배열에 쓸 수 있다.

즉, 이제 다음과 같은 코드를 작성할 수 있다(동일한 프로젝트에서 발췌함).

```
var
  DI: array of Integer;
  I: Integer;
begin
  DI := [1, 2, 3, 4, 5, 6];
  Insert([8, 9], DI, 4);
  Delete(DI, 2, 1); // 세 번째 항목을 제거한다. (0 기반 인덱스임)
```

오픈 배열 파라미터 Open Array Parameters

배열을 사용하는 매우 특별한 상황이 있는데, 바로 유연한 파라미터 목록을 함수에게 전달하는 것이다. 배열을 직접 전달하는 것 외에 특별한 구문 규약 [syntax convention](#) 두 가지가 있다(이 절과 다음 절에서 설명한다). 이 규약 중 하나를 활용한 예로 `Format` 함수가 있다 (이미 예문에서 사용한 적이 있다). `Format` 함수의 두 번째 파라미터에는 그 자리에서 값들이 정의되는 배열 [in-place defined array](#) 이 사용된다.

C 언어(및 C 구문을 기반으로 하는 다른 일부 언어)와 달리, 전통적인 파스칼 언어에서는 함수나 프로시저의 파라미터 수는 항상 고정되어 있다. 그러나 오브젝트 파스칼에서는 파라미터의 수가 달라도 루틴에게 전달하는 방법이 있다. 그 자리에서 정의되는 배열 안에 담으면 된다. 즉, 오픈 배열 파라미터라는 기술을 사용하면 된다.

참고 역사적으로, 오픈 배열 [Open Array](#) 파라미터는 동적 배열 [dynamic array](#) 보다 더 먼저 사용되었다. 하지만, 오늘날 이 두 기능은 작동 방식이 매우 비슷하여 거의 구분할 수 없다. 그래서 이 책은 동적 배열을 먼저 설명한 다음에 오픈 배열 파라미터를 다루었다.

오픈 배열 파라미터의 기본적인 정의 [basic definition](#) 는 타입이 지정된 [typed](#) 동적 배열 타입과 같고, 앞에는 `const` 지정자 [specifier](#) 를 붙인다. 즉, 파라미터의 타입은 알려주지만 배열에 해당 타입으로 된 요소가 몇 개 포함될지는 알려주지 않아도 된다. 기본적인 정의를 예문으로 보자 (`OpenArray` 예제에서 발췌함)

```
function Sum(const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

위 함수를 호출할 때, `array of Integer` 상수 표현식에 변수 (개별 값을 계산하는데 사용됨)가 들어가도 된다.

```
X := Sum([10, Y, 27 * I]);
```


동적인 array of Integer 가 준비되면, 이것을 루틴 [routine](#) 에게 직접 전달할 수 있다 (단, 기반 타입이 동일한, 즉 기본 타입이 Integer 인 오픈 배열 파라미터를 가지고 있는 루틴이어야 함). 배열은 통째로 파라미터로 전달된다. 아래 예문과 같다.

```
var
  List: array of Integer;
  X, I: Integer;
begin
  // 동적 배열 초기화(Initialize)
  SetLength(List, 10);
  for I := Low(List) to High(List) do
    List[I] := I * 2;

  // 함수를 사용하여 배열 요소의 합계 구하기
  X := Sum(List);
```

위 코드는 동적 배열을 파라미터로 사용했다. 정적 배열이라면 어떨까? 마찬가지로. 기반 타입이 일치하는 오픈 배열 파라미터를 가진 루틴에게 전달할 수 있다. 또한 slice 함수를 호출하여 기존 배열의 일부(두 번째 파라미터에 명시한 숫자만큼)만 전달할 수도 있다. 아래 코드는 정적 배열을 전달하는 방법을 보여준다. Sum 함수에게 배열(또는 그 일부)을 전달하고 있다.

```
var
  List: array[1..10] of Integer;
  X, I: Integer;
begin
  // 정적 배열 초기화(Initialize)
  for I := Low(List) to High(List) do
    List[I] := I * 2;

  // 함수를 사용하여 배열 요소의 합계 구하기
  X := Sum(List);
  Show(X.ToString);

  // 배열 일부의 합계 구하기
  X := Sum(Slice(List, 5));
  Show(X.ToString);
```

타입 변형 오픈 배열 파라미터 [Type-Variant Open Array Parameters](#)

타입이 지정된 [typed](#) 동적 배열 [open array](#) 파라미터 외에도, 오브젝트 파스칼 언어에는, 타입 변형 [type-variant](#) 즉, 타입이 지정되지 않은 [untyped](#) 오픈 배열을 정의할 수 있다. 이 특별한 배열은 요소의 개수가 정의되지 않을 뿐만 아니라, 요소의 데이터 타입도 정의되지 않기 때문에 배열의 각 요소가 서로 다른 타입으로 전달될 수 있다. 이것은 이 언어에서 예외적으로 타입 안정성이 완전히 지켜지지 않는 영역 중 하나다.

기술적으로, 파라미터를 `array of const` 타입으로 정의하면 요소의 개수와 타입을 원하는 대로 답아서 함수에게 전달할 수 있다. 예를 들어, 아래에 있는 `Format` 함수의

정의를 보자 (이 함수를 사용하는 방법은 6 장에서 문자열을 다루면서 살펴본다. 하지만, 우리는 이미 앞에서 한번 사용해 보았다).

```
function Format(const Format: string;
  const Args: array of const): string;
```

위 함수에서 두 번째 파라미터는 오픈 배열이다. 즉, 값의 개수가 정해져 있지 않다. 실제로, 이 함수를 호출하는 방법은 다음과 같다.

```
N := 20;
S := '합계: ';
Show(Format('합계: %d', [N]));
Show(Format('Int: %d, Float: %f', [N, 12.4]));
Show(Format('%s %d', [S, N * 2]));
```

파라미터를 상수 값, 변수 값, 표현식 중 어느 것으로도 전달할 수 있다는 점에 주목하자. 이런 종류의 함수를 선언하는 것은 간단하다. 하지만, 코딩은 어떻게 하면 될까? 파라미터의 타입을 알려면 어떻게 할까? 타입 변형 오픈 배열 파라미터의 값은 TVarRec 타입의 요소들과 호환된다.

참고 TVarRec 레코드를 TVarData 레코드(Variant 타입이 사용)와 혼동하지 말자. 이 두 구조는 목적도 다르고 서로 호환되지도 않는다. 심지어 변환될 수 있는 타입 목록도 다르다. 왜냐하면 TVarRec은 오브젝트 파스칼 데이터 타입을 담을 수 있는 반면, TVarData는 윈도우 OLE 데이터 타입을 담을 수 있기 때문이다. 배리언트 [variant](#)에 대해서는 이 장의 뒷부분에서 다룬다.

다음은 타입 변형 오픈 배열 값과 TVarRec 레코드가 지원하는 데이터 타입이다.

vtInteger	vtBoolean	vtChar
vtExtended	vtString	vtPointer
vtPChar	vtObject	vtClass
vtWideChar	vtPWideChar	vtAnsiString
vtCurrency	vtVariant	vtInterface
vtWideString	vtInt64	vtUnicodeString

레코드 구조를 보자. 그 안에는 타입을 담는 필드(vType)와 실제 데이터를 접근하는 필드가 있다 (레코드에 대해서는 몇 페이지 뒤부터 자세히 설명한다. 참고로 이것은 레코드 중에서도 고급 사용법에 해당한다)

전형적인 접근 방식은 case 문을 써서, 다양한 타입으로 전달되는 파라미터들을 각자의 타입에 맞게 연산하는 것이다. 아래 예제에서는 다양한 타입으로 전달되는 값들의 합계를 구한다 (SumAll 함수 예제에서 발췌함). 즉 문자열을 정수로, 문자를 그 문자가 위치한 순서 값으로, 불리언 bool True 를 1 로 변환한 후에 모두 더한다. 이 코드는 확실히 상당히 수준이 높다 (또한 포인터 역참조 [dereference](#) 를 사용한다). 그러니 지금 완전히 이해하지 못하더라도 걱정하지 말자.

```
function SumAll(const Args: array of const): Extended;
var
```



```

I: Integer;
begin
  Result := 0;
  for I := Low(Args) to High(Args) do
    case Args[I].VType of
      vtInteger:
        Result := Result + Args[I].VInteger;
      vtBoolean:
        if Args[I].VBoolean then
          Result := Result + 1;
      vtExtended:
        Result := Result + Args[I].VExtended^;
      vtWideChar:
        Result := Result + Ord(Args[I].VWideChar);
      vtCurrency:
        Result := Result + Args[I].VCurrency^;
    end; // Case
  end;
end;

```

위 함수를 호출하는 코드는 아래와 같다 (OpenArray 예제에 추가되어 있음)

```

var
  X: Extended;
  Y: Integer;
begin
  Y := 10;
  X := SumAll([Y * Y, 'k', True, 10.34]);
  Show('합계: ' + X.ToString);
end;

```

이 호출은 y 의 제곱, K 의 자리 순서 값(107 이다), 불리언 `bool`의 값 1, Extended 타입 숫자를 모두 더한다. 결과는 다음과 같다.

합계: 218.34

레코드 데이터 타입 Record Data Types

배열은 동일한 항목들로 구성되는 목록을 정의하고, 숫자 인덱스를 사용하여 접근한다. 이와 달리, 레코드는 타입이 다양한 여러 요소들의 묶음을 정의하고, 접근할 때는 요소의 이름을 사용한다. 즉, 레코드는 이름이 붙은 항목들 즉 필드 `field`들로 구성되는 목록이다. 그리고 각 필드는 각자 자신들의 데이터 타입이 있다. 레코드 타입 정의 `type definition`에서는 모든 필드들을 나열하고 각 필드에 이름을 붙인다. 참조할 때 사용하기 위해서다. 파스칼 초창기에는 레코드에 필드만 있었지만, 이제는 메서드 `method`와 연산자 `operator`도 가질 수 있다. 이 장에서 살펴볼 것이다.

참고 레코드는 프로그래밍 언어 대부분에서 사용된다. C 언어에서는 `struct` 키워드로 정의된다. C++에서는 메서드를 포함하도록 정의가 확장되었다. 오브젝트 파스칼에도 이러한 확장이 반영되었다. 일부 "순수" 객체-지향 언어들은 클래스 `class` 개념만 있고 레코드 `record`나 구조체 `structure` 개념이 없지만, C#은 최근 이 개념을 다시 도입했다.

코드 조각을 보자 (RecordsDemo 예제에서 발췌함). 레코드 타입을 정의하고, 그 타입으로 여러 변수를 선언하고, 그 변수들을 사용하여 서술하는 코드다.

```
type
  TMyDate = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;
var
  ABirthday: TMyDate;
begin
  ABirthday.Year := 1997;
  ABirthday.Month := 2;
  ABirthday.Day := 14;
  Show( '태어난 연도 ' + ABirthday.Year.ToString);
```

참고 레코드 *record*라는 용어는 오브젝트 파스칼 언어에서 때때로 명확하지 않게 사용되는 경향이 있다. 레코드 타입, 그리고 레코드 타입인 변수(즉 레코드 인스턴스)는 서로 다른 요소인데, 두 요소 모두 그냥 레코드라고 부른다. 이 점은 클래스 타입과 명확히 구분되도록 클래스의 인스턴스를 오브젝트 *object*라고 부르는 것과 비교된다.

오브젝트 파스칼에서, 레코드 데이터 구조는 단순히 필드들의 목록만이 아니라 훨씬 더 많은 것들이 있다. 이 장에서 하나씩 살펴보겠다. 지금은 레코드에 대한 전통적인 접근 방식부터 시작하자. 레코드가 사용하는 메모리 영역은 로컬 *local/지역* 변수의 경우 스택 *stack* 메모리이고, 글로벌 *global/전역* 변수의 경우 글로벌 메모리다. SizeOf 를 호출하면 변수나 타입에게 필요한 바이트 *byte* 개수를 얻을 수 있다. 아래 문장과 같다.

```
Show( '레코드 크기: ' + SizeOf(ABirthday).ToString);
```

Win32에서 기본 *default* 컴파일러 설정이면 반환 값은 8이다(6이 아니라 8을 반환하는 이유는 Integer *정수* 필드에 4 바이트, 각 *Byte* 바이트 필드에 각각 2 바이트가 사용되기 때문이다. 이 점은 필드 정렬 *Fields Alignments* 절에서 설명한다).

이처럼, 레코드는 값 *value* 타입이다. 즉, 레코드를 다른 레코드에 할당하면 전체 복사본이 만들어진다. 복사본을 변경한다고 해서 원본 레코드에 영향이 가지 않는다. 아래 코드 조각을 통해 이 개념을 직접 보자.

```
var
  ABirthday: TMyDate;
  ADay: TMyDate;
begin
  ABirthday.Year := 1997;
  ABirthday.Month := 2;
  ABirthday.Day := 14;

  ADay := ABirthday;
  ADay.Year := 2008;

  Show(MyDateToString(ABirthday));
  Show(MyDateToString(ADay));
```


출력은 (한국어 또는 국제 날짜 형식에서) 다음과 같다.

```
1997.2.14
2008.2.14
```

레코드는 할당할 때도 그렇지만, 함수에 파라미터로 전달할 때도 똑같이 복사 작업이 수행된다. 위 코드에서 `MyDateToString` 에게 전달되는 레코드들 역시 새 복사본이다.

```
function MyDateToString(MyDate: TMyDate): string;
begin
    Result := MyDate.Year.ToString + '.' +
              MyDate.Month.ToString + '.' +
              MyDate.Day.ToString;
end;
```

위 함수는 호출될 때마다 항상 레코드 데이터 전체를 복사해서 사용한다. 복사 작업을 피하려면, 또는 원본 레코드를 변경할 수 있도록 하고 싶으면, 참조 파라미터 [reference parameter](#) 를 사용한다고 명시해야 한다. 그 방법은 아래와 같다.

```
procedure IncreaseYear(var MyDate: TMyDate);
begin
    Inc(MyDate.Year);
end;

var
    ADay: TMyDate;
begin
    ADay.Year := 2020;
    ADay.Month := 3;
    ADay.Day := 18;

    IncreaseYear(ADay);
    Show(MyDateToString(ADay));
```

위에서 `IncreaseYear` 를 호출하여 증가시킨 `Year` 필드의 값은 원본 레코드의 값이다. 따라서 이 레코드는 파라미터로 들어갈 때의 값보다 1년이 증가된 값을 가지게 된다.

```
2021.3.18
```

레코드 배열 사용 [Using Arrays of Records](#)

언급했듯이, 배열은 여러 번 반복되는 데이터 구조를 표현한다. 이와 달리, 레코드는 서로 다른 요소로 구성된 단일 구조를 표현한다. 서로 방향성이 다른 이 두 타입 생성자는 함께 사용되는 경우, 즉 레코드를 담은 배열이 매우 흔하게 사용된다 (배열을 담은 레코드 역시 가능하지만, 흔하지는 않다).

레코드 배열의 코드는 다른 배열과 비슷하다. 배열 요소마다 레코드가 들어가므로, 레코드 타입의 크기가 곧 배열 요소의 크기다. 보다 정교하게 [sophisticated](#) 요소들을 다루는 목록 [list](#) 들을 사용하는 방법으로 컬렉션 [collection](#) 또는 컨테이너 클래스 [container class](#)

등이 있고 그것들을 따로 뒤에서 살펴보겠지만, 레코드 배열을 가지고도 해낼 수 있는 것들이 데이터 관리 측면에서 매우 많다.

TMyDate 타입으로 된 배열을 예문으로 보자. 이 배열을 할당, 초기화, 사용하는 코드는 다음과 같다 (RecordsTest 예제에서 발췌함)

```
var
  DatesList: array of TMyDate;
  I: Integer;
begin
  // 배열 요소를 할당한다
  SetLength(DatesList, 5);

  // 무작위(Random) 값을 배열 요소에 할당한다
  for I := Low(DatesList) to High(DatesList) do
  begin
    DatesList[I].Year := 2000 + Random(50);
    DatesList[I].Month := 1 + Random(12);
    DatesList[I].Day := 1 + Random(27);
  end;

  // 배열 값을 표시한다
  for I := Low(DatesList) to High(DatesList) do
    Show(I.ToString + ': ' +
      MyDateToString(DatesList[I]));
```

무작위 [Random](#) 데이터를 사용하므로 출력은 매번 달라진다. 참고로, 내가 실행한 결과는 다음과 같다.

```
0: 2014.11.8
1: 2005.9.14
2: 2037.9.21
3: 2029.3.12
4: 2012.7.2
```

참고 배열 안의 레코드가 자동으로 초기화 될 수도 있다(단, 매니지드 레코드 [managed\(관리되는\) record](#)인 경우). 이 기능은 델파이 10.4 시드니에 도입되었다. 자세한 내용은 이 장의 뒷부분에서 다룬다.

배리언트 레코드 [Variant\(변형\) Records](#)

이 언어는 초기 버전부터, 레코드 타입 안에는 변형 [variant](#) 부분을 넣을 수 있었다. 즉, 필드 여러 개를 동일한 메모리 영역 하나에 매핑 [mapping](#) 되도록 할 수 있다. 데이터 타입이 서로 다른 필드들이라도 말이다. (이것은 C 언어에서 [union](#)에 해당한다.) 다른 한 편으로, 배리언트 [variant\(변형\)](#) 필드들 또는 배리언트 필드 그룹들을 사용하여 레코드 안에서 동일한 메모리 위치를 접근할 수 있다. 즉, 동일한 메모리 위치를 접근하지만, 각 필드 별 각자의 관점(데이터 타입)으로 값에 접근한다. 비슷하지만 다른 데이터를 저장하고, 타입 캐스팅 [casting/변환](#) 과 비슷한 효과를 얻을 때 주로 사용된다 (이 언어 초기, 즉 직접 타입 캐스팅이 허용되지 않던 시절에 사용됨). 배리언트 레코드 [Variant\(변형\)](#)

Record 타입 사용은 점차 객체-지향과 기타 현대식 기술에 의해 교체되었다. 그런데, 일부 시스템 라이브러리에서 특수한 경우에 여전히 내부적으로 사용된다.

배리언트 레코드 타입을 쓰면, 타입-안전 **type-safe** 이 지켜지지 않는다. 그리고 권장하는 프로그래밍 관행도 아니다. 특히 초보자에게는 더욱 그렇다. 어쨌든 오브젝트 파스칼 전문가가 되기 전까지는 이 기능을 다룰 필요가 없을 것이다. 따라서 이 책은 실제 샘플이나 자세한 설명을 제공하지 않는다. 정말로 힌트를 얻고 싶다면, "타입 변형 오픈 배열 파라미터" 절에 있는 데모에서 TVarRec 을 사용하는 코드를 보기 바란다.

필드 정렬 Fields Alignments

레코드와 관련된 또 다른 고급 주제는 레코드의 필드 정렬 방식이다. 이는 레코드의 실제 크기를 이해하는 데 도움이 된다. 라이브러리에 있는 레코드들을 살펴보면 **packed** 키워드가 붙어 있는 것을 자주 볼 수 있다. 이 키워드는 레코드에서 가능한 최소 바이트 수를 사용하라는 의미이다. 비록 그 결과로 인해 데이터에 접근하는 동작이 조금 더 느려지더라도 말이다.

필드 정렬에 의한 차이는 다양한 필드들을 전통적으로 16-비트 또는 32-비트로 정렬하는 것과 관련이 있다. 따라서 **byte** 바이트 하나는 그 뒤에 Integer 정수가 있다면 32 비트를 차지할 수도 있다. 비록 실제 사용되는 공간은 8 비트인데도 말이다. 뒤에 있는 Integer 를 32-비트를 경계로 접근하면 코드 실행 속도가 더 빠르기 때문이다.

참고 필드 크기와 필드 정렬은 그 타입의 크기에 따라 달라진다. 크기가 2의 거듭제곱 (즉 2^N)에 딱 맞지 않는 타입의 경우, 크기는 2의 거듭제곱 중 자신을 담을 수 있는 가장 작은 수이다. 예를 들어, Extended 타입은 10바이트를 사용하지만, 레코드 안에서는 16바이트를 차지한다 (packed를 사용하지 않는 레코드인 경우).

필드 정렬은 일반적으로 데이터 구조(레코드 등)에서 사용된다. 데이터 구조에 들어 있는 각 필드들을 접근하는 속도를 일부 CPU 아키텍처에서 더 높이기 위해서다. 필드 정렬을 바꾸려면, 컴파일러 지시어인 \$ALIGN 에 파라미터를 다르게 넣으면 된다.

{ \$ALIGN 1 }는 메모리 사용량을 절약한다. 즉 사용할 수 있는 모든 바이트를 활용한다. 이 설정은 레코드 정의에서 **packed** 지정자 specifier 를 붙였을 때와 같다. 이와 정반대 옵션인 { \$ALIGN 16 }은 가장 큰 정렬을 사용한다. 또다른 옵션으로는 4 와 8 이 있다.

예를 들어, 앞에 있던 RecordsTest 프로젝트로 가서 레코드 정의에 **packed** 키워드를 추가한다면 다음과 같다.

type

```
TMyDate = packed record
  Year: Integer;
  Month: Byte;
  Day: Byte;
end;
```


SizeOf 를 호출한 결과는 이제 8 이 아닌 6 이다.

더 수준 높은 예제를 보자. 오브젝트 파스칼 개발에 능숙하지 않다면 건너뛰어도 좋다. 아래와 같은 구조를 생각해보자 (AlignTest 예제에서 사용 가능).

```
type
  TMyRecord = record
    C: Byte;
    W: Word;
    B: Boolean;
    I: Integer;
    D: Double;
  end;
```

{\$ALIGN 1}을 쓰면 위 구조는 16 바이트를 사용한다 (SizeOf 에서 반환되는 값이다). 그리고 각 필드의 상대 [relative](#) 메모리 주소는 다음과 같다.

```
C: 0; W: 1; B: 3; I: 4; D: 8
```

참고 상대 주소 [relative address](#)를 계산하려면, 레코드를 할당한 다음, 레코드 구조체를 가리키는 포인터의 숫자 값을 기준으로 삼고, 해당 레코드 필드를 가리키는 포인터의 숫자 값과 차이가 얼마인지 계산한다. 예: `UIntPtr(@MyRec1.w) - UIntPtr(@MyRec1)`. 포인터 개념과 주소 연산자(@)에 대해서는 이 장의 뒷부분에서 다룬다.

이와 달리, 필드 정렬 즉 {\$ALIGN}을 4 로 변경하면(데이터 접근이 최적화될 수 있음), 사용되는 크기는 20 바이트이고, 상대 주소는 다음과 같다.

```
C: 0; W: 2; B: 4; I: 8; D: 12
```

극단적인 옵션을 적용해보자. {\$ALIGN 16}을 사용하면 이 구조체가 차지하는 용량은 24 바이트이고, 상대 주소는 다음과 같다.

```
C: 0; W: 2; B: 4; I: 8; D: 16
```

with 문은 어떨까? [What About the With Statement?](#)

레코드나 클래스를 쓸 때 전통적으로 사용되는 문장 중에는 with 문이 있다. With 는 파스칼 구문에만 있던 키워드인데, 나중에 자바스크립트와 비주얼 베이직에 도입되었다. 이 키워드는 코드를 적게 작성할 수 있어서 매우 편하지만, 매우 위험할 수도 있다. 코드를 읽기가 훨씬 어렵기 때문이다.

with 문에 대해서는 많은 논쟁이 있는데, 나는 With 문을 되도록 적게 사용해야 한다는 데 동의하는 편이다. 어쨌든, 이 책에서 With 문을 다루는 것은 중요하다고 생각해서 할애했다 (goto 문은 그렇지 않다).

참고 오브젝트 파스칼 언어에서 goto 문을 없애는 것이 타당한지에 대한 논쟁이 있었다. 또한 with 를 모바일 버전에서 제거할지에 대한 토의도 있었다. 합법적인 몇 가지 사용법이 있지만, with

문이 유발할 수 있는 범위^{scope} 문제를 고려한다면, 이 기능을 중단시킬 (또는 C#에서와 같이 별칭 이름이 필요하도록 바꿀) 이유가 충분하다.

with 문은 그저 문장 단축에 불과하다. record 타입 변수(또는 클래스의 오브젝트)를 참조하는 경우, 그 이름을 매번 반복하지 않으려면 with 문을 사용할 수 있다. 예를 들어, 아래 예문은 레코드 타입을 표현하는 코드다.

```
var
  ABirthday: TMyDate;
begin
  ABirthday.Year := 2008;
  ABirthday.Month := 2;
  ABirthday.Day := 14;
```

with 문을 사용하면, 뒷부분을 아래와 같이 작성해도 된다.

```
with ABirthday do
begin
  Year := 2008;
  Month := 2;
  Day := 14;
end;
```

이 접근 방식은 오브젝트 파스칼 프로그램에서 컴포넌트나 다른 클래스를 참조하는데 사용할 수 있다. 일반적으로 컴포넌트나 클래스로 작업할 때 with 문을 사용하면, 일부 코드를 생략할 수 있다. 특히 중첩된 데이터 구조에서 더 편하다.

그렇다면, 왜 with 문 사용을 권장하지 않는 것일까? 잡아내기 어려운 미묘한 에러가 생길 수 있기 때문이다. 예를 들면, 다른 변수 감추기 [variable masking](#) 등의 문제가 있다. 찾기 힘든 에러들을 여기에서 설명하기는 쉽지 않다. 그러니, 가벼운 상황을 살펴보자. 아마 머리를 굽적지게 될 수도 있을 것이다. 아래 예문은 레코드 타입을 선언한 다음 그 타입을 사용하고 있다.

```
type
  TMyRecord = record
    MyName: string;
    MyValue: Integer;
  end;

procedure TForm1.Button2Click(Sender: TObject);
var
  Record1: TMyRecord;
begin
  with Record1 do
    begin
      MyName := '홍길동';
      MyValue := 22;
    end;

    with Record1 do
      Show(Name + ': ' + MyValue.ToString);
```


틀린 코드가 없다! 컴파일도 되고, 실행도 된다. 하지만, 출력된 결과는 (적어도 대충 봤을 때) 예상한 것과 다르다.

Form1: 22

출력 결과 중 앞부분의 값은 우리가 지정한 레코드에 있는 값이 아니다. 두 번째 with 문에 적힌 Name 필드는 실수로 잘못 적은 것이기 때문이다. Name 필드는 앞에서 사용한 레코드 안에 없다. 전혀 다른 필드인데 우연히 이 범위 안에서 사용될 수 있었으므로 에러가 발생하지 않았을 뿐이다 (Button2Click 메서드는 폼 오브젝트 안에 있는 메서드다. 그런데, 폼에는 Name 이라는 필드가 있다).

만약 아래와 같이 작성했다면,

```
Show(Record1.Name + ': ' + Record1.MyValue.ToString);
```

컴파일러는 그 레코드 구조에 Name 필드가 없다고 에러 메시지로 알려주었을 것이다.

대체로, with 문은 현재의 범위 안에 새 식별자를 도입한다고 볼 수 있는데, 그로 인해 기존 식별자를 숨기게 되거나, 또는 같은 범위에 있는 다른 식별자에 잘못 접근할 수 있다. 이것이 바로 with 문을 사용하지 말아야 할 좋은 이유다. 더 나아가 with 문 여러 개를 사용한다면 상황은 더 나빠진다.

```
with MyRecord1, MyDate1 do...
```

위 코드 뒤에 나오는 코드를 읽기는 매우 힘들 것이다. 블록 안에 적힌 필드가 어느 레코드에 해당하는 필드인지를 알아 내야 하기 때문이다.

메서드를 가지는 레코드 Records with Methods

오브젝트 파스칼의 레코드는 최초 파스칼 언어의 레코드나 C 언어의 struct 구조체보다 더 강력하다. 실제로 레코드는, 클래스(뒤에서 설명한다)처럼, 메서드 즉 프로시저와 함수를 가질 수 있다. 심지어 언어 연산자들을 사용자 정의 방식으로 재정의할 수도 있다(연산자 오버로딩이라는 기능이다). 이 내용은 다음 절에서 살펴본다.

메서드들을 가지는 레코드는 클래스와 다소 비슷하다. 하지만, 가장 중요한 차이가 있다. 이 두 구조의 메모리가 관리되는 방식이 서로 다르다. 오브젝트 파스칼의 레코드는 현대 프로그래밍 언어들이 가지고 있는 아래의 두 가지 기본 특징들을 가지고 있다.

- **메서드 Method**: 해당 레코드 데이터 구조에 연결된 함수function와 프로시저procedure이다. 메서드들은 자신이 연결된 레코드 안의 필드에 직접 접근할 수 있다. 즉, 메서드란 레코드 타입 정의에 선언된(또는 포워드 선언forward declaration 된) 함수와 프로시저다.

- **캡슐화(Encapsulation)**: 데이터 구조의 일부 필드(또는 메서드)를 숨겨서 다른 코드에서는 직접 접근하지 못하게 하는 기능이다. 캡슐화를 실현하려면 접근 지정자([access specifier](#)) `private`를 사용한다. 이와 달리 외부에 노출하고 싶은 필드와 메서드에는 `public`을 붙인다. 레코드에서 기본([default](#)) 지정자는 `public`이다.

이제 확장 레코드에 대한 핵심 개념을 알았으니, 레코드를 정의하는 예문을 보자 (RecordMethods 예제에서 발췌함).

```
type
  TMyRecord = record
    private
      FName: string;
      FValue: Integer;
      FSomeChar: Char;
    public
      procedure Print;
      procedure SetValue(NewString: string);
      procedure Init(NewValue: Integer);
    end;
```

레코드 구조는 위와 같이 비공개 `private` 와 공개 `public` 두 부분으로 구분되어 있다. `private` 와 `public` 키워드는 얼마든지 반복해도 된다. 즉 구역을 여러 개로 나눌 수도 있다. 하지만, 이렇게 두 구역으로 명확하게 구분하면 코드를 읽을 때 확실히 도움이 된다. 메서드들은 레코드 정의 안에 (클래스 정의와 마찬가지로) 나열하는데, 구현 코드는 여기에 넣지 않는다. 즉, 레코드의 메서드들은 포워드 선언 ([forward declaration](#)) 이다.

메서드의 실제 코드는 어떻게 작성할까? 즉 구현 또는 정의를 완성하려면 어떻게 할까? 그 방식은 글로벌 함수나 글로벌 프로시저를 코딩하는 것과 거의 똑같다. 차이점은 메서드 이름을 적는 방식이다. 즉, 레코드 타입 이름과 메서드 이름을 모두 함께 적어야 한다. 그리고 나면, 그 구현 코드 블록 안에서는 해당 레코드에 속한 필드들이나 메서드들을 직접 참조할 수 있게 되므로 더 이상 레코드 이름을 앞에 붙이지 않아도 된다.

```
procedure TMyRecord.SetValue(NewString: string);
begin
  FName := NewString;
end;
```

팁 메서드 선언을 먼저 작성하고 다시 그 다음에 전체 정의를 작성하는 것이 지루하게 느껴질 수 있다. 하지만 IDE 편집기에서 Ctrl+Shift+C 조합을 누르면, 자동으로 코드가 만들어진다. 또한 Ctrl+Shift+위/아래 화살표 키를 누르면, 그 메서드 선언과 정의 사이를 바로 넘나들 수 있다.

이 레코드 타입에 선언된 다른 메서드들의 코드는 다음과 같다.

```
procedure TMyRecord.Init(NewValue: Integer);
begin
  FValue := NewValue;
  FSomeChar := 'A';
end;
```



```
function TMyRecord.ToString: string;
begin
    Result := FName + ' [' + FSomeChar + ']: ' + FValue.ToString;
end;
```

이 레코드를 사용하는 방법을 보여주는 코드 조각은 다음과 같다.

```
var
    MyRec: TMyRecord;
begin
    MyRec.Init(10);
    MyRec.SetValue('안녕하세요');
    Show(MyRec.ToString);
```

짐작한대로, 출력은 다음과 같다.

안녕하세요 [A]: 10

위 코드는 메서드를 사용하고 있다. 그럼 이제 필드를 사용하고 싶으면 어떻게 할까?

```
MyRec.FValue := 20;
```

위 코드는 실제로 컴파일이 된다. 그리고 작동도 한다. 아마 놀랐을 것이다. 앞에서 우리는 FValue 라는 필드를 private 구역 안에 선언했다. 즉 오직 메서드만 접근할 수 있도록 했을 뿐, 필드는 직접 접근을 허용하지 않았기 때문이다.

사실, 오브젝트 파스칼에서 private 접근 지정자는 다른 유닛에서만 효력이 있다. 따라서 위 코드가 만약 다른 유닛 파일에 들어 있다면 허용되지 않았을 것이다. 그런데, 이 타입이 선언된 유닛과 같은 유닛에 있는 코드는 접근을 제한하지 않는다. 이 점은 클래스에서도 마찬가지다.

Self: 레코드 뒤에 숨겨진 마법 Self: The Magic Behind Records

같은 레코드 타입으로 만들어진 2 개의 레코드 즉 MyRec1 과 MyRec2 가 있다고 가정해 보자. 메서드를 호출하여 그 코드를 실행하면, 그 메서드는 두 레코드 복사본 중 어떤 레코드로 작업해야 하는지 어떻게 알까? 메서드를 정의하면, 컴파일러는 파라미터 하나를 뒤에 숨겨서 추가하는데, 거기에는 메서드가 적용되는 레코드에 대한 참조가 담긴다.

즉, 위의 메서드 호출은 컴파일러에 의해 다음과 같이 변환된다.

```
// 우리가 작성한 코드
MyRec.SetValue('안녕하세요');
```

```
// 컴파일러가 생성하는 코드
SetValue(@MyRec, '안녕하세요');
```


위 의사 코드 `pseudo code` 안에 있는 `@`는 *address of* 연산자다. 레코드 인스턴스의 메모리 위치를 얻을 때 사용된다.

참고 주소 *address of* 연산자는 "포인터는 어떤가?"[What About Pointers?](#)라는 이 장의 마지막 절에서 간략히 다룰 것이다.

위와 같이 호출하는 코드가 변경된다고 해도, 호출되는 메서드 안에서는 실제로 이 숨겨진 파라미터를 어떻게 참조하고 사용할까? 암시적으로 `Self` 라는 특수 식별자를 사용한다. 따라서 위 메서드의 구현 코드를 다음과 같이 작성해도 사실 똑같다.

```
procedure TMyRecord.SetValue(NewString: string);
begin
    Self.FName := NewString;
end;
```

위 코드 역시 컴파일이 된다. 하지만, `Self` 를 명시적으로 적는 것은 거의 의미가 없다. `Self` 를 명시적으로 적어야 하는 경우는 레코드를 통째로 다른 함수에게 파라미터로 전달하는 경우 등이다. 이런 경우는 클래스에서 훨씬 더 자주 있다. 클래스에도 이와 정확히 똑같은 파라미터가 메서드에 숨겨져 있으며, 식별자 역시 동일하게 `Self` 다.

명시적으로 `Self` 파라미터를 사용하여 (필수가 아니더라도) 코드 가독성을 높일 수 있는 상황이 있다. 동일한 레코드 타입 두 개를 다루어야 하는 경우다. 예를 들어, 인스턴스의 값이 같은지 테스트할 때 이렇게 명시적으로 써주면 코드 읽기가 쉽다.

```
function TMyRecord.IsSameName(ARecord: TMyRecord): Boolean;
begin
    Result := Self.FName = ARecord.FName;
end;
```

참고 "숨겨진" `Self` 파라미터는 C++ 및 Java에서는 그 이름이 `this`이다. 하지만 Objective-C(와 오브젝트 파스칼에서는)에서는 `Self`라는 이름을 사용한다.

레코드 초기화하기 Initializing Records

레코드 타입(또는 레코드 인스턴스)의 변수를 글로벌 변수로 정의하면 해당 필드들이 초기화 `initialize` 된다. 하지만 스택에 변수를 정의하면 (함수나 프로시저의 로컬 변수로 정의하면) 초기화되지 않는다. 아래 코드를 보자 (`RecordMethods` 예제에서 발췌함).

```
var
    MyRec: TMyRecord;
begin
    Show(MyRec.ToString);
```

위 코드를 출력한 결과는 어느 정도 무작위 `random` 이다. `MyRec` 레코드의 필드 중 문자열 `String` 은 빈 `empty` 문자열로 초기화된다. 그런데, 문자 `Char` 필드와 정수 `Integer` 필드는 자신들이 차지한 메모리 위치에 때마침 들어있던 값을 받게 된다 (문자 변수나 정수

변수가 스택 `stack` 에서 영역을 차지할 때 생기는 일반적인 현상). 출력은 일반적으로 실제 컴파일 상황 또는 실행 환경에 의해 달라진다. 예를 들면 다음과 같이 출력된다.

```
| [□]: 1637580
```

그러므로, 레코드를 초기화 `initialize` 하는 것은 매우 중요하다 (변수 대부분이 그렇듯이) 사용하기 전에 초기화를 해야 비논리적인 데이터를 읽는 상황을 피할 수 있다. 심하면 애플리케이션이 깨지기도 한다. 이 상황을 해소하는 방법은 크게 두 가지다. 첫 번째는 레코드용 생성자 `constructor` 를 사용하는 것이다(곧이어 설명한다). 두 번째는 매니지드 레코드 `managed(관리되는) record` 를 사용하는 것이다 (델파이 10.4 에 도입된 기능인데, 이 장의 뒷부분에 가서 설명한다).

레코드와 생성자 Records and Constructors

일반 생성자 `regular constructor` 부터 시작해보자. 레코드는 생성자 `constructor` 라는 특별한 메서드 유형을 지원하기 때문에 우리는 레코드의 데이터를 초기화할 수 있다. 다른 메서드와 달리 생성자는 레코드 타입에 적용되며, 새 인스턴스 `instance` 를 정의할 때 사용된다. (또한, 이미 존재하는 인스턴스에도 여전히 적용할 수도 있다). 레코드에 생성자를 추가하는 방법은 다음과 같다.

```
type
TMyNewRecord = record
  public
    constructor Create(NewString: string);
    function ToString: string;
```

생성자는 일종의 메서드다. 따라서 코드를 넣을 수 있다. 예를 들면 아래와 같다.

```
constructor TMyNewRecord.Create(NewString: string);
begin
  Name := NewString;
  Init(0);
end;
```

생성자가 있으니 이제 레코드를 초기화할 수 있다. 아래 두 코딩 스타일 중 하나를 사용하면 된다.

```
var
MyRec, MyRec2: TMyNewRecord;
begin
  MyRec := TMyNewRecord.Create('내 자신');
  MyRec2.Create('내 자신');
```

레코드 생성자에는 반드시 파라미터가 있어야 한다는 점에 유의하자. 그냥 `Create` 만 선언하려고 하면 에러가 발생한다. 메시지는 "파라미터가 없는 생성자는 레코드 타입에서 허용되지 않습니다" 이다.

참고 Create 생성자를 오버로드 [overload](#)하거나, 또는 생성자 [constructor](#)의 이름을 다르게 하면, 생성자를 여러 개 가질 수 있다. 자세한 내용은 클래스용 생성자를 다룰 때 설명한다. 그런데, 곧 살펴보겠지만, 매니지드 레코드 [Managed record](#)는 이와 다른 구문 [syntax](#)을 사용하며, 파라미터 없는 생성자를 도입하지도 않는다. 매니지드 레코드는 Initialize라는 클래스 메서드 [class method](#)를 사용한다.

연산자에게 새 기반을 제공하기 [Operators Gain New Ground](#)

연산자 오버로드하기 [overloading](#) 역시 레코드와 관련된 오브젝트 파스칼 언어 기능이다. 즉 내가 만든 데이터 타입을 다루는 대한 표준 연산(덧셈, 곱셈, 비교 등)의 구현을 직접 맞춤 정의할 수 있다. 예를 들어, 더하기 연산자(특수한 Add 메서드)를 직접 구현하고 나서, 그 다음부터 + 기호를 사용하여 그 구현을 호출하도록 하는 개념이다. 연산자를 정의하려면 `class operator`를 사용한다 (두 키워드를 같이 붙여서 사용함).

참고 이미 존재하는 예약어 [reserved word](#)를 재사용함으로써, 언어 설계자들은 기존 코드에 아무런 영향을 주지 않을 수 있었다. 예약어 재사용은 키워드를 조합할 때 특히 자주 사용된다. 예를 들면, `strict private`, `class operator`, `class var` 등이 있다.

키워드 조합인 `class operator`에서 `class`라는 용어는 클래스 메서드 [class method](#)와 관련된 개념이다. 이 개념은 12장에서 설명하겠다. 이 키워드 조합으로 된 지시어 뒤에는 해당 연산자 [operator](#)의 이름을 적어준다 (예: Add).

```
type
  TPointRecord = record
public
  class operator Add(A, B: TPointRecord): TPointRecord;
```

이렇게 오버로드를 했으니, 이제 + 기호를 사용하여 더하기 연산자를 호출할 수 있다.

```
var
  A, B, C: TPointRecord;
begin
  C := A + B;
```

그렇다면 이렇게 사용할 수 있는 연산자로는 어떤 것들이 있을까? 기본적으로, 이 언어에 있는 연산자가 모두 해당된다. 하지만, 새 언어 연산자를 정의하지는 못한다.

- **캐스트** [cast\(변환\)](#) 연산자: Implicit, Explicit
- **단항** [Unary](#) 연산자: Positive, Negative, Inc, Dec, LogicalNot, BitwiseNot, Trunc, Round
- **비교** [Comparison](#) 연산자: Equal, NotEqual, GreaterThan, GraterThanOrEqual, LessThan, LessThenOrEqual
- **이항** [Binary](#) 연산자: Add, Subtract, Multiply, Divide, IntDivide, Modulus, ShiftLeft, ShiftRight, LogicalAnd, LogicalOr, LogicalXor, BitwiseAnd, BitwiseOr, BitwiseXor
- **매니지드 레코드** [Managed Record](#) 연산자: Initialize, Finalize, Assign (다음 절인 "연산자와 사용자 정의 매니지드 레코드"에서 설명한다. 델파이 10.4에서 추가됨)

연산자를 호출하는 코드에서는 이름을 사용하지 않고 해당 기호를 사용한다. 이 특별한 이름들은 정의 [definition](#) 안에서만 사용된다. 정의할 때 `class operator` 를 앞에 붙이기 때문에 이름 충돌을 피할 수 있다. 예를 들어, `Add` 메서드를 가진 레코드에 `Add` 연산자를 추가해도 이름이 충돌하는 문제가 없다.

연산자를 정의할 때는 파라미터를 명시한다. 그리고 그 연산자에 명시된 파라미터와 정확히 일치하게 “호출”을 해야 적용이 된다. 타입이 다른 두 값을 더하는 연산자를 정의하려면, `Add` 연산자를 두 개 명시해야 한다. 피연산자가 앞에 나올 수도 있고 뒤에 나올 수도 있을 텐데, 사실, 연산자 정의가 피연산자의 위치 전환까지 자동으로 해주지는 않기 때문이다. 게다가, 파라미터의 타입도 매우 정확하게 명시해야 한다. 자동으로 타입 변환 [type conversion](#) 이 반영되지 않기 때문이다. 따라서, 연산자의 오버로드 버전들을 여러 개 정의해야 하는 경우가 많다.

주목해야 할 또 다른 중요한 점이 있다. 데이터 변환 [data conversion](#) 을 위해 두 가지 특수 연산자인 `Implicit` 와 `Explicit` 를 정의할 수 있다. `Implicit` 는 암묵적 타입 캐스팅 [implicit type cast](#) (즉 조용한 변환 [silent conversion](#))을 정의할 때 사용된다. 이것은 완벽하고 손실이 전혀 없어야 한다. 그런데, `Explicit` 는 명시적 타입 캐스팅 [explicit type cast](#) 을 사용해 변수를 원래 타입에서 원하는 타입으로 변환하도록 직접 요청하는 경우에만 작동한다.

`Implicit` 와 `Explicit` 연산자는 둘 다 함수의 반환 타입 [return type](#) 을 기반으로 오버로드 하기가 가능하다. 메서드 오버로드에서 반환 타입은 일반적으로 고려 대상이 아니다. 하지만, 타입 캐스트 [cast\(변환\)](#) 의 경우에는, 실제로 컴파일러는 결과의 예상 타입을 알고, 타입캐스트 [typecast](#) 연산자들 중 어느 것을 적용할지 결정할 수 있다. 레코드에 연산자 몇 개를 정의한 예문을 보자 (`OperatorsOver` 예제에서 발췌함).

```
type
  TPointRecord = record
    private
      X, Y: Integer;
    public
      procedure SetValue(X1, Y1: Integer);
      class operator Add(A, B: TPointRecord): TPointRecord;
      class operator Explicit(A: TPointRecord): string;
      class operator Implicit(X1: Integer): TPointRecord;
  end;
```

위 레코드의 메서드들을 구현하는 코드는 아래와 같다.

```
class operator TPointRecord.Add(A, B: TPointRecord): TPointRecord;
begin
  Result.X := A.X + B.X;
  Result.Y := A.Y + B.Y;
end;

class operator TPointRecord.Explicit(A: TPointRecord): string;
begin
  Result := Format('%d:%d', [A.X, A.Y]);
end;
```



```
class operator TPointRecord.Implicit(X1: Integer): TPointRecord;
begin
    Result.X := X1;
    Result.Y := 10;
end;
```

이 레코드를 사용하는 방법은 매우 간단하다. 아래와 같이 코드를 작성하면 된다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    A, B, C: TPointRecord;
begin
    A.SetValue(10, 10);
    B := 30;
    C := A + B;
    Show(string(C));
end;
```

두 번째 할당(`B := 30;`)은 Implicit 캐스트 [cast](#) 연산자가 사용된다. 캐스트 코드가 없기 때문이다. 이와 달리, `Show` 호출은 캐스트 코드가 있기 때문에 명시적 [explicit](#) 타입 변환 [type conversion](#) 이 사용된다. 또한 `Add` 연산자는 파라미터로 전달되는 원본을 변경하는 것이 아니라 새 값을 반환한다는 점을 알아 두자.

참고 연산자가 새로운 값을 반환한다는 사실 때문에, 클래스를 위한 연산자 오버로드를 생각하는 건 더 어려워진다. 연산자가 임시 오브젝트를 새로 생성하면 그것을 폐기하는 것은 누구의 몫일까?

연산자 오버로드의 뒤편 [Operators Overloading Behind the Scenes](#)

이 내용은 짧지만 상당히 수준이 높다. 처음 읽을 때는 건너뛰어도 좋다.

잘 알려지지 않은 사실이지만, 기술적으로는 내부 이름 [fully qualified internal name](#) 앞에 `&`를 두 개 붙여서(예: `&&op_Addition`) 연산자 [operator](#) 를 호출할 수 있다. 이것은 연산자 기호를 사용하는 것과 효과가 같다. 예를 들어, 위 레코드의 더하기 연산자를 부를 때 아래와 같이 코딩할 수도 있다 (전체 목록은 데모에 들어 있다).

```
C := TPointRecord.&&op_Addition(A, B);
```

여러분이 코드를 위와 같이 작성하고 싶을 경우는 거의 없을 것이다. (연산자를 정의하는 목적은 오로지 더 친근한 표기법을 쓰기 위해서다. 일반 메서드 이름보다, 더 나아가 그보다도 추한 직접 호출보다 더 친근한 방식을 마련하기 위해서다)

피연산자 위치 전환을 구현하기 [Implementing Commutativity](#)

직접 정의한 레코드에 정수를 더하는 기능을 추가하고 싶다면, 연산자를 아래와 같이 정의하면 된다 (`OperatorsOver` 예제에서 발췌함. 레코드 타입이 살짝 다름).

```
class operator TPointRecord2.Add(A: TPointRecord2;
    B: Integer): TPointRecord2;
begin
    Result.X := A.X + B;
    Result.Y := A.Y + B;
end;
```


참고 이미 앞에서 만든 레코드 타입이 있는데도, 굳이 똑같은 구조로 새 타입을 만들어서 연산자를 정의한 이유가 있다. 앞에서 만든 레코드 타입은 이미 정수^{Integer}를 레코드 타입으로 암시적으로 변환하는 기능을 정의하고 있기 때문에 따로 더하기 연산자를 만들지 않아도 정수와 그 레코드 타입을 가지고 더하기 연산을 할 수 있기 때문이다. 이 문제는 곧 설명한다.

이제 위에 정의된 레코드에 정수 값(굵긴이: 원문에는 부동 소수점 값이라고 되어 있음)을 합법적으로 추가할 수 있다:

```
var
  A: TPointRecord2;
begin
  A.SetValue(10, 20);
  A := A + 10;
```

그런데, 더하기를 반대로 하는 아래 코드는 어떻게 될까?

```
A := 30 + A;
```

실패하고 에러가 표시된다.

```
[dcc32 Error] E2015 Operator not applicable to this operand type
(굵긴이: 이 피연산자 타입에 맞지 않는 연산자입니다)
```

언급했듯이, 피연산자의 타입이 서로 다른 경우, 자동으로 피연산자의 위치를 서로 바꿔주지 않는다. 따라서 반드시 구체적으로 구현해야 한다. 그 방법은 자리를 바꿔서 직접 호출하거나 또는 연산자의 다른 버전(아래 예문)을 호출하는 것이다.

```
class operator TPointRecord2.Add(B: Integer;
  A: TPointRecord2): TPointRecord2;
begin
  Result := A + B; //피연산자 위치 전환을 구현(Implementing Commutativity)
end;
```

암시적 캐스트와 타입 승격 ^{Implicit Cast and Type Promotion}

알아야 할 중요한 점이 있다. 연산자가 관여하는 호출을 풀어내는 규칙은 메서드가 관여하는 호출을 풀어내는 전통적인 ^{traditional} 규칙과 다르다. 연산자가 관여하는 호출은 자동으로 타입 승격 ^{type promotion} 이 된다. 따라서, 하나의 표현식이 결국 오버로드된 연산자에 해당되는 여러 가지 버전들을 호출하게 될 가능성이 있다. 즉 애매모호한 호출 ^{ambiguous call} 문제가 생길 수 있다. 따라서 Implicit 연산자를 작성할 때는 매우 주의해야 한다.

앞의 예제를 바탕으로 다음 표현식 ^{expression} 을 생각해 보자.

```
A := 50;
C := A + 30; // 첫 번째 경우
C := 50 + 30; // 두 번째 경우
C := 50 + TPointRecord(30); // 세 번째 경우
```


모두 합법적이다! 첫 번째 경우, 컴파일러는 30 을 알맞은 레코드 타입으로 변환한다. 두 번째 경우에는 할당 후에 변환이 이루어진다. 세 번째 경우에는 명시적 타입 변환 문장에 의해 변환된 레코드와 더하기를 해야 하기 때문에 그 앞의 정수 50 에는 암시적 타입 변환이 강제로 적용된다. 즉, 두 번째 경우만 다른 결과가 출력된다. 세 경우 각각의 출력 (X, Y 값)과 확장된 문장을 아래에서 보면 쉽게 이해할 수 있다.

```
// 출력
(80:20)
(80:10)
(80:20)

// 확장된 문장
C := A + TPointRecord(30);
// 즉, (50:10) + (30:10)

C := TPointRecord (50 + 30);
// 80을 변환하기 때문에 (80:10)이 된다.

C := TPointRecord(50) + TPointRecord(30);
// 즉, (50:10) + (30:10)
```

연산자와 사용자 정의 매니지드 레코드 Operators and Custom Managed Record

델파이 레코드에서 특별한 연산자 세트를 사용하면 사용자가 매니지드 레코드를 정의할 수 있다. 본론에 앞서, 레코드 메모리 초기화 [initialization](#) 규칙을 다시 정리하고, 일반 [plain](#) 레코드와 매니지드 [managed\(관리되는\)](#) 레코드의 차이점을 보자.

델파이의 레코드는 필드를 가질 수 있는데, 어떤 데이터 타입도 필드가 될 수 있다. 레코드 안에 일반 [plain](#) 필드(예: 숫자, 열거되는 값 [enum](#) 등)만 있으면 컴파일러는 별로 할 일이 없다. 레코드를 생성 [creating](#)하고 폐기 [disposing](#)하는 일은 그저 메모리 할당 [allocate](#)과 할당했던 메모리 영역 해제 [free](#)를 할 뿐이다. (기본 설정에서 [by default](#), 델파이는 레코드를 0 으로 초기화 [zero-initialize](#) 하지 않으니 주의하자. 참고로, 배열이나 새 오브젝트 인스턴스의 경우에는 델파이가 초기화 한다. 뒤에서 배우게 될 것이다).

레코드 안에 있는 필드가 만약 컴파일러에 의해 관리되는 타입이라면 (예: 문자열 [string](#), 인터페이스 [interface](#)), 컴파일러는 추가 코드를 넣는다. 초기화 [initialization](#)와 종료화 [finalization](#)를 관리하기 위해서다. 예를 들어, 문자열은 참조 카운트가 적용되는 타입이다. 그런데 문자열을 담고 있는 레코드가 생존 범위 [scope](#)를 벗어나면, 그 레코드 안에 있던 문자열에 대한 참조 카운트는 줄어야 한다. 그래야 그 문자열에게 할당되었던 메모리가 해제될 [deallocate](#) 수 있다. 다시 말해, 우리가 코드에서 매니지드 레코드를 사용하면, 컴파일러는 그 코드를 감싸는 [try-finally](#) 블록을 자동으로 추가한다. 그렇기 때문에, 예외 [exception](#)가 발생해도 레코드 안의 데이터가 메모리에 남지 않는다고 우리가 확신할 수 있다.

10.4 부터, 델파이 레코드 타입은 사용자 지정 [custom](#) 초기화와 종료화를 지원한다. 이 초기화와 종료화는 컴파일러가 매니지드 레코드를 위해 수행하는 기본 작업들에 더해 추가로 수행된다. 레코드의 필드에 사용되는 데이터 타입이 무엇이든 상관없이, 우리는 레코드를 선언하면서 사용자 지정 초기화와 종료화 코드를 레코드에 넣을 수 있다. 또한 초기화와 종료화의 구현 코드는 우리가 직접 작성한다. 이런 레코드를 "[사용자 정의 매니지드 레코드 Custom Managed Record](#)"라고 한다.

개발자가 레코드를 사용자 정의 매니지드 레코드로 바꾸려면, 10.4 에서 새로 추가된 아래 세 가지 연산자 중 하나를 (또는 그 이상을) 레코드 타입에 추가하면 된다.

- **Initialize** 연산자: 레코드를 위해 메모리가 할당된 후에 호출된다. 해당 필드들에 초기 값 [initial value](#) 을 설정하는 코드를 넣을 수 있다.
- **Finalize** 연산자: 레코드를 위해 할당되었던 메모리가 해제되기 전에 호출된다. 필요한 정리를 수행하는 코드를 넣을 수 있다.
- **Assign** 연산자: 레코드 데이터가 타입이 같은 다른 레코드로 복사되어 들어갈 때 호출된다. 한 레코드의 정보를 다른 레코드로 복사하는 작업을 특정 사용자 지정 방식으로 수행하도록 코드를 넣을 수 있다.

참고 예외 [exception](#) 가 발생해도 매니지드 레코드의 종료화 [finalization](#) 는 실행된다 (컴파일러가 자동으로 try-finally 블록을 만들어 넣기 때문임). 그래서, 리소스 할당 보호 또는 정리 작업을 구현하기 위한 대체 방법으로 자주 사용된다. 9장의 "매니지드 레코드를 사용하여 커서를 복원하기" 부분에서 사용 예제를 살펴보겠다.

초기화 연산자와 종료화 연산자를 가지는 레코드 [Records with Initialize and Finalize Operators](#)

초기화 연산자와 종료화 연산자를 넣어 레코드를 선언하는 간단한 코드부터 보자.

```
type
  TMyRecord = record
    Value: Integer;
    class operator Initialize(out Dest: TMyRecord);
    class operator Finalize(var Dest: TMyRecord);
  end;
```

물론 위 두 클래스 메서드 [class method](#) 를 정의하는 코드는 직접 작성해야 한다. 예를 들어, 메서드 실행을 로그 [log\(기록\)](#) 에 남기거나 또는 레코드의 값을 초기화할 수 있다. 아래 예문은 Value 필드를 초기화하고, 이 레코드를 담은 메모리 위치에 대한 참조를 로그로 남겨 어느 레코드가 이 동작을 수행하는지를 기록한다 (ManagedRecords_101 예제 프로젝트에서 발췌함).

```
class operator TMyRecord.Initialize(out Dest: TMyRecord);
begin
  Dest.Value := 10;
  Log('생성됨' + IntToHex(Integer(Pointer(@Dest))));
end;
```



```
class operator TMyRecord.Finalize(var Dest: TMyRecord);
begin
  Log( '파괴됨' + IntToHex(Integer(Pointer(@Dest)))));
end;
```

위 생성 [construction](#) 방식과 레코드에서 예전부터 제공되던 생성 방식의 차이는 자동 호출 [automatic invocation](#) 이다. 아래 예문처럼 코드를 작성하면, 초기화 코드와 종료화 코드가 모두 호출된다. 그리고 우리가 만든 매니지드 레코드 인스턴스에는 try-finally 블록이 자동으로 추가된다 (컴파일러가 알아서 반영함).

```
procedure LocalVarTest;
var
  My1: TMyRecord;
begin
  Log(My1.Value.ToString);
end;
```

위 코드를 실행하면 아래와 비슷한 로그를 얻는다 (주소는 다를 것이다).

```
생성됨 0019F2A8
10
파괴됨 0019F2A8
```

또 다른 상황은 인라인 변수 [inline variable](#) 를 사용할 때다. 아래 코드를 보자.

```
begin
  var T: TMyRecord;
  Log(T.Value.ToString);
```

위 코드를 실행하면, 앞에서 본 것과 똑같은 순서로 로그가 생성된다.

Assign 연산자 [The Assign Operator](#)

일반적으로, 할당 연산자(:=)는 레코드 필드의 모든 데이터를 그대로 [flatly](#) 복사한다. 매니지드 타입(예: 문자열)을 담고 있는 레코드도 컴파일러가 올바르게 처리한다.

여러분이 사용자 정의 [custom](#) 데이터 필드와 사용자 정의 초기화를 만들었다면, 아마도 기본 [default](#) 동작을 변경하고 싶었기 때문일 것이다. 이런 이유로, 사용자 정의 매니지드 레코드에서는 할당 연산자 [assignment operator](#) 를 우리가 직접 정의하는 것도 가능하다. 할당 연산자를 호출할 때는 := 구문 [syntax](#) 을 사용하지만, 정의할 때는, Assign 을 사용한다.

```
class operator Assign(var Dest: TMyRecord;
  const [ref] Src: TMyRecord);
```

이 연산자 정의는 규칙을 매우 정확하게 지켜야 한다. 즉, 첫 번째 파라미터는 참조로 전달 [pass by reference](#) 되는 파라미터(**var**)이고, 두 번째 파라미터는 참조로 전달 [pass by reference](#) 되는 **const** 파라미터이어야 한다. 그러지 않으면, 컴파일러는 다음과 같은 에러 메시지를 표시한다.


```
[dcc32 Error] E2617 First parameter of Assign operator must be a var
parameter of the container type
[dcc32 Hint] H2618 Second parameter of Assign operator must be a
const[Ref] or var parameter of the container type
```

Assign 연산자를 부르는 예문은 다음과 같다.

```
var
  My1, My2: TMyRecord;
begin
  My1.Value := 22;
  My2 := My1;
```

위 코드를 실행해 얻은 로그는 다음과 같다 (제공되는 데모에서는 각 레코드에 순번을 붙여주는데, 아래 로그에는 그것도 보여주고 있다).

```
생성됨 5 0019F2A0
생성됨 6 0019F298
5를 복사하여 6에 넣음
파괴됨 6 0019F298
파괴됨 5 0019F2A0
```

파괴 순서는 생성 순서와 반대라는 점을 눈여겨보자. 마지막으로 생성된 레코드가 가장 먼저 소멸된다.

매니지드 레코드를 파라미터로 전달하기 Passing Managed Records as Parameters

파라미터로 전달되거나 함수에 의해 반환되는 경우에도, 매니지드 레코드의 동작은 일반 레코드와 다를 수 있다. 아래 루틴 [routine](#) 들은 이런 다양한 상황을 보여준다.

```
procedure ParByValue(Rec: TMyRecord);
procedure ParByConstValue(const Rec: TMyRecord);
procedure ParByRef(var Rec: TMyRecord);
procedure ParByConstRef(const [ref] Rec: TMyRecord);
function ParReturned: TMyRecord;
```

일일이 살펴볼 필요 없이 (ManagedRecords_101 데모를 실행하면 확인할 수 있다), 정보를 요약하면 다음과 같다.

- ParByValue 메서드는, 새 레코드를 생성하고 (사용할 수 있다면) 할당 연산자를 불러 데이터를 복사하여 넣는다. 메서드가 범위를 벗어나면 이 복사본은 삭제된다.
- ParByConstValue 메서드는, 복사도 하지 않고 호출도 하지 않는다.
- ParByRef 메서드는, 복사도 하지 않고 호출도 하지 않는다.
- ParByConstRef 메서드는, 복사도 하지 않고 호출도 하지 않는다.
- ParReturned 메서드는, (초기화를 통해) 새 레코드를 생성한다. 생성된 새 레코드는 이 메서드 안에서만 유지된다. `my1 := ParReturned` 와 같이 할당하는 코드가 있다면, Assign 연산자를 호출하여 반환한다. 생성됐던 새 임시 레코드는 삭제된다.

예외와 매니지드 레코드 Exceptions and Managed Records

일반적으로 레코드는 예외 `exception` 가 발생하는 경우에 깨끗이 지워진다. `try-finally` 블록을 명시적으로 적지 않아도 제거된다는 점은 매니지드 레코드가 오브젝트와 근본적으로 다른 점이며, 이것이 정말 유용하게 사용되는 주된 이유다.

```
procedure ExceptionTest;
begin
  var A: TMRE;
  var B: TMRE;
  raise Exception.Create('에러 메시지');
end;
```

이 프로시저는 생성자를 두 번 호출하고, 소멸자도 두 번 호출한다. 다시 말하지만, 이것이 매니지드 레코드의 근본적인 차별점이자 핵심 기능이다.

매니지드 레코드들을 담는 배열 Arrays of Managed Records

매니지드 레코드 `managed record` 들을 담는 정적 배열 `static array` 을 정의하면, 선언 시점에 안에 들어가는 레코드들이 초기화된다. 그 시점에 `initialize` 연산자가 호출되기 때문이다.

```
var
  A1: array[1..5] of TMyRecord; // 초기화(Initialization) 코드가 호출됨
begin
  Log('배열에 레코드 담기');
```

배열이 범위를 벗어나면 그 안의 레코드들은 모두 소멸된다. 매니지드 레코드들을 담는 동적 배열 `dynamic array` 을 정의하는 경우에는, 초기화 코드가 호출되는 시점이 동적 배열의 크기를 지정하는 (`SetLength` 함수) 호출을 하는 시점이다.

```
var
  A2: array of TMyRecord;
begin
  Log('동적 배열에 레코드 담기');
  SetLength(A2, 5); // 초기화(Initialization) 코드가 호출되는 지점
```

배리언트 Variants(변형)

오브젝트 파스칼에 느슨한 타입 지정이라는 개념이 생겨서 `Variant` 라는 네이티브 타입이 생긴 것은 원래 윈도우의 OLE 와 COM 을 완전하게 지원하기 위해서였다. 배리언트 `Variant` 라는 이름 때문에 배리언트 레코드 `variant record` 가 연상되고, 오픈 배열 파라미터 `open array parameter` 와 구현 상 공통점이 조금 있긴 해도, `Variant` 는 완전히 별개의 기능이며, 구현이 매우 독특하다 (윈도우 개발 세상 바깥에는 거의 없는 언어 구현).

여기에서, OLE 에 대해서 그리고 이 데이터 타입이 쓰이는 상황(예: 데이터 셋 `dataset` 에 접근하기 위한 필드)에 대해서 언급하지 않겠다. 지금은 그저 이 데이터 타입을 일반적인 관점에서만 설명한다.

16장에 가면 동적 타입 `dynamic type`, RTTI, 리플렉션 `reflection`을 다루는 다시 보게 될 것이다. 거기에서는 이와 관련된 (하지만, 타입-안전성이 보장되고 `type-safe` 훨씬 더 빠른) RTL 데이터 타입인 `TValue`에 대해서도 다룰 것이다.

배리언트는 타입을 갖지 않는다 Variants Have No Type

일반적으로, 변수가 배리언트 타입이면, 기본 `basic` 데이터 타입이면 무엇이든 그 변수에 저장하고, 다양한 연산과 타입 변환을 수행할 수 있다. 자동 타입 변환 `type conversion`은 오브젝트 파스칼 언어의 일반적인 접근인 타입-안전 `type-safe`이라는 가치를 거스른다. 오히려 동적 타입 지정 `dynamic typing`에 가까운 구현이다. 이런 구현은 Smalltalk, Objective-C 등에서 처음 시작되어, 요즘 널리 사용되는 스크립트 언어인 JavaScript, PHP, Python, Ruby에서 볼 수 있다.

배리언트는 타입 확인과 계산이 실행 중에 처리된다. 컴파일러는 코드에서 발생될 수 있는 에러를 경고하지 않는다. 따라서 광범위하게 테스트를 해야 찾을 수 있다. 크게 보면, 배리언트를 사용하는 코드 부분은 인터프린트 되는 코드 `interpreted code`라고 간주할 수 있다. 실행되기 전에는 연산들을 풀어낼 수 없기 때문에 속도에 영향을 끼친다는 점에서 인터프린트 되는 코드와 마찬가지로 때문이다.

이제 Variant 타입을 사용하는 것에 대해 경구를 했으니, 이것을 가지고 무엇을 할 수 있는지 살펴보자. 배리언트 변수를 선언하는 기본부터 보자.

```
var
  V: Variant;
```

위 변수에는 여러 가지 타입의 값을 할당할 수 있다.

```
V := 10;
V := '안녕하세요';
V := 45.55;
```

배리언트 타입인 변수에 들어간 값은 호환이 되는 안되든 상관없이 어떤 타입에도 복사해 넣을 수 있다. 호환되지 않는 타입에게 할당한 경우, 컴파일러는 에러라고 알려주지 않으며, 변환은 실행 시간에 수행된다. 그 변환이 합당하지 않으면 실행 중 에러 `runtime error`가 발생한다. 기술적으로, 배리언트에는 실제 데이터와 함께 타입 정보가 저장되므로, 편리하지만 느리고 안전하지 않은 몇 가지 런타임 작업을 수행할 수 있다.

아래 코드를 보자(VariantTest 예제에서 발췌함). 위 코드를 조금 확장한 것이다.

```
var
  V: Variant;
  S: string;
begin
  V := 10;
  S := V;
  V := V + S;
```



```
Show(V);

V := '안녕하세요';
V := V + S;
Show(V);

V := 45.55;
V := V + S;
Show(V);
```

웃기지 않은가? 놀랍지 않게도, 결과는 다음과 같다

```
20
안녕하세요10
55.55
```

문자열 `string` 을 담은 배리언트를 변수 `s` 에 할당한 것은 그렇다 치자. 그런데, 변수 `s` 에 정수나 부동 소수점 숫자를 담은 배리언트도 할당할 수 있다. 게다가, 이 배리언트를 사용해서 값을 연산할 수 있다. `V := V + S` 연산은 다양하게 해석되는데, 그 해석은 배리언트에 들어간 데이터에 따라 달라진다. 위 코드를 보면, 같은 연산 표현이지만, 정수 더하기, 부동 소수점을 더하기, 그리고 문자열 이어 붙이기를 수행하고 있다.

표현식 `expression` 을 작성할 때 배리언트를 넣는 것은 위험하다. 문자열에 숫자 하나가 들어있을 때는 모든 것들이 작동하지만, 그렇지 않을 때는 예외가 생길 수 있다. 꼭 사용해야만 하는 이유를 댈 수 없다면, `Variant` 타입을 사용하면 안 된다. 오브젝트 파스칼의 표준 데이터 타입 그리고 타입-점검 접근 방식을 고수하라.

배리언트를 자세히 보기 Variants in Depth

배리언트를 더 자세히 이해하고 싶은 사람들을 위해, 배리언트의 작동 방식과 배리언트를 더 잘 제어할 수 있는 방법에 대한 몇 가지 기술적 정보를 추가하겠다. RTL 에는 `TVarData` 라는 배리언트 레코드 타입이 있다. 이 타입은 `Variant` 타입과 메모리 배치 `layout` 가 똑같다. `TVarData` 를 사용하면 배리언트의 실제 타입을 접근할 수 있다. `TVarData` 구조 안에는 그 배리언트의 타입을 담은 `VType` 을 포함해 예약된 필드들 몇 개와 실제 값이 들어 있다. 또한 여기에는 `null` 값이라는 개념이 있어서 `NULL(nil 이 아님)` 을 써서 할당할 수 있다는 점을 알아 두자.

참고 더 자세한 내용은 System 유닛의 해당 RTL 소스 코드 안에 있는 `TVarData` 정의를 참조하면 된다. 단순한 구조와는 거리가 멀기 때문에 어느 정도 경험이 있는 개발자들만 배리언트 타입의 구현 세부 사항을 살펴볼 것을 권한다.

`VType` 필드에 사용할 수 있는 값들은 OLE 자동화에서 사용할 수 있는 데이터 타입에 해당한다. 그리고 종종 OLE 타입 또는 배리언트 타입이라고 부른다. 해당되는 타입 전체를 알파벳 순서로 나열하면 다음과 같다.

varAny	varArray	varBoolean	varByte
varByRef	varCurrency	varDate	varDispatch
varDouble	varEmpty	varError	varInt64
varInteger	varLongWord	varNull	varOleStr
varRecord	varShortInt	varSingle	varSmallint
varString	varTypeMask	varUInt64	varUnknown
varUString	varVariant	varWord	

위 목록에 있는 배리언트 타입의 상수 이름은 대부분 쉽게 이해할 수 있다.

배리언트를 연산할 수 있는 함수들 역시 많다. 특정 타입으로 변환하거나 배리언트의 타입 정보를 요청하는데 사용할 수 있는 것들이다(예: `VarType` 함수). 이런 타입 변환 함수와 할당 함수들은 실제로 배리언트를 사용하는 표현식을 작성하면 자동으로 호출된다. 배리언트를 지원하는 다른 루틴 [routine](#) 들은 배리언트 배열 [variant array](#) 을 다루는 것들이다. 다시 말하지만, 윈도우에서 OLE 를 통합할 때에만 거의 사용된다.

배리언트는 느리다 [Variants Are Slow](#)

Variant 타입을 사용하는 코드는 느리다. 데이터 타입을 변환할 때만 느린 게 아니라, 정수를 담은 배리언트 두 개를 더하는 것도 느리다. 거의 인터프리터 되는 코드만큼 느리다. 속도를 비교하기 위해 Variant 를 기반으로 한 코드와 Integer 를 기반으로 한 코드로 똑같은 알고리즘을 작성해보았다. `VariantTest` 예제 프로젝트에서 두 번째 버튼에 적용된 코드를 보면 된다.

이 프로그램은 루프 [loop](#) 를 실행하고 그 속도를 측정하고 진행률 표시에 상태를 표시한다. 두 루프는 코드가 거의 똑같다. 그러니 `Int64` 를 기반으로 하는 코드는 생략하고 배리언트를 기반으로 하는 코드만 보자.

```
const
  MaxNo = 10_000_000; // 1 천만
var
  Time1, Time2: TDateTime;
  N1, N2: Variant;
begin
  Time1 := Now;
  N1 := 0;
  N2 := 0;

  while N1 < MaxNo do
  begin
    Inc(N2, N1);
    Inc(N1);
  end;

  // 우리는 반드시 그 결과를 사용해야 한다.
  Time2 := Now;
  Show(N2);
  Show('배리언트: ' + FormatDateTime(
    'ss.zzz', Time2-Time1) + ' 초');
```


위에 있는 시간을 재는 코드는 살펴볼 가치가 있다. 모든 종류의 성능 테스트에 쉽게 적용할 수 있기 때문이다. 보다시피, 이 프로그램은 `Now` 함수를 사용하여 현재 시간을 가져오고, `FormatDateTime` 함수를 사용하여 시간 차이를 초("ss")와 밀리초("zzz")로만 표시한다. 실제로 이 예제는 속도 차이가 너무 커서 시간 측정을 정확히 하지 않아도 알 수 있다. 내가 윈도우 가상 머신에서 얻은 수치는 다음과 같다.

```
49999995000000 배리언트: 01.169초
49999995000000 정수 : 00.026초
```

내 가상 머신에서는, 배리언트 기반 코드가 약 50 배 더 느리다! 프로그램이 실행되는 장비에 따라 실제 측정 값은 다르지만 비교 결과는 크게 다르지 않다. 안드로이드 휴대폰에서도 그 비율이 비슷하다. 하지만, 전체 실행 시간은 훨씬 더 오래 걸린다.

```
49999995000000 배리언트: 07.717초
49999995000000 정수 : 00.157초
```

내 휴대폰은 윈도우보다 전체를 실행하는데 시간이 6 배 더 오래 걸렸다. 하지만, 7 초 이상 차이가 났으니, 사용자에게 배리언트-기반 구현은 눈에 띄게 느릴 것이다. 이와 달리 `Int64`-기반 구현은 휴대폰에서도 여전히 매우 빠르다(사용자는 1/10 초 차이를 거의 느끼지 못할 것이다).

포인터는 어떤가? [What About Pointers?](#)

오브젝트 파스칼 언어에서 기반이 되는 또 다른 데이터 타입은 포인터 [pointer](#) 이다. 몇몇 객체-지향 언어들은 이 강력하지만 위험한 언어 구조를 숨기기 위해 많은 노력을 기울여 왔다. 하지만, 오브젝트 파스칼은 필요할 때 프로그래머가 포인터를 사용할 수 있도록 한다 (일반적으로는 자주 사용되지 않는다).

그렇다면 포인터란 무엇일까? 그리고 그 이름은 어디에서 왔을까? 대부분의 다른 데이터 타입들과 달리, 포인터는 실제 값을 담지 않는다. 실제 값을 담고 있는 변수가 있다고 할 때, 포인터는 그 변수에 접근할 수 있는 간접적인 참조를 담는다. 이를 좀 더 기술적으로 표현하자면, 포인터 타입인 변수에는 다른 변수의 메모리 주소가 들어간다. 이때 그 다른 변수의 타입은 해당 포인터에 지정된 [given](#) 데이터 타입 (또는 정의되지 않은 [undefined](#) 타입)일 수 있다.

참고 수준이 꽤 높은 내용이지만 이 책에 담았다. 기본 [basic](#) 내용에 해당하는 주제가 아니지만 개발자라면 누구나 알고 있어야 할 핵심 지식에 해당되기 때문이다. 이 언어를 처음 접했다면, 처음 읽을 때는 이 부분을 건너뛰고 싶을 수도 있다. 게다가, (명시적인) 포인터가 없는 프로그래밍 언어를 사용해 본 개발자라면 이 짧은 내용을 흥미롭게 읽을 수 있을 것이다.

포인터 타입을 정의할 때는 특정 키워드가 아니라 특수 기호인 캐럿(^)을 사용한다. 예를 들어, 다음 예문은 Integer 타입 변수를 가리키는 포인터 타입을 정의하고 있다.

```
type
  TPointerToInt = ^Integer;
```

포인터 변수를 위와 같이 정의하였으니, 이제 할당할 수 있다. 지정된 타입으로 된 다른 변수의 주소를 @ 연산자를 사용해서 할당한다. (PointersTest 예제에서 발췌함).

```
var
  P: ^Integer;
  X: Integer;
begin
  X := 10;
  P := @X;
  // 포인터를 이용하여 X의 값을 변경한다
  P^ := 20;
  Show('X: ' + X.ToString);
  Show('P^: ' + P^.ToString);
  Show('P: ' + UIntPtr(P).ToHexString(8));
```

위 코드에서 포인터 P는 변수 X를 참조한다. 따라서, P^를 사용하여 그 변수의 값을 참조하고, 읽고, 변경할 수 있다. 포인터에 담긴 값(즉 변수 X가 담긴 메모리 주소)을 그대로 표현할 수도 있다. 그러려면 포인터를 숫자로 타입 캐스트 [cast\(변환\)](#) 해야 한다. 위에서는 특별한 타입인 UIntPtr을 사용했다(자세한 내용은 아래 참고에서 설명한다). 그 결과, 위 코드는 일반 숫자 값이 아니라 16 진수 값을 표현한다. 메모리 주소는 일반적으로 16 진수로 표현한다. 출력되는 결과는 다음과 같다 (포인터에 담긴 메모리 주소는 컴파일러에 따라 달라질 수 있다).

```
X: 20
P^: 20
P: 0018FC18
```

경고 포인터를 Integer로 캐스트 [cast](#)하는 것은 32-비트 플랫폼에서만 올바르다. 더 큰 메모리 공간을 사용하게 설정된 경우에는 Cardinal 타입을 써야 한다. 64-비트 플랫폼이라면, NativeUInt를 사용하는 것이 더 좋다. 그런데, 이 타입은 별칭 [alias](#)이 있다. 포인터를 의미하는 특별한 이름인 UIntPtr가 바로 그것이다. 이 별칭을 사용하는 것이 이 상황에서 가장 좋다. 코드의 의도를 컴파일러와 개발자 모두에게 명확하게 표시할 수 있기 때문이다.

요약해서 명확히 설명하자면, 포인터 P가 있을 때,

- 포인터를 직접 사용하면(P를 사용하면) 포인터가 가리키는 메모리 위치의 주소를 참조하게 된다.
- 포인터를 역참조 [dereference](#)하면(P^를 사용하면) 그 메모리 위치에 담긴 실제 내용을 참조하게 된다.

이미 정해진 메모리 위치를 참조하는 위와 같은 상황이 아니라, 새로운 특정 메모리 블록을 동적으로 힙 [heap](#)에 할당 받고 그 위치를 포인터를 통해 참조하고자 한다면, `New` 프로시저를 사용한다. 이렇게 한 경우라면, 포인터를 통해 접근하던 값이 더 이상 필요하지 않게 되었을 때에는 동적으로 할당 받은 그 메모리 영역을 제거해 주어야 한다. 그 방법은 `Dispose`를 호출하는 것이다.

참고 일반적인 메모리 관리와 힙 [heap](#)의 작동 방식은 13장에서 다룬다. 간단히 말해, 힙은 (큰) 메모리 영역이며, 정해진 순서 없이 메모리 블록을 할당하고 해제할 수 있다. `New`와 `Dispose`의 대안으로, 우리는 `GetMem`과 `FreeMem`을 사용할 수 있는데, 그러려면 할당할 크기를 직접 명시해야 한다 (이와 달리, `New`와 `Dispose`의 경우에는 컴파일러가 할당 크기를 자동적으로 결정한다). 컴파일 시점에 할당 크기를 알 수 없는 경우에 `GetMem`과 `FreeMem`이 유용하다.

메모리를 동적으로 할당하는 코드 조각을 보자.

```
var
  P: ^Integer;
begin
  // 초기화(Initialization)
  New(P);
  // 연산(Operations)
  P^ := 20;
  Show(P^.ToString);
  // 종료(Termination)
  Dispose(P);
```

메모리를 사용한 후 폐기 [dispose](#)하지 않으면, 결국 사용 가능한 메모리를 모두 소모하게 되므로 프로그램이 깨질 수 있다. 더 이상 필요하지 않은 메모리를 해제하지 못하는 것을 이른바 메모리 누수 [memory leak](#)라고 한다.

경고 안전한 코드가 되려면, 위 코드는 사실 `try-finally` 블록을 사용해야 한다. 이 주제는 지금 소개하지 않고, 9장에서 다룰 것이다.

포인터에 넣을 값이 없으면, `nil` 값을 할당할 수 있다. 포인터가 가리키고 있는 값이 있는지 확인하려면 포인터가 `nil` 인지 아닌지를 테스트하면 된다. 그 방법은 동일성 여부를 직접 비교하거나 또는 `Assigned` 함수를 사용한다 (아래 코드 참조).

이러한 종류의 테스트는 자주 사용된다. 유효하지 않은 포인터를 역참조 [dereference](#) 하면 메모리 접근 위반 [access violation](#)이 발생하기 때문이다 (운영체제에 따라 효과가 조금씩 다름).

```
var
  P: ^Integer;
begin
  P := nil;
  Show(P^.ToString);
```


위 코드의 효과는 PointersTest 예제를 실행하면 볼 수 있다. (윈도우에서) 표시되는 에러는 다음과 비슷할 것이다.

```
Access violation at address 0080B14E in module 'PointersTest.exe'.
Read of address 00000000.
```

포인터 데이터를 더 안전하게 접근하는 방법 중 하나는 "포인터가 null 이 아님"을 확인하는 안전 검사를 다음과 같이 넣는 것이다.

```
if P <> nil then
  Show(P^.ToString);
```

앞서 언급했듯이, 다른 대안이 있다. 읽기가 더 쉽기 때문에 일반적으로 선호되는 방법인데, 바로 의사 함수인 Assigned 를 사용하는 것이다. 코드는 아래와 같다.

```
if Assigned(P) then
  WriteLn(P^.ToString);
```

참고 Assigned는 진짜 함수가 아니다. 컴파일러에 의해 알맞은 코드로 "해소"되기 때문이다. 또한, Assigned는 함수 타입 *procedural type* 변수 (즉, 메서드에 대한 참조)에도 사용할 수 있다. 이 경우에는 실제로 그 함수를 불러내지 않고, 그저 할당이 되었는지 여부만 확인한다.

오브젝트 파스칼에는 Pointer 데이터 타입도 있다. 이것은 타입이 지정되지 않은 포인터 (예: C 언어의 void*)이다. 타입이 지정되지 않은 포인터를 사용하는 경우에는 New 대신 GetMem 을 사용하고 할당할 바이트의 개수를 명시해야 한다. 그 크기 값은 타입이 있어야 추론 *infer* 할 수 있는데 타입을 모르기 때문이다. 메모리 변수에 할당할 크기가 정의되지 않는 경우에는 매번 GetMem 프로시저가 필요하다.

오브젝트 파스칼에서 포인터를 쓸 일이 거의 없다는 사실은 흥미로운 장점이다. 하지만, 그럼에도 불구하고 이 기능 사용이 허용되기 때문에, 매우 효율적인 저수준 함수를 구현하거나 운영체제의 API 를 호출할 때 도움이 된다. 어쨌든, 포인터를 이해하는 것은 수준 높은 프로그래밍을 하는데 중요하다. 또한 델파이의 오브젝트 모델을 완전히 이해하는데도 필요하다. 델파이 모델의 이면에서 포인터 (일반적으로 참조라고 부른다)가 사용되기 때문이다.

경고 변수에 포인터가 담겨있고 그 포인터는 다른 변수를 가리키고 있는데, 그 다른 변수가 범위를 벗어나거나 해제 *free* 된다면, 그 포인터가 가리키는 메모리 공간에는 정의되지 않음 *undefined*이거나, 또는 뭔가 다른 데이터가 담겨 있게 된다. 이로 인한 버그는 찾기가 매우 어려울 수 있다.

파일 타입을 알고 있나요? File Types, Anyone?

오브젝트 파스칼에서 소개할 마지막 데이터 타입 생성자는 파일 *file* 타입이다. 파일 타입은 물리적인 디스크 파일을 나타낸다. 이 타입은 파스칼 언어의 독특한 기능이다.

오래된 언어이든 현대식 언어이든 이 타입을 원시 **primitive** 데이터 타입으로 가지고 있는 언어는 거의 없다. 오브젝트 파스칼 언어에는 `file` 키워드가 있다. 이것은 `array` 또는 `record` 와 같은 타입 지정자 **type specifier** 이다. `file` 키워드를 사용하면 새 타입을 정의할 수 있고 그리고 나서 그 새 타입을 사용하여 새 변수를 선언할 수 있다.

```
type
  IntFile = file of Integers;
var
  IntFile1: IntFile;
```

`file` 키워드를 사용할 때 데이터 타입을 명시하지 않는 것도 가능하다. 그러면 타입이 지정되지 않은 파일을 명시하는 것이다. 또는 `TextFile` 타입을 사용할 수도 있다. 이 타입은 RTL **런타임 라이브러리**의 `System` 유닛에 정의되어 있으며, ASCII 문자가 담기는 파일(또는 요즘에는 바이트 **Byte**가 담기는 파일)을 선언할 때 사용된다.

여전히 파일을 직접 사용하는 것이 지원되기는 하지만, 요즘에는 점점 덜 사용되고 있다. RTL 안에는 바이너리 파일과 텍스트 파일을 훨씬 높은 수준에서 관리할 수 있는 클래스들이 많이 들어 있어 있기 때문이다 (예: 유니코드로 인코딩 되는 텍스트 파일 지원 등).

델파이 애플리케이션은 일반적으로 RTL 스트림(`TStream` 과 그것의 파생 클래스)을 사용하여 복잡한 파일 읽기와 쓰기 동작을 다룬다. 스트림 **Stream** 은 가상 **virtual** 파일을 나타낸다. 이것은 물리적인 파일, 메모리 블록, 메모리 블록, 소켓, 기타 이처럼 계속 이어지는 바이트들을 매핑할 수 있다.

`file` 루틴이 여전히 사용되는 영역 중 하나는 콘솔 애플리케이션을 작성할 때다. 거기에서는 `Write`, `WriteLn`, `Read`, 그리고 기타 관련 함수를 사용하여 특수한 타입의 파일, 표준 입력과 표준 출력을 작업할 수 있다(C 와 C++는 콘솔의 입력 및 출력을 이와 유사하게 지원한다. 다른 많은 언어들 역시 비슷한 서비스들이 있다).

06: 문자열String에 관한 모든 것

문자열String은 모든 프로그래밍 언어에서 가장 일반적으로 사용되는 데이터 타입이다. 오브젝트 파스칼은 문자열 처리가 상당히 간단하다. 그러면서도 매우 빠르고 매우 강력하다. 우리는 문자열의 기본 정도는 쉽게 이해할 수 있고 이미 앞에서도 문자열을 사용했다. 하지만, 문자열의 뒤편은 언뜻 봤을 때 보다 조금 더 복잡하다. 텍스트Text를 잘 다루려면, 문자열과 관련된 여러 가지 주제를 알아 둘 필요가 있다. 문자열 처리를 완전히 이해하려면, 유니코드Unicode 표현에 대해 알아야 하고, 문자열이 문자 배열Array of Character에 어떻게 매핑 되는지를 이해해야 하며, 델파이의 런타임 라이브러리Run Time Library 중에서 가장 많이 활용되는 문자열 처리 방법(문자열을 텍스트 파일로 저장하기 또는 읽어오기 등)에 대해 배워야 한다.

오브젝트 파스칼에서, 문자열을 다룰 때는 여러 가지 방식들이 있다. 사용할 수 있는 데이터 타입과 방식도 여러 가지다. 이 장은 표준 문자열 데이터 타입을 중심으로 설명한다. 하지만, 델파이 컴파일러에서 여전히 사용할 수 있는 더 오래된 문자열들(AnsiString 등) 역시 잠시 살펴본다. 문자열을 살펴보기에 앞서, 처음부터 차근차근 나가보자. 유니코드 표현부터 시작한다.

유니코드Unicode: 전 세계를 위한 문자 집합

오브젝트 파스칼 문자열String 관리는 유니코드Unicode 문자 세트character set, 특히 UTF-16이라는 유니코드Unicode 표현representation을 중심으로 돌아간다. 문자열이 기술적으로 어떻게 구현되는지 자세히 들여다보기 전에, 유니코드 표준을 완전히 이해할 수 있게 몇 가지 주제를 살펴보자.

유니코드는 ‘전 세계 모든 문자 집합들과 그 집합들 안에 있는 모든 문자마다 각자의 고유한 설명과 시각적 표현, 그리고 고유한 숫자 값, 즉 유니코드Unicode 코드 포인트code point를 가진다’라는 기본 개념을 바탕으로 한다 (유니코드가 단순하면서도 동시에 복잡한 이유다)

참고 유니코드 컨소시엄 웹 사이트인 <https://www.unicode.org>에는 문서가 풍부하게 있다. 총정리 자료는 "The Unicode Standard" 책이다. <https://www.unicode.org/book/aboutbook.html>에서 이 책을 찾을 수 있다.

개발자 모두가 유니코드에 익숙한 건 아니다. 많은 개발자들이 아직도 ASCII 등 오래되고 제한적인 표현이나 ISO 인코딩의 관점에서 문자를 생각한다. 그러니 과거 표준들을 간략히 보면서 유니코드의 특징(과 복잡성)을 더 잘 이해해보자.

문자의 과거: ASCII부터 ISO 인코딩Encoding까지

문자 표현Character representation은 미국 표준 정보 교환 코드(ASCII American Standard Code for Information Interchange) 즉, 60년대 초에 개발된 컴퓨터 문자 표준 인코딩에서 시작되었다. ASCII에는 영어 알파벳 26 자의 소문자와 대문자, 숫자 10 개, 많이 쓰이는 문장 부호, 제어 문자 여러 개가 포함되었다(오늘날에도 여전히 사용되고 있다).

ASCII는 7비트 인코딩 시스템을 사용하여 서로 다른 문자 128개를 표현한다. 그 중에서 시각적으로 표현될 수 있는 문자는 #32(공백)에서 #126(물결표시) 사이에 있는 문자들이다 (아래 그림 6.1 참고, 오브젝트 파스칼로 만든 애플리케이션을 윈도우에서 실행하고 그 화면을 발췌한 화면임).

그림 6.1:

인쇄 가능한
ASCII 문자 세트를
보여주는 표

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	␣

ASCII가 문자 표현의 기반이 되었다는 점은 분명하다(ASCII 기본 세트에 있는 128자는 유니코드에서도 여전히 핵심 부분이다) 하지만, 얼마 못 가서 곧 8번째 비트를 사용하여 이 기본 세트에 추가로 128자를 포함시키는 확장 버전들에 의해 밀려났다.

이제 문제가 생긴다. 전 세계에 언어가 너무 많은데 무슨 문자를 추가하여 이 확장 세트(때로는 ASCII-8라고 부름)를 만들 것인지를 판단할 간단한 방법을 찾기가 어렵다. 그 결과를 요약하면, 윈도우는 코드 페이지code page라는 서로 다른 문자 세트를 채택했다. 그리고, 사용자의 로캘locale / 지역 설정 구성과 윈도우 버전에 따라 서로 다른 코드 페이지를 적용했다. 윈도우 코드 페이지 이외에도 이런 페이지 구성 방식의 접근을 기반으로 하는 비슷한 표준들이 많은데, 이 페이지들은 국제 ISO 표준의 한 부분으로 들어갔다.

가장 많이 활용되는 것은 ISO 8859 표준이다. 이 표준에 정의된 여러 가지 지역 세트들 중에서는 가장 많이 사용된 것은 ISO 8859-1라고 하는 라틴어 세트다 (글쎄, 좀 더 정확하게 말하자면 서구 국가 대부분에서 널리 사용되는 세트다).

참고 윈도우 1252 코드 페이지는 ISO 8859-1세트와 부분적으로 비슷하지만 완전히 준수하지는 않는다. 윈도우는 추가 문자로 € 기호, 추가 따옴표 등을 128에서 150까지의 영역에 추가해 놓았는데, 윈도우의 확장판에 있는 이런 문자들은 라틴어 세트에 있는 다른 모든 값들과는 달리 유니코드 코드 포인트를 준수하지 않는다.

유니코드 코드 포인트Code Point들과 시각적 문자Grapheme들

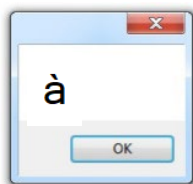
설명을 보다 정확하게 하려면, 코드 포인트Code Point라는 개념 외에 한 가지 개념을 더 언급해야 한다. ‘시각적 문자Grapheme’ 하나를 표현하기 위해 사용되는 코드 포인트는 대체로 하나이지만, 실제로, 가끔은 코드 포인트 여러 개가 사용되어야 하는 경우가 있다. 일반적으로 이는 글자letter 하나가 아니라 글자letter들의 조합 또는 글자letter들과 기호symbol들의 조합인 경우에 그렇다. 예를 들어 라틴 문자 a를 나타내는 코드 포인트(#\$0061) 바로 뒤에 악센트를 나타내는 코드 포인트(#\$0300)가 있으면, 악센트 표시가 달린 문자 하나가 표현된다.

오브젝트 파스칼 코드는 아래와 같다(CodePoints 예제의 일부다). 이 코드가 실행되면 악센트가 있는 문자 하나가 메시지 상자에 나타난다(그림 6.2 참조).

```
var
  Str: String;
begin
  Str := #$0061 + #$0300;
  ShowMessage(Str);
```

그림 6.2:

시각적 문자Grapheme 하나를 표현하기 위해 코드 포인트 여러 개가 필요할 수도 있다



위 코드는 문자 즉 코드 포인트를 두 개를 사용한다. 하지만, 화면에 나타나는 시각적 문자 `grapheme`는 오직 하나다. 실제로 라틴 문자 집합을 사용한다면 위 코드처럼, 코드 포인트 두 개를 쓰지 않아도 표현할 수 있다. 이 문자가 라틴 문자 집합 안에 있기 때문이다. 즉, 그 문자의 유니코드 코드 포인트 (라틴 문자 집합에서, 악센트 강세가 있는 `a`의 코드 포인트는 `$00E0`이다) 하나만으로 표현할 수 있다. 하지만, 라틴 문자가 아닌 다른 문자 집합을 사용한다면, 이 문자(와 올바른 출력)를 얻는 유일한 방법은 위 코드와 같이 유니코드 코드 포인트 두 개를 결합하는 것이 뿐이다.

비록, 표현된 것은 악센트가 있는 문자 하나이지만, (오직 표현만 그럴 뿐) 이 표현에 사용된 두 값(코드 포인트)까지 자동으로 변환되지는 않고 그대로 있다. 따라서, 값 자체는 단일 문자 `à`와 같지 않다. (문자 두 개가 이어진) 문자열이다.

참고 시각적 문자-`grapheme`를 렌더링할 때 코드 포인트 여러 개를 사용하는 것은 운영체제의 지원 능력 그리고 사용되는 지정된 텍스트 렌더링 기술에 따라 달라질 수 있다. 따라서 일부 시각적 문자들은, 어떠한 운영체제에서 올바르게 출력된다고 보장할 수 없다.

코드 포인트에서 바이트(UTF)로 변환하기

ASCII는 문자와 그 문자에 해당하는 숫자를 직접 매핑하는 쉬운 방식을 썼다. 하지만, 유니코드가 접근하는 방식은 더 복잡하다. 이미 언급했듯이, 유니코드 문자 집합 안의 모든 요소는 저마다 고유한 코드 포인트가 지정되어 있다. 하지만 각 코드 포인트와 실제로 그것이 가리키는 값을 서로 매핑하는 방식은 종종 더 복잡하다.

유니코드를 이해하기 어려운 이유는, 코드 포인트(즉 유니코드 문자별로 지정된 숫자 값)가 메모리 또는 파일에 저장되려면 물리적인 실제 바이트`byte`로 표현되어야 하는 데, 동일한 코드 포인트라도 물리적 표현 방법은 여러 가지이기 때문이다.

모든 문자를 4 바이트`byte`로 표현한다면, 간단한 단 하나의 규칙으로 모든 유니코드 코드 포인트를 표현할 수 있다. 고정 길이 (즉 모든 문자가 항상 같은 양의 바이트를 사용하는) 방식으로는 이 방법이 유일하다. 하지만 개발자 대부분은 이 방식이 메모리와 처리 시간 면에서 지나치게 비싸다고 생각할 것이다.

참고 오브젝트 파스칼에서 4 바이트 유니코드 코드 포인트를 표현하려면 UCS4Char 데이터 타입을 사용하면 된다.

그래서 유니코드 표준에는 메모리를 대체로 더 적게 사용하는 다른 표현법들이 정의되어 있다. 하지만 각 심볼`symbol`에서 사용되는 바이트 수가 일정하기 않다. 기본적으로, 가장 공통된 요소일수록 더 짧은 표현을, 사용 빈도가 드문 요소일수록 더 긴 표현을 사용하도록 하겠다는 생각이 반영되었다.

이처럼 유니코드 코드 포인트를 물리적으로 표현하는 서로 다른 방식들을 유니코드 변환 형식(또는 UTF`Unicode Transformation Format`)이라고 한다. 이 형식들은 유니코드 표준에

들어있는데, 실제로 *매핑* 알고리즘이다. 이 알고리즘들은 (각 문자마다 배정된 절대 숫자 값인) 코드 포인트를 고유한 바이트^{byte} 나열로 변환한다. 그리고, 이런 매핑은 양방향으로 사용할 수 있어서, 다른 표현으로 서로 변환하는 것이 가능하다.

표준에 정의된 변환 형식은 세 가지다. 즉 8, 16, 32가 있다. 이 숫자는 유니코드 문자 세트의 앞부분에 위치한 문자들(즉, 맨 앞에 있는 128자)의 코드 페이지를 표현하는데 드는 비트^{bit}의 개수이다. 재미있게도 이 세 형식 모두 최대 4 바이트까지 사용한다.

- **UTF-8**은 문자를 1~4바이트 가변-길이^{variable-length} 인코딩으로 변환한다. UTF-8은 HTML 및 이와 유사한 프로토콜에 널리 사용되는데, 그 이유는 사용하는 문자 대부분이 (HTML 태그처럼) ASCII의 하위 세트에 안에 해당하는 경우, 인코딩 된 결과가 상당히 작기 때문이다.
- **UTF-16**은 (윈도우^{Windows}, macOS 등) 많은 운영체제에서 널리 사용된다. UTF-16은 문자 대부분이 2바이트로 표현된다. 따라서 적당히 작으면서도 빠른 속도로 처리할 수 있어서 상당히 편리하다.
- **UTF-32**는 (모든 코드 포인트를 표현하는 길이가 동일하므로) 처리하기에 적합하지만 메모리 소모가 크기 때문에 실제로 사용되는 경우는 많지 않다.

UTF-16이 ‘모든’ 코드 포인트를 2 바이트로 직접 매핑할 수 있다고 오해하는 경우가 많다. 하지만, 유니코드에 정의된 코드 포인트가 10만 개가 넘는다는 사실을 생각한다면, 2 바이트 즉 64K 개 요소를 가지고 모든 코드 포인트와 1:1로 매핑하는 것이 불가능하다는 것을 쉽게 알 수 있다. 하지만, 개발자는 유니코드의 일부 하위 집합만 사용함으로써 사용되는 모든 문자를 ‘문자 당 2바이트로 표현’이라는 고정 길이에 맞추려는 경우가 있는데, 초창기에는 이 유니코드의 하위 집합을 UCS-2라고 불렀고, 지금은 BMP(다국어 기본 평면 ^{Basic Multilingual Plane})라고 부른다. 어쨌든 이것은 유니코드의 하위 집합일 뿐이며 유니코드에 있는 *여러 플레인* 중 하나일 뿐이다.

참고 멀티-바이트 표현(UTF-16 및 UTF-32)은 그 여러 바이트 중 어느 것이 앞에 나오는지를 결정해야 한다는 문제가 있다. 표준에 따르면 모든 형태가 허용된다. 따라서, UTF-16 BE(빅-엔디언 ^{big-endian}) 또는 LE(리틀-엔디언 ^{little-endian})를 사용할 수 있다. UTF-32도 마찬가지다. 빅-엔디언 바이트 나열^{serialization}에서는 자릿수가 가장 큰 바이트가 먼저이고, 리틀-엔디언 바이트 나열^{serialization}에서는 자릿수가 가장 작은 바이트가 먼저이다. 바이트 나열 형태는 해당 UTF 표현과 함께 파일 안에 표시가 되는 경우가 많은데, BOM(바이트 순서 표시 ^{Byte Order Mark})이라는 헤더를 통해 표시된다.

바이트 순서 표시 (BOM) ^{Byte Order Mark}

유니코드 문자가 저장된 텍스트 파일이 있다면, 그 파일은 무슨 UTF 형식을 사용했는지 코드 포인트가 인코딩 되었는지를 알아 낼 방법이 있다. 파일 시작 부분에 있는 헤더 또는 마커에 이 정보가 저장된다. 바이트 순서 표시(BOM) ^{Byte Order Mark}이라고 부른다. 이것은 사용된 유니코드 변환 형식 그리고 (리틀-엔디언 또는 빅-엔디언 즉 LE 또는

BE 등) 바이트 나열 형태를 기록한 서명^{signature}이다. 아래 표에는 다양한 BOM에 대한 요약이 나와 있는데, 각각 2 또는 3 또는 4바이트 길이로 기록할 수 있다.

00 00 FE FF	UTF-32, BE
FF FE 00 00	UTF-32, LE
FE FF	UTF-16, BE
FF FE	UTF-16, LE
EF BB BF	UTF-8

우리는 이 장의 뒷부분에서 오브젝트 파스칼이 스트리밍^{streaming} 클래스 안에서 BOM을 어떻게 관리하지를 볼 것이다. BOM은 파일의 맨 앞에 나타나며 그 바로 뒤부터 실제 유니코드 데이터가 들어간다. 따라서 AB라는 텍스트가 들어 있는 UTF-8 파일에는 아래와 같이 16진수 값 5개(BOM에 3개, 문자에 2개)가 기록된다.

EF BB BF 41 42

텍스트 파일에 이러한 서명이 없는 경우 일반적으로 ASCII 텍스트 파일로 간주된다. 하지만, 사실, 무슨 텍스트가 어떤 인코딩으로 처리되었든, 얼마든지 파일 안에 기록할 수 있다.

참고 반면에, 웹 요청^{request} 또는 기타 인터넷 프로토콜을 통해 데이터를 수신하는 경우에는 BOM에 의존하지 않고 인코딩을 나타내는 특정 헤더(웹 프로토콜의 일부임)를 활용한다.

유니코드 살펴보기

앞에서 본 ASCII 문자 표처럼, 표 형태로 유니코드 문자를 나열하는 프로그램은 어떻게 만들까? 써로게이트 쌍 ^{surrogate pairs/대리 쌍}은 일단 빼고, BMP(다국어 기본 평면 ^{Basic Multilingual Plane}) 안에 있는 코드 포인트를 표시하는 것부터 시작해보자.

참고 UTF-16으로 인코딩 되었다고 해서 그 안에 있는 숫자 값들 모두가 진짜 UTF-16 코드 포인트인 것은 아니다. (써로게이트^{surrogate/대리자} 라는) 유효하지 않은 숫자 값, 즉 여럿이 짝을 지어서, 코드 포인트가 65,535 보다 큰 숫자에 배정된 문자를 표현하는 값들이 있을 수 있다. 예를 들어, 낮은음자리표를 표현하려면 써로게이트 쌍^{surrogate pairs/대리 쌍}을 사용한다. 낮은음자리표의 유니코드 코드 포인트는 U+1D122이므로, 3 바이트에 해당되는데, UTF-16로는 4 바이트 즉 2 바이트 값 두 개(D834와 DD22)를 짝지어 연결하여 이 코드 포인트를 표현한다.

BMP의 모든 요소를 한 화면에 표시하려면 256 * 256 모눈이 필요하다. 너무 크기 때문에 ShowUnicode 예제는 탭으로 나누어 두 페이지로 구성했다. 첫번째 탭에는 전체 요소를 256개로 나눈 묶음을 목록으로 나열하고, 두번째 페이지에는 각 묶음에 해당하는 문자를 16 * 16 모눈 안에 표시한다. BMP가 어떻게 구성되어 있는 지만 궁금하고 그 내부에는 관심이 없다면, 코드를 훑어보기만 해도 된다.

프로그램을 실행하면, TabControl의 첫 페이지에 있는 ListView 컨트롤에 256개 묶음이 나열된다. 각 묶음은 자신이 가진 문자 256개 중 맨 앞과 맨 뒤의 문자를 보여준다. 아래 코드는 폼의 OnCreate 이벤트 [event](#) 핸들러 [handler](#) 코드와 각 문자를 표시하는데 사용되는 간단한 함수다. 실행된 화면은 그림 6.3과 같다.

```
// 도우미(Helper) 함수
function GetCharDescr(NChar: Integer): string;
begin
    if Char(NChar).IsControl then
        Result := 'Char #' + IntToStr(NChar) + ' [ ]'
    else
        Result := 'Char #' + IntToStr(NChar) + ' [' + Char(NChar) + ']';
end;

procedure TForm2.FormCreate(Sender: TObject);
var
    I: Integer;
    ListItem: TListItem;
begin
    for I := 0 to 255 do // 256 페이지 * 각 256 문자
    begin
        ListItem := ListView1.Items.Add;
        ListItem.Tag := I;
        if (I < 216) or (I > 223) then
            ListItem.Text :=
                GetCharDescr(I * 256) + '/' + GetCharDescr(I * 256 + 255)
        else
            ListItem.Text := 'Surrogate Code Points (대리자 코드 포인트)';
        end;
    end;
end;
```

그림 6.3:

ShowUnicode 예제의
첫 페이지에는 유니코드
문자들의 묶음 목록이
길게 나열된다



아래에서 “페이지”의 번호를 ListView 안에 있는 각 항목의 Tag 프로퍼티에 저장하는 코드를 잘 보자. 사용자가 항목 중 하나를 선택하면, TabControl의 두 번째 페이지로 이동한다. 이때 저장된 정보를 사용해 그 영역에 해당하는 256개 문자를 StringGrid에 채워서 보여준다.

```
procedure TForm2.ListView1ItemClick(const Sender: TObject;
  const AItem: TListItem);
var
  I, NStart: Integer;
begin
  NStart := AItem.Tag * 256;
  for I := 0 to 255 do
    begin
      StringGrid1.Cells[I mod 16, I div 16] :=
        IfThen(not Char(I + NStart).IsControl, Char(I + NStart), '');
    end;
  TabControl1.ActiveTab := TabItem2;
```

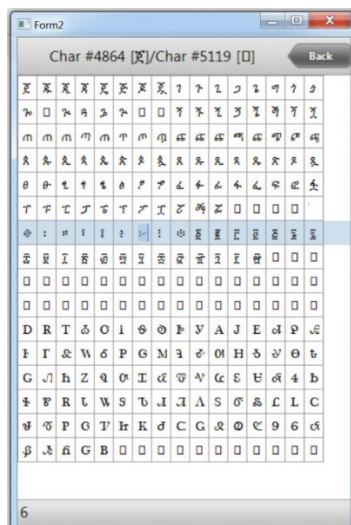
위 코드에 IfThen 함수가 있다. IfThen 함수는 첫번째 파라미터에 있는 조건이 참이면 두번째 파라미터의 값을 반환하고, 그렇지 않으면 세번째 파라미터의 값을 반환한다. 위 조건 테스트에서는 Char 타입용 헬퍼 클래스의 메서드인 IsControl를 사용하여 화면에서 눈으로 볼 수 없는 제어 문자들을 걸러서 제외하고 있다.

참고 오브젝트 파스칼의 IfThen 함수는 C 기반 프로그래밍 언어 대부분이 사용하는 ?: 연산자와 동작이 거의 비슷하다. IfThen 함수는 문자열String용 버전과 정수Integer용 버전이 따로 있다. 문자열용 버전을 쓰려면 System.StrUtils 유닛을, 정수용 버전을 쓰려면 System.Math 유닛을 uses 문 안에 넣어주어야 한다.

이 애플리케이션이 출력하는 유니코드 문자 표는 그림 6.4와 같다. 단, 출력되는 결과는 무슨 글꼴이 선택되었는지 그리고 운영체제가 무엇인지에 따라 달라진다는 점을 알아 두자.

그림 6.4:

ShowUnicode 예제의
두 번째 페이지에는
실제 유니코드 문자가
표 안에 표시된다



Char^{문자} 타입을 다시 살펴보기

유니코드에 대한 소개가 끝났으니, 본 주제로 돌아가서 오브젝트 파스칼 언어에서 문자^{character}와 문자열^{string}을 어떻게 다루는지 알아보자. 2장에서는 이미 Char 데이터 타입을 소개했고, Char 타입용 헬퍼^{helper} 함수들이 System.Character 유닛 안에 들어 있다는 점을 언급했었다. 이제 유니코드에 대해 더 잘 이해했을 테니, 2장에서 있던 내용으로 돌아가서 세부 사항을 더 자세히 살펴보자.

가장 먼저, Char 타입이라고 해서 무조건 유니코드 코드 포인트를 나타내고 있다고 보면 안 된다. Char 데이터 타입 요소의 크기는 사실 2 바이트이다. Char 타입이 유니코드의 BMP(다국어 기본 평면 ^{Basic Multilingual Plane}) 안에 들어 있는 문자의 코드 포인트를 표현한다는 점은 맞지만, Char 하나는 쌍로게이트 쌍 ^{surrogate pairs} / 대리 쌍의 한 부분에 불과할 수도 있다.

기술적으로는, 쌍로게이트 쌍을 전혀 사용하지 않아도 모든 유니코드 코드 포인트를 직접 표현하는 것이 가능하다. 오브젝트 파스칼에는 요소 당 4 바이트^{byte}를 사용하는 UCS4Char 타입이라는 문자 타입이 따로 있기 때문이다. 하지만, UCS4Char 타입이 실제로 사용되는 경우는 거의 없는데, 그렇게 많은 메모리를 사용하면서까지 써야 할 이유를 인정받기 어렵기 때문이다. 하지만, 곧 보게 되겠지만 System.Character 유닛 안에는 UCS4Char 타입용 연산들도 몇 가지 들어 있다.

Char 타입으로 다시 돌아가보자. Char 타입은 요소의 순서가 정해진 타입 즉, Ordinal^[오디널] 타입 중 하나라는 점을 기억하자. 그러므로 시퀀스^{sequence/순번} 개념이 있으며, 코드에서 Ord, Inc, Dec, High, Low와 같은 연산을 할 수 있다. Char 타입용 헬퍼^{helper} 등 Char를 다루는 확장 연산들은 대부분 RTL 기본 유닛인 System 유닛이 아니라 System.Character 유닛 안에 들어 있다. 따라서, 확장 연산을 사용하려면 uses 문에 System.Character 유닛을 넣어주어야 한다.

Character 유닛을 사용하여 유니코드를 연산하기

유니코드 문자^{character}(와 물론 유니코드 문자열^{string}도 포함)를 다루는 연산은 대부분 System.Character 라는 유닛에 정의되어 있다. 이 유닛에는 TCharHelper라는 클래스 헬퍼가 정의되어 있으며, Char 타입 변수에서 해당 연산들을 직접 사용할 수 있다.

참고 System.Character 유닛에는 TCharacter라는 레코드^{record}도 정의되어 있다. TCharacter는 기본적으로 정적 클래스 함수들의 집합체이며, 이 함수들은 마찬가지로 System.Character 유닛 안에 정의되어 있는 글로벌 루틴 ^{global routine/전역 루틴}들과 매핑 된다. 이 함수들은 모두 더이상 유지보수가 되지 않기 때문에, 유니코드에서 Char 타입을 다루려면 TCharHelper를 사용하는 것이 더 바람직하다.

System.Character 유닛에는 흥미로운 열거^{Enum} 타입 두 개도 정의되어 있다. 하나는 TUnicodeCategory다. 이 타입에는 유니코드 문자들을 구분하는 다양한 카테고리들이

나열되어 있는데, 예를 들어 컨트롤, 공백, 대문자, 소문자, 소수점, 구두점, 수학 기호 등이 여기에 해당된다. 또 한가지 열거 타입은 TUnicodeBreak다. 여기에는 다양한 공백space들(공백의 종류는 한 가지가 아니다), 하이픈hyphen들, 제어 표시들에 대한 모뎀family이 정의되어 있다. 이는 ASCII 연산에 익숙한 개발자에게 큰 변화다. 우선, 유니코드에서는 숫자가 0에서 9 사이의 문자로만 국한되지 않는다. 또한 공백 역시 문자 #32 만으로 국한되지 않는다. 그 외에도 256개 요소로 구성된 (훨씬 더 간단한) 문자 집합에서 적용하던 가정assumption들과는 다른 점들이 많다.

Char 타입용 헬퍼Helper에 있는 메서드Method는 40 가지가 넘는다. 따라서, 다양한 조건 테스트와 연산을 할 수 있다. 예를 들면,

- 문자의 숫자 표현을 가져온다(GetNumericValue).
- 유니코드 문자의 카테고리가 무엇인지를 알아내거나(GetUnicodeCategory), 문자가 어느 특정 카테고리에 해당되는 지를 확인한다(IsLetterOrDigit, IsLetter, IsDigit, IsNumber, IsControl, IsWhiteSpace, IsPunctuation, IsSymbol, IsSeparator). 앞서 데모에서는 IsControl 연산을 사용했었다.
- 문자가 소문자인지 대문자인지 확인하거나(IsLower, IsUpper), 대소문자를 변환한다(ToLower, ToUpper).
- 문자가 UTF-16 씨로게이트 쌍의 일부인지 확인한다(IsSurrogate, IsLowSurrogate, IsHighSurrogate). 그리고 씨로게이트 쌍을 여러 가지 방법을 통해 변환한다.
- 문자를 UTF-32로 변환하거나(ConvertFromUtf32, ConvertToUtf32), UCS4Char 타입으로 변환한다(ToUCS4Char).
- 주어진 문자 리스트list 안에 문자가 포함되어 있는지 확인한다(IsInArray).

알아 둘 점이 있다. 이 연산들 중에는 특정 변수에는 적용하지 못하고 타입에만 적용할 수 있는 것들이 있다. 그런 경우에는 Char 타입을 접두사로 붙여서 호출해야 한다. 아래의 두 번째 예문에 있는 코드에서 볼 수 있다.

유니코드 문자에 대한 연산을 조금 실험할 수 있게 CharTest라는 예제를 만들었다. 데모의 예시 중에서 유니코드 요소를 대상으로 대문자 연산과 소문자 연산을 호출하는 코드를 살펴보자. RTL 안에 오래 전부터 있던 UpCase 함수는 ANSI 표현에 해당하는 26개 영문자만 대문자로 변환할 수 있다. 유니코드 문자들 중에는 대문자 표현을 가진 것들이 더 있는데도, 그것들을 처리하지 못한다 (사실 대소문자 개념이 보편적이라고 볼 수는 없다. 대소문자 표현이 모든 문자 집합에 있는 건 아니기 때문이다).

다음은 악센트가 있는 문자를 대문자로 변환하는 코드다(CharTest 예제에서 발췌함).

```
var
  Ch1: Char;
begin
  Ch1 := 'ù';
  Show( 'UpCase ù: ' + UpCase(Ch1));
  Show( 'ToUpper ù: ' + Ch1.ToUpper);
```


아래는 위 코드가 실행된 결과다. 오래된 UpCase 함수는 악센트가 있는 라틴 문자를 대문자로 변환하지 못하지만, ToUpper 함수는 대문자로 올바르게 변환한다.

```
UpCase ù: ù
ToUpper ù: Ù
```

Char 타입용 헬퍼 [helper](#) 안에는 유니코드 관련 함수들이 많다. 아래는 그 중 몇 개다. (역시 CharTest 예제에 있음). 아래 코드는 문자열 [String](#) 변수 하나를 정의하고 문자들을 담는다. 그 중 한 문자는 BMP(유니코드 코드 포인트의 첫 64K) 밖에 있다. 이어서, 문자열에 담긴 요소들을 대상으로 다양한 조건을 테스트한다. 그 결과는 모두 True다.

```
var
  Str1: string;
begin
  Str1 := '1. ' + #9 + Char.ConvertFromUtf32(128) +
    Char.ConvertFromUtf32($1D11E);
  ShowBool(Str1.Chars[0].IsNumber);
  ShowBool(Str1.Chars[1].IsPunctuation);
  ShowBool(Str1.Chars[2].IsWhiteSpace);
  ShowBool(Str1.Chars[3].IsControl);
  ShowBool(Str1.Chars[4].IsSurrogate);
end;
```

위 코드 안에서 결과를 화면에 출력하는데 사용된 함수의 코드는 아래와 같다.

```
procedure TForm1.ShowBool(Value: Boolean);
begin
  Show(BoolToStr(Value, True));
end;
```

참고 유니코드 코드 포인트 \$1D11E는 높은음자리표 기호이다.

유니코드 문자 리터럴 [Literal](#)

지금까지 여러 예제를 통해서 문자열 타입의 변수에 개별 문자 [Character](#) 리터럴 [literal](#) 또는 문자열 [String](#) 리터럴 [literal](#)을 할당하는 것을 보았다. 리터럴을 적을 때는 # 접두사를 적고 그 뒤에 바로 숫자를 적는 일반적인 방식이 매우 간편하다. 하지만 예외가 조금 있다. 이전 버전과의 호환성을 위해, 이 일반 [plain](#) 문자 리터럴은 문맥에 따라 다르게 변환된다. 아래 예문은 숫자 값 128을 문자열에 할당한다. ISO 8859-1 표준의 ASCII-8 세트에서 128 번째 문자인 유로 통화 기호(€)를 할당하려는 코드다.

```
var
  Str1: string;
begin
  Str1 := #128;
```

유니코드에서 유로 통화 기호(€)의 코드 포인트는 \$8364다. 즉, 위 코드는 유니코드를 준수하지 않고 있다. 사실 이 코드를 윈도우에서 실행하여 얻는 값은 공식 ISO 코드 페이지에 따른 결과가 아니라, 마이크로소프트가 윈도우에 반영한 독자적인 구현에

따른 결과다. 기존 코드를 유니코드로 옮기기 쉽도록, 오브젝트 파스칼 컴파일러는 문자열 리터럴이 두 자리 숫자면 ANSI 문자로 처리할 수 있다(이것은 사용자의 실제 코드 페이지가 무엇인가에 따라 달라진다). 놀랍게도, 위 코드를 통해서 얻은 값을 아래와 같이 다시 Char로 변환한 다음 그것의 숫자 표현을 다시 구하면 올바르게 바뀐 숫자를 얻을 수 있다. 따라서 아래 구문을 실행하면,

```
Show(Str1 + ' - ' + IntToStr(Ord(Str1.Chars[0])));
```

결과는 아래와 같다.

```
€ - 8364
```

이전 코드를 완전히 마이그레이션하고, ANSI 기반 리터럴 값이 전혀 없는 것을 선호하는 개발자라면, 컴파일러의 이런 동작을 제어하기 위해 \$HIGHCHARUNICODE라는 컴파일러 지시어^{directive}를 사용할 수 있다. 이 지시어는 #\$80에서 #\$FF 사이의 리터럴 값을 컴파일러가 어떻게 다루는지를 결정한다. 위에서 설명한 내용과 그 결과는 기본 옵션(OFF)이 반영되었을 때다. 이 옵션을 ON으로 지정하면 동일한 프로그램을 실행했을 때 아래와 같이 출력된다(위에서 본 결과와 다르다).

```
☒ - 128
```

즉, 리터럴의 숫자가 유니코드에 실제로 배정된 코드 포인트로 해석되기 때문에, 유니코드 코드 포인트 128에 배정된 문자인 인쇄할 수 없는 제어 문자가 출력된다. 이 옵션과 관계없이 유니코드에 배정된 그대로 코드 포인트(또는 #\$FFFF 아래의 모든 유니코드 코드 포인트)를 표현하려면, 4 자리를 모두 채워서 적으면 된다.

```
Str1 := #$0080;
```

위와 같이 하면 \$HIGHCHARUNICODE 지시어 설정과 관계없이 결코 유로 통화 기호로 해석되지 않는다.

참고 위의 코드와 이에 해당하는 데모는 오직 미국 또는 서유럽 로캘^{locale/지역 설정}에서만 작동한다. 로캘이 다르다면, 128에서 255 사이의 문자가 다르게 해석된다.

4 자리 숫자를 적는 방식은 한중일 문자를 표현할 수 있다는 점도 역시 좋다. 한글 두 글자를 표현하는 방법은 아래와 같다(옮긴이: 원문은 일본 문자로 되어 있음).

```
Str1 := #$C138#$C885;
Show(Str1 + ' - ' + IntToStr(Ord(Str1.Chars[0])) +
      ' - ' + IntToStr(Ord(Str1.Chars[1])));
```

표현되는 한글 문자와 해당 각 문자에 해당되는 정수는 다음과 같다.

```
세종 - 49464 - 51333
```

리터럴에 #\$FFFF 보다 큰 요소를 그대로 사용해도 된다(옮긴이: 예를 들면 #\$1D11E). 이런 리터럴은 써로게이트 쌍으로 알맞게 변환된다.

1-바이트byte 문자는 어떨까?

앞서 언급했듯이, 오브젝트 파스칼 언어는 Char 타입을 WideChar에 매핑한다. 하지만, AnsiChar 타입도 여전히 정의되어 있다. 주로 기존 코드와의 호환성을 위해 남아있다. 1-바이트 데이터 구조에는 대체로 Byte 타입 사용을 권장한다. 하지만, 1-바이트 문자 처리를 할 때 AnsiChar가 편리하다는 점 역시 사실이긴 하다.

AnsiChar는 델파이 일부 버전의 경우 모바일 플랫폼 컴파일러에서 사용할 수 없었다. 하지만, 10.4부터는 AnsiChar를 모든 컴파일러에서 똑같이 쓸 수 있다. 플랫폼 API에 데이터를 매핑하거나 파일에 저장할 때, 여러분은 (지원되긴 해도) 이 오래된 1-바이트 Char 타입을 피해야 한다. 유니코드 인코딩을 쓰는 게 훨씬 더 좋다. 그런데, 1-바이트 문자 처리가 2-바이트 처리보다 더 빠르고 메모리를 적게 쓴다는 점 역시 사실이다.

문자열String 데이터 타입

오브젝트 파스칼의 문자열String 데이터 타입은 단순 문자 배열보다 훨씬 더 복잡하다. 대부분의 프로그래밍 언어에 있는 비슷한 데이터 타입에 비해 훨씬 뛰어난 기능들을 갖추고 있다. 지금부터 오브젝트 파스칼의 문자열 데이터 타입을 받치는 주요 개념들을 소개하겠다. 그리고 이어서, 이 문자열 데이터의 특징들을 보다 자세히 살펴본다.

아래 목록은 오브젝트 파스칼 언어에서 문자열이 작동하는 방식을 이해하는 데 필요한 핵심 개념들이다 (다시 말하지만, 이런 개념들을 많이 알지 못해도, 문자열을 사용하기가 어렵지 않다. 내부 동작이 매우 투명하기 때문이다).

- 문자열 타입은 데이터가 힙heap에 **동적으로 할당**dynamically allocated된다. 문자열 변수는 힙에 있는 실제 데이터를 가리키는 참조reference일 뿐이다. 개발자는 이점을 크게 걱정하지 않아도 된다. 컴파일러가 이를 투명하게 처리하기 때문이다. 동적 배열dynamic array이 그렇듯이, 새 문자열은 선언되면 그 문자열은 빈empty 상태이다.
- 문자열에 데이터를 할당하는 방법은 여러 가지가 있다. 그런데, 개발자가 직접 **특정 양의 메모리를 할당**할 수도 있다. SetLength 함수를 호출하면 된다. 이 함수의 파라미터parameter에는 문자열 안에 들어갈 문자 (각 2 바이트)의 개수를 넣는다. 문자열의 길이Length를 증가시킬 때 이 방식을 사용하면, 기존 데이터가 보존된다 (하지만, 물리적으로 새 메모리 위치로 옮겨질 수 있다). 반대로 길이Length를 줄일 때 사용하면, 그 문자열 데이터 중 일부를 잃게 될 것이다. SetLength 함수를 문자열에서 사용하는 경우는 거의 없다. 오직 문자열 버퍼string buffer를 특정 플랫폼의 운영 체제 함수로 전달할 때만 흔하게 사용된다.
- 메모리에 있는 문자열의 **크기를 증가**시키고 싶는데 (문자열에 다른 문자열을 덧붙이기), 문자열 데이터가 위치한 메모리의 옆 공간을 이미 다른 것이 차지하고 경우에는, 기존 메모리 위치를 그대로 지키면서 크기만 증가할 수는 없다. 따라서 이 경우에는 문자열 데이터 전체가 다른 공간으로 복사된다.

- 문자열 데이터를 비우려면 **clear** 참조 자체를 작업할 필요가 없이, 해당 참조에 빈 **empty** 문자열을 넣으면 된다. 빈 문자열은 “ 또는 이를 표현하는 상수 **constant**인 **Empty**를 사용하면 된다.
- 오브젝트 파스칼의 규칙에 따르면, **문자열의 길이**(Length를 호출하여 얻을 수 있음)는 유효한 **valid** 요소의 수이다. 할당된 **allocated** 요소의 수가 아니다. C 언어는 초창기부터 문자열 종결자 **string terminator** (**#0**) 라는 개념이 있었지만, 이와 달리 오브젝트 파스칼은, 초창기부터, 메모리의 특정 공간(해당 문자열의 일부 공간)을 사용하여 문자열의 실제 길이를 저장하는 방식을 더 좋아했다. 하지만, 때때로 문자열에 종결자가 있는 경우를 볼 수도 있을 것이다.
- 문자열은 **참조-카운팅** **reference-counting** 메커니즘을 사용한다. 참조- 카운팅 방식은 문자열이 차지한 메모리의 위치를 참조하고 있는 문자열 변수의 개수를 계속 파악하여 기록한다. 그러다가 그 문자열을 더 이상 사용되지 않게 되면, 즉 그 문자열 데이터를 참조하는 문자열 변수가 하나도 없게 되면 그 데이터가 차지하던 메모리를 해제 **free** 한다.
- 문자열은 **copy-on-write** **쓰는 시점에 복사** 기술을 사용한다. 그래서 매우 효율적이다. 문자열을 또 다른 문자열에 할당하거나 문자열 파라미터로 전달할 때, 실제 데이터는 전혀 복사되지 않는다. 단지 그 문자열 데이터에 있는 참조 카운트만 1 만큼 증가한다. 하지만, 그 문자열을 참조하는 곳 중 어느 하나가 문자열의 내용 즉 데이터를 변경하면, 시스템은 바로 그때 해당 데이터를 복사를 하고 그 복사본에만 변경을 반영한다. 다른 모든 기존 참조에는 아무런 변화가 없다.
- **문자열 합치기** **string concatenation**를 사용하여 기존 문자열에 내용을 추가할 때, 오브젝트 파스칼은 대체로 매우 빠르다. 심각한 단점도 없다. 이 방법 이외에도 다른 방법들이 있긴 하지만, 문자열 합치기는 빠르고 강력하다. 요즘 많은 프로그래밍 언어들은 그렇지 않다.

지금까지의 설명이 혼란스럽다면, 문자열을 어떻게 사용하는지 보자. 앞에서 설명한 문자열 연산 중 몇 가지를 데모를 통해 보겠다. ‘참조-카운팅 **reference-counting**’과 ‘**copy-on-write** **쓰는 시점에 복사**’ 등에 대해 보게 될 것이다. 하지만 지금은 문자열 헬퍼 **helper**의 연산들 그리고 RTL에서 문자열을 다루는 기초 함수들 몇 가지로 다시 돌아가 보자.

먼저, 앞에서 목록으로 정리했던 요소 중 몇 가지를 실제 코드를 통해서 살펴보겠다. 문자열 연산은 매우 매끄럽기 때문에, 문자열 메모리 구조의 안을 들여다보지 않고는, 어떤 일이 실제로 일어나는지 완전히 파악하기가 어렵다. 다만, 이 장에서 다루기에는 너무 수준 높은 내용이므로 이 책 뒷부분에 가서 따로 보기로 하자. 자 이제 간단한 문자열 연산부터 시작하자. **Strings101** 예제에 있는 예제들이다.

```
var
  String1, String2: string;
begin
  String1 := '안녕하세요';
  String2 := String1;
  Show('1: ' + String1);
```



```
Show( '2: ' + String2);
String2 := String2 + ', 다시한번';
Show( '1: ' + String1);
Show( '2: ' + String2);
end;
```

위 코드를 실행하면, 같은 내용을 두 개의 문자열에 할당한 후에, 그 중 하나를 변경하면 다른 문자열에 영향이 가지 않는다는 점을 알 수 있다. 즉, String1은 String2가 변해도 영향을 받지 않는다.

```
1: 안녕하세요
2: 안녕하세요
1: 안녕하세요
2: 안녕하세요, 다시한번
```

하지만, 뒤에 나오는 데모를 통해 더 자세히 설명하겠지만, String2에 String1을 할당하는 코드는 문자열의 실제 데이터를 복사하지 않는다. 데이터 전체 복사는 문자열이 변경될 때 수행된다. 이른바 *copy-on-write* 쓰는 시점에 복사 라는 특징이다.

이해해야 할 중요한 내용이 하나 더 있다. 문자열의 길이Length가 어떻게 관리되는지 그 방식을 알아야 한다. 오브젝트 파스칼은 문자열의 길이를 실제 값에서 읽어서 얻는다 (문자열의 메타 데이터에 저장되어 있는 값을 직접 읽기 때문에 속도가 매우 빠르다). 반대로, 메모리를 미리 할당하고 싶으면, 즉 SetLength를 호출하면, 지정한 길이만큼 메모리가 확보되지만, 그 메모리 공간에 대한 초기화는 대체로 수행되지 않는다.

SetLength는 애플리케이션 밖에 있는 시스템 함수에게 문자열을 버퍼Buffer로 전달할 때에만 주로 사용된다. 이때 만약 공백Blank 문자열 필요하다면, 의사-생성자pseudo-creator인 Create를 사용할 수 있다. SetLength는 문자열의 앞뒤를 자를 목적으로 사용될 수도 있다. 지금 설명한 내용은 아래 코드를 통해 볼 수 있다.

```
var
  String1: string;
begin
  String1 := '안녕하세요';
  Show(String1);
  Show( 'Length: ' + String1.Length.ToString);
  SetLength(String1, 100);
  Show(String1);
  Show( 'Length: ' + String1.Length.ToString);
  String1 := '안녕하세요';
  Show(String1);
  Show( 'Length: ' + String1.Length.ToString);
  String1 := String1 + string.Create( ' ', 100);
  SetLength(String1, 100);
  Show(String1);
  Show( 'Length: ' + String1.Length.ToString);
```

위 코드의 결과는 아래와 비슷하다.

Length: 100

하지만 파라미터로 전달된 문자열이 함수 안에서 수정되는 못하게 하고 싶다면 어떻게 할까? 이 경우, 파라미터로 전달할 때 제어자^{modifier}인 `const` 키워드를 사용하여 최적화를 반영할 수 있다. 이렇게 하면 함수나 프로시저 안에서 해당 문자열을 변경되지 못하도록 컴파일러가 막아주는 효과가 있을 뿐만 아니라, 결과적으로 파라미터를 전달하는 동작 자체가 최적화된다. 실제로 파라미터로 전달되는 `const` 문자열을 받은 함수는 시작할 때 그 문자열의 참조 카운트를 증가시키는 함수를 실행할 필요가 없고, 끝날 때 참조 카운트를 감소시킬 필요도 없다. 함수 안에서 이 문자열이 변경되지 않는다는 것을 컴파일러가 이미 알고 있기 때문이다.

오브젝트 파스칼에서 문자열을 관리하는 루틴^{routine}들은 속도가 매우 빠르지만, 어느 루틴이든 수천 번 또는 수백만 번 실행된다면 프로그램에 조금이라도 추가 부담을 준다. 따라서 함수 안에서 문자열 파라미터의 값을 변경할 필요가 없는 문자열이라면 `const`로 전달하는 것이 좋다 (잠재적인 이슈가 있긴 하다. 이 이슈에 대해서는 아래의 참고에 설명되어 있다).

문자열 파라미터를 전달하는 세 가지 방식은 아래 프로시저 선언문에서 볼 수 있다.

```
procedure ShowMsg1(Str: string);
procedure ShowMsg2(var Str: string);
procedure ShowMsg3(const Str: string);
```

참고 전달받은 문자열을 함수와 메서드 안에서 변경하지 않는다면, 문자열을 `const` 파라미터로 전달해야 한다는 주장이 최근 몇 년 동안 강하게 제기되었다. 하지만 매우 중요한 주의사항이 있다. 상수^{constant}로 전달된 문자열 파라미터에 대해서는, 컴파일러가 그 문자열의 참조만 가져올 뿐, (참조 카운팅 등) "관리"를 전혀 하지 않는다. 즉, 문자열이 저장된 메모리 상의 위치를 가리키는 포인터^{pointer}로만 취급한다. 이 경우, 컴파일러는 루틴 안의 코드를 확인하여 상수 파라미터가 변경되지 못하도록 방지하긴 하지만, 상수 파라미터가 가리키고 있는 곳 즉 실제 문자열이 있는 메모리 위치에서 발생하는 일들은 전혀 제어하지 못한다. 문자열은 변경되면, 메모리에서 그 문자열의 배치^{layout}와 위치^{location}가 영향을 받을 수 있다. `const`가 아닌 일반 문자열 파라미터라면 이런 문제를 잘 대처한다 (문자열이 여러 곳에서 참고되고 있다면 자동으로 copy-on-write^{쓰는 시점에 복사} 동작을 수행한다). 하지만 `const` 문자열 파라미터는 메모리 위치만 가리키고 있으므로, (문자열이 변경되는 경우에) 유효하지 않은 위치를 계속 가리키게 되고, 유효하지 않은 메모리 위치를 사용하면 메모리 접근^{access} 오류가 발생할 가능성이 높다.

[] 사용 및 문자열String에서 문자 카운팅 모드Character Counting Mode

오브젝트 파스칼나 기타 다른 프로그래밍 언어를 사용한 경험이 있다면 알겠지만, 문자열 안에 있는 요소에 접근하는 연산은 매우 중요하며, 대체로 배열의 요소에 접근할 때와 같이 대괄호([])를 사용하여 접근한다.

오브젝트 파스칼에서 이 연산을 수행하는 방법이 두 가지이며, 서로 조금씩 다르다.

- `Chars[]` 문자열 타입용 헬퍼의 연산(전체 목록은 다음 부분에 나열되어 있음)은 읽기 전용이며, 0-기반 인덱스를 사용한다.
- 표준 `[]` 문자열 연산자는 읽기와 쓰기를 모두 지원하며, 파스칼에서 전통적으로 사용하던 1-기반 인덱스를 기본적으로 사용한다. 이 설정은 컴파일러 지시어를 사용하여 변경할 수 있다.

이 문제에 대한 간략한 역사적 배경과 몇 가지 관련 사항에 대해서는 아래 참고에 보다 명확하게 설명되어 있다. 관심이 없다면 건너뛰어도 좋다. 굳이 설명한 이유는 지난 시간 동안 일어난 일들을 살펴보지 않고는, 이 언어가 지금처럼 작동하는 이유를 이해하기 어렵기 때문이다.

참고 잠시 시간을 거슬러 올라가 지금에 이르게 된 과정을 설명하겠다. 파스칼 언어 초창기에는 문자열을 문자의 배열처럼 취급했고, 배열의 첫 번째 요소(배열의 0번째 요소)를 사용하여 문자열에 있는 유효한 문자의 수, 즉 문자열의 길이`length`를 저장했다. 당시 C 언어는 문자열의 길이를 매번 다시 계산해야만 했다. 즉 NULL 종결자`terminator`를 찾을 때까지 헤아리는 방식이었다. 하지만, 파스칼 코드는 길이가 저장된 바이트만 확인하면 바로 얻을 수 있었다. 파스칼은 이처럼 길이를 저장하기 위해 0 번째 바이트를 사용했기 때문에 실제 첫 문자가 있는 위치의 인덱스가 1이었다.

시간이 지나면서, 거의 모든 언어이 0-기반으로 문자열과 배열을 사용했다. 그 후, 오브젝트 파스칼에서는 0-기반 동적 배열`dynamic array`이 채택되었고 RTL과 컴포넌트 라이브러리 대부분 역시 0-기반 데이터 구조를 사용했다. 하지만 문자열 만은 예외였다.

모바일 세상으로 옮겨갈 때, 오브젝트 파스칼 언어 설계자는 0-기반 문자열을 "우선하기"로 결정했다. 그렇지만, 개발자들이 계속해서 이전 방식을 사용할 수 있도록 컴파일러 지시어로 이 동작을 제어할 수 있게 해 놓았다. 이는 기존 오브젝트 파스칼로 만든 소스 코드가 있는 개발자들을 고려한 것이다. 하지만 델파이 10.4에서 다시 원래의 결정으로 돌아갔다. 이유는 대상 플랫폼이 무엇이든 소스 코드가 일관성을 가지는 것이 더 중요하다는 점을 고려했기 때문이었다. 즉, "다른 최신 언어와 같아야 한다"는 목표보다 "단일 소스 멀티 플랫폼"이라는 목표를 더 선호하기로 결정했던 것이다.

인덱스`index` 기준의 차이를 알기 쉽게 설명하기 위해, 유럽과 북미의 층수 계산 방식을 비교해보자 (다른 나라는 솔직히 잘 모르겠다). 유럽에서는 땅에 닿은 층이 0층이고, 그보다 한 층 높은 층이 1층이다(공식적으로는 "지상 1층"`floor one above ground`)이라고 표시하기도 한다). 북미에서는 1층이 지면에 있는 층이고, 그보다 한 층 높으면 2층이다. 즉, 미국은 층수 인덱스가 1-기반인 반면 유럽은 0-기반 층수 인덱스를 사용한다.

문자열의 경우, 프로그래밍 언어 대부분이 0-기반 표기법을 쓴다. 하지만, 델파이와 대부분의 파스칼 계열 언어들은 1-기반 표기법을 사용한다.

문자열 인덱스를 더 자세히 설명하자면, 위에서 언급했듯이, `Chars[]` 표기는 항상 0-기반 인덱스를 사용한다. 따라서 다음 코드를 실행하면,

```
var
  String1: string;
begin
```



```
String1 := 'Hello world';
Show(String1.Chars[1]);
```

결과는 다음과 같다:

```
e
```

[1]를 문자열 변수 바로 뒤에서 직접 사용하면 출력은 어떻게 될까?

```
Show(String1[1]);
```

기본 설정이라면, 출력은 H이다. 그러나 만약 컴파일러에서 \$ZEROBASEDSTRING을 ON으로 정의한 경우라면, e가 된다. 현재(즉, 델파이 10.4 버전 이후) 권장 사항은 모든 코드에서 1-기반 문자열을 사용하는 것이다. 그렇게 하면, 기본 모델이 0-기반이었던 과도기 시절의 코드 인해 발생될 수 있는 이슈들을 피할 수 있다.

하지만 \$ZEROBASEDSTRING이 무엇이든 영향을 받지 않는 코드를 작성하고 싶다면 어떻게 할까? 인덱스 추상화를 해 볼 수 있다. 즉 Low(String)을 이용해 첫번째 인덱스를, High(String)을 이용해 마지막 인덱스를 얻는다. 아래 코드는 컴파일러 설정 상 문자열 인덱스 기반이 무엇이든 항상 알맞은 값을 반환한다.

```
var
  S: string;
  I: Integer;
begin
  S := 'Hello world';
  for I := Low(S) to High(S) do
    Show(S[I]);
```

다시 말해, 문자열 안에 있는 요소들이 범위는 항상 Low 함수의 결과에서 시작하고 High 함수의 결과로 끝난다.

참고 문자열은 문자열일 뿐이다. 0-기반 문자열이라는 개념은 완전히 잘못된 것이다. 표기 방식이 무엇이든 문자열 데이터가 메모리 안에 들어 구조는 다르지 않다. 따라서 문자열을 전달할 때, 전달받는 함수가 무슨 인덱스를 기반으로 하든 관계없이 얼마든지 문자열을 전달할 수 있으며 전혀 문제가 없다. 다시 말해서, 0-기반 표기법으로 문자열을 접근하는 코드 조각을 문자열에 넣었다고 가정할 때, 그 문자열을 1-기반 표기법을 사용하도록 설정되어 컴파일 된 함수에게 전달할 수 있다.

문자열String 합치기concatenating

이미 언급했지만, 다른 언어와 달리, 오브젝트 파스칼은 문자열 직접 합치기concatenation를 완전히 지원하며, 실제로 꽤 빠르게 동작한다. 지금은 문자열을 합치는 코드 몇 개로 속도 테스트를 하겠다. 속도 비교 대상인 TStringBuilder 클래스에 대한 설명은 나중에 18 장에서 간략히 다루기로 하고 지금은 닷넷.NET에서 문자열을 다른 문자열들과 조합할 때 사용하는 표기법을 따르고 있다는 정도로만 알아 두자. TStringBuilder

를 사용하는 경우에는 여러 이유가 있지만, 성능이 가장 큰 이유는 아니다 (아래 예제를 통해 확인할 수 있다).

그렇다면, 오브젝트 파스칼에서는 문자열을 어떻게 합치는가? 우리는 연산자 `+`를 사용한다.

```
var
  Str1, Str2: string;
begin
  Str1 := 'Hello, ';
  Str2 := 'world';
  Str1 := Str1 + ' ' + Str2;
```

할당의 왼쪽과 오른쪽 모두에서 `Str1` 변수가 사용된다는 점을 주목하자. 문자열을 새로 만들어서 할당하는 대신, 이런 식으로 기존 문자열에 내용을 더 추가할 수도 있다.

두 방식 모두 가능하긴 하다. 하지만 기존 문자열에 내용을 추가하는 쪽이 성능을 더 좋게 한다.

반복^{loop} 안에서도 이런 문자열 합치기 방식을 사용할 수 있다. `LargeString` 예제에 있는 다음 코드를 보자.

```
uses
  System.Diagnostics;

const
  MaxLoop = 2_000_000; // 2백만

var
  Str1, Str2: string;
  I: Integer;
  T1: TStopwatch;
begin
  Str1 := 'Marco ';
  Str2 := 'Cantu ';

  T1 := TStopwatch.StartNew;
  for I := 1 to MaxLoop do
    Str1 := Str1 + Str2;

  T1.Stop;
  Show( 'Length: ' + Str1.Length.ToString);
  Show( 'Concatenation: ' + T1.ElapsedMilliseconds.ToString);
end;
```

윈도우^{Windows} 가상 머신과 안드로이드^{Android} 장비에서 이 코드를 실행한 처리 시간은 다음과 같다(컴퓨터가 훨씬 더 빠름).

```
Length: 12000006 // 윈도우 (가상 머신에서)
Concatenation: 59
Length: 12000006 // 안드로이드 (장비에서)
Concatenation: 991
```


비슷한 코드를 TStringBuilder 클래스를 사용하여 구현해 실행한 결과는 아래와 같다. 코드는 지금 설명하지 않겠다 (TStringBuilder 클래스는 18 장에서 다룬다). 지금은 실제 처리 시간을 위에서 본 일반 합치기의 처리 시간과 비교해 보기 바란다.

```
Length: 12000006 // 윈도우 (가상 머신에서)
StringBuilder: 79
Length: 12000006 // 안드로이드 (장비에서)
StringBuilder: 1057
```

보다시피 문자열 직접 합치기가 가장 빠르다는 점을 안심하고 믿을 수 있다.

String 헬퍼Helper 연산

문자열 타입의 중요성을 감안하면, 문자열 타입용 헬퍼Helper 안에 많은 연산들이 들어 있다는 점이 놀랍지 않다. 게다가, 그 연산들은 대부분의 애플리케이션에서 중요하고 많이 사용된다. 따라서 지금 이 목록을 주의 깊게 살펴볼 가치가 있다.

텔파이에는, 문자열을 조작하는 글로벌(global/전역) 함수들(오래된 것들임)과 문자열 타입용 헬퍼helper의 메서드(method)들이 있다. 이들 사이에는 중요한 차이가 있다. 고전적인 연산은 1-기반 문자열이라고 가정하는 경향이 있다. 하지만, 문자열 헬퍼 연산은 0-기반 논리 logic를 사용한다!

아래는 문자열 헬퍼 연산들(대부분 오버로드된 overloaded 버전들을 많이 가지고 있음)을 논리적으로 묶어서 나열한 목록이다. 여기에는 연산이 수행하는 동작을 간략히 설명하고 있다. 대체로 이름들이 직관적인 편이라는 점을 알 수 있을 것이다.

- 복사, 부분 복사: Copy, CopyTo, Join, SubString 등
- 문자열 변경: Insert, Remove, Replace 등
- 다양한 데이터 타입을 문자열로 변환: Parse, Format 등
- 다양한 데이터 타입으로 변환 (가능한 경우에 작동함): ToBoolean, ToInteger, ToSingle, ToDouble, ToExtended. 문자열을 문자 배열로 변환은 ToCharArray
- 공백, 특정 문자 등을 문자열에 채워 넣기: PadLeft, PadRight, 오버로드된 Create. 그 반대로 문자열 앞뒤에 있는 공백 제거는 TrimRight, TrimLeft, Trim
- 문자열끼리 비교하거나 동일한지 확인: Compare, CompareOrdinal, CompareText, CompareTo, Equals. 하지만, = 연산자와 비교 연산자도 사용 가능
- 대소문자 변환: LowerCase / UpperCase, ToLower / ToUpper, ToUpperInvariant
- 문자열 내용에 대한 조건 테스트: Contains, StartsWith, EndsWith 등. 문자열 안을 검색할 때, 주어진 문자의 위치 찾기 IndexOf (첫 문자부터 또는 특정 위치부터 검색), 이와 비슷한 IndexOfAny (문자 배열 안에서 요소 하나를 검색), 뒤에서부터 찾는 LastIndexOf, LastIndexOfAny, 특수 목적 연산 IsDelimiter, LastDelimiter
- 문자열에 관한 일반 정보 접근: Length (문자 수 반환), CountChars (써로게이트 쌍도 고려함), GetHashCode (해당 문자열의 해쉬를 반환), 비어 있는 지를 확인할 때는 IsEmpty, IsNullOrEmpty, IsNullOrWhiteSpace

- 문자열 만의 고유 연산: Split(특정 문자를 기준으로 문자열을 여러 개로 나눔), QuotedString / DeQuotedString (문자열을 따옴표로 감싸기 / 따옴표 제거)
- 개별 문자에 접근: Chars[] (대괄호 안에는 해당 요소의 인덱스 번호를 넣는다). 값을 읽는 용도이므로, 변경할 때는 사용할 수 없다. 다른 모든 문자열 헬퍼가 그렇듯이 0-기반 인덱스를 사용한다.

알아야 할 중요한 점이 있다. 모든 문자열 헬퍼의 메서드들은 문자열 다루기 관행 (문자열 요소는 0부터 시작해서 문자열의 길이에서 1을 뺀 값까지 올라간다는 개념으로 다른 많은 언어에서 사용되고 있다)을 따라서 만들어졌다. 이미 언급했지만 강조할 필요가 있어서 다시 한번 더 언급하겠다. 문자열 헬퍼 [helper](#)의 연산은 0-기반 인덱스를 파라미터 [parameter](#)와 반환값 [return value](#)에서 사용한다.

참고 Split 연산은 비교적 최근에 오브젝트 파스칼 RTL에 추가되었다. 예전에 많이 쓴 방식은 문자열 안에 특정한 줄 구분자 [line separator](#)를 설정해 놓은 후에, 그 문자열을 TStringList에 넣고, 그 스트링리스트 안의 각 줄 [line](#)에 접근하는 방식으로 각 문자열 얻었다. Split 연산은 이보다 훨씬 더 효율적이고 유연하다.

문자열에 직접 적용할 수 있는 연산들이 많지만, 지금은 매우 일반적인 연산이면서 비교적 간단한 몇 가지 연산에만 집중하겠다. StringHelperTest 예제는 여기에는 버튼이 두 개 있다. 각 버튼은 코드 첫 부분에서 문자열 하나를 만들고 표시한다.

```
var
  Str1, Str2: string;
  I, NIndex: Integer;
begin
  Str1 := '';

  // 문자열 만들기
  for I := 1 to 10 do
    Str1 := Str1 + 'Object ';

  Str2:= string.Copy(Str1);
  Str1 := Str2 + 'Pascal ' + Str2.SubString(10, 30);
  Show(Str1);
```

위 데모는 Copy 함수를 사용한다. Copy는 문자열의 별칭 [alias](#)이 아니라 문자열의 데이터 자체를 복사하는 함수다. 그렇지만 이 데모에서는 별 차이가 없다. 그 뒤에 나오는 SubString은 문자열에서 일부를 추출한다. 결과는 다음과 같다.

```
Object Object Object Object Object Object Object Object Object Object
Pascal ect Object Object Object Object
```

이렇게 초기 작업을 완료한 다음, 첫 번째 버튼은 부분 문자열을 검색하는 코드를 실행한다. 이 검색은 위치를 옮기면서 반복한다. 즉, 지정되는 인덱스의 위치부터 주어진 문자열(이 예에서는 한 글자, 즉 대문자 “O”)을 찾는다. 그리고 일치하는 것이 있을 때 마다 그 횟수를 헤아린다.


```
// 부분 문자열 찾기
Show( 'Pascal 이 있는 위치: ' +
      Str1.IndexOf( 'Pascal' ).ToString );

// 일치하는 횟수 세기
I := -1;
NCount := 0;
repeat
  I := Str1.IndexOf( 'O', I + 1 ); // 다음 요소부터 검색
  if I >= 0 then
    Inc( NCount ); // 1개 찾음
until I < 0;

Show( 'O 가 발견된 횟수: ' +
      NCount.ToString + ' 번' );
```

위 반복 루프에서, 검색을 처음 시작할 때는 인덱스가 -1이다. 그 다음부터는 일치하는 문자열이 발견된 위치의 바로 뒤 위치가 인덱스가 된다. 이것을 반복하면서 발견된 횟수를 센다. 일치하는 요소가 더 이상 없으면 IndexOf가 -1을 반환하므로, 이 루프는 그 때 종료된다. 결과는 다음과 같다.

```
Pascal 이 있는 위치: 70
O 가 발견된 횟수: 14 번
```

두 번째 버튼은 문자열에서 하나 이상의 요소를 찾아서 다른 요소로 바꾸는 코드를 실행한다. 첫 번째 부분에서는 찾아낸 문자열의 앞에 있는 부분과 뒤에 있는 부분을 복사하고 그 사이에는 새로 교체되어 들어갈 텍스트를 넣는다. 두 번째 부분은 Replace 함수를 사용한다. 이 함수는 알맞은 플래그(rfReplaceAll)를 전달하기만 하면 일치하는 것 여러 개를 간단히 바꿀 수 있다. 코드는 아래와 같다.

```
// 1 개 바꾸기
NIndex := Str1.IndexOf( 'Pascal' );
Str1 := Str1.SubString( 0, NIndex ) + 'Object' +
        Str1.SubString( NIndex + ( 'Pascal' ).Length );
Show( Str1 );

// 여러 개 바꾸기
Str1 := Str1.Replace( 'O', 'o', [rfReplaceAll] );
Show( Str1 );
```

결과가 꽤 길기 때문에, 읽기 쉽게 각 문자열의 중간 부분만 적었다.

```
...Object Pascal ect. Object Object...// 원래 문자열
...Object Object ect. Object Object...// 1 개 바꾸기 결과
...object object ect. object object...// 여러 개 바꾸기 결과
```

다시 말하지만, 이것은 문자열 타입용 헬퍼^{helper}를 사용하여 수행할 수 있는 다양한 문자열 연산 중 극히 일부일 뿐이다.

RTL에 있는 문자열 함수들 더 보기

문자열 헬퍼를 구현하면서, 함수 이름은 다른 프로그래밍 언어에서 흔히 볼 수 있는 연산자 이름을 따라가기로 결정했다. 그 결과, 헬퍼의 문자열 연산의 이름들과 원래부터 오브젝트 파스칼에 있던 기존 연산의 이름들(여전히 글로벌 함수로 제공되고 있음)이 서로 다른 경우가 꽤 있다.

일치하지 않는 함수 이름 몇 가지는 다음 표와 같다.

글로벌	문자열 타입 헬퍼
Pos	IndexOf
IntToStr	Parse
StrToInt	ToInteger
CharsOf	Create
StringReplace	Replace

참고 글로벌 함수와 Char 헬퍼 연산 사이에는 큰 차이가 있다는 점을 다시 강조한다. 전자는 문자열 안에 들어 있는 요소를 인덱스를 사용해 접근할 때, 1-기반 표기법을 사용하는 반면, 후자는 0-기반 표기법을 사용한다(앞에서 설명했다).

위 목록에는 RTL의 문자열 함수들 중 가장 널리 사용되고 이름도 달라진 것들만 나열했다. 그 외에도 UpperCase, QuotedString 등 많은 함수들이 있는데, 이것들은 기존 이름을 그대로 쓴다. System.SysUtils 유닛 안에는 문자열을 다루는 함수가 훨씬 더 많이 들어있다. System.StrUtils 유닛에도 역시 문자열을 다루는 함수들이 많은데 주로 문자열 헬퍼에 들어있지 않은 문자열 처리를 하는 것들이다.

System.StrUtils 유닛에서 주목할 만한 함수는 다음과 같다.

- **ResemblesText**: Soundex 알고리즘(단어를 식별할 때 실제 철자 대신 단어의 소리를 사용하는 알고리즘)을 구현한다
- **DupeString**: 문자열이 주어지면 요청된 복사본 수만큼 중복하여 반환
- **IfThen**: 조건이 True이면, 앞에 있는 문자열을 반환하고, False이면 뒤에 있는 문자열을 반환 (이 장의 앞부분에 있는 예제 코드에서 이 함수를 사용했었다)
- **ReverseString**: 문자열 안에 있는 문자들의 순서를 반대로 뒤집은 결과를 반환

문자열에 서식을 반영하기 **Formatting Strings**

비록 더하기(+) 연산자로 문자열을 합칠 수 있고, 변환 함수들을 사용하여 다양한 데이터 타입의 값들이 포함된 복합적인 문자열을 만들 수 있지만, 더 강력한 다른 방법이 있다. 숫자, 통화-currency 값, 기타 문자열에 서식을 반영하고 합쳐서 원하는 문자열을 만들 수 있다. Format 함수를 호출하면 이런 복합적인 문자열을 구성할 수 있다. 아주 오래 전부터 있던 함수이지만, 여전히 매우 많이 사용되는 메커니즘이다. 오브젝트 파스칼 이외에도, 프로그래밍 언어 대부분에서 사용되는 방식이다.

참고 "print format string" 함수 가족 즉 `printf` 함수들은 프로그래밍 초창기의 FORTRAN 66, PL/1, ALGOL 68 등의 언어로 거슬러 올라간다. 독특한 서식 문자열 *format string* 구조는 지금도 여전히 사용되고 있으며(그리고 오브젝트 파스칼에서도 사용됨), C 언어의 `printf` 함수와 비슷하다. 역사적인 개요는 en.wikipedia.org/wiki/Printf_format_string을 참고.

`Format` 함수의 첫번째 파라미터는 문자열이다. 이 문자열은 기본 텍스트와 자리 표시 *placeholder*(% 기호로 표시) 몇 개로 구성된다. 그리고 두번째 파라미터에는 값들의 배열이 들어간다. 이 배열에는 대체로 문자열 안에 있는 자리 표시의 개수만큼의 요소, 즉 각 자리 표시에 들어갈 값들이 나열된다. 예를 들어 숫자 2 개를 문자열 안에 넣으려면 다음과 같이 한다.

```
| Format ( 'First %d, Second %d', [N1, N2] );
```

여기에서 `N1`과 `N2`는 정수 *integer* 값이다. 첫번째 자리 표시에는 첫번째 값이, 두번째 자리 표시에는 두 번째 값이 들어 간다. 만약 자리 표시의 출력 타입(% 기호 뒤의 문자)과 그 자리에 들어갈 값의 타입이 일치하지 않으면 런타임 오류가 발생한다. 이와 마찬가지로 자리 표시에 들어갈 값들이 충분히 제공되지 않을 때에도 런타임 오류가 발생한다. 컴파일 시 타입 검사가 없는 점이 `Format` 함수의 최대 단점이다.

`Format` 함수는 오픈-배열 파라미터 *open-array parameter*(5장에서 설명한 대로 파라미터의 수와 데이터 타입을 원하는 대로 넣을 수 있는 파라미터)를 사용한다. 자리 표시는 %d 이외에도 여러 가지가 있다. 자리 표시는 `Format` 함수에 정의되어 있으며 아래 표에 간략히 나열해 놓았다. 자리 표시에 값이 출력될 때는 기본 형식이 반영된다. 하지만, 추가로 형식 명시 *specifier*를 사용하면 출력 형식을 바꿀 수 있다. 예를 들어, 길이 명시 *specifier*를 사용하여 출력되는 문자의 수를 명시하거나, 정밀도 명시를 사용하여 소수점 자릿수를 명시할 수 있다. 예를 들면,

```
| Format( '%8d', [N1] );
```

위 코드는 정수 `N1`의 값을 8 글자 문자열로 변환하는데, 빈자리는 공백으로 채운다. 그리고 오른쪽 맞춤을 반영한다(왼쪽 맞춤으로 지정하려면 빼기 즉 - 기호를 사용). `Format` 함수에서 자리 표시에 사용하는 여러 데이터 타입은 다음과 같다.

d (decimal)	해당 정수 <i>integer</i> 값이 10진수를 표현하는 문자열로 변환됨
x (hexadecimal)	해당 정수 <i>integer</i> 값이 16진수를 표현하는 문자열로 변환됨
p (pointer)	해당 포인터 <i>pointer</i> 값이 16진수로 표현된 문자열로 변환됨
s (string)	해당 문자열, 문자, <code>PChar</code> (문자 배열을 가리키는 포인터)의 값이 문자열로 복사되어 반영됨
e (exponential)	해당 부동-소수점 <i>floating-point</i> 값이 과학 표기식 문자열로 변환됨

f (floating point) 해당 부동-소수점 [floating-point](#) 값이 부동-소수식 문자열로 변환됨.

g (general) 해당 부동-소수점 [floating-point](#) 값이 가능한 가장 짧은 10진수 문자열로 변환되는 데, 부동-소수식 또는 지수식 중 하나가 반영됨

n (number) 해당 부동-소수점 [floating-point](#) 값이 부동-소수식 문자열로 변환되는데 천 단위 구분 표시가 반영됨 (대체로 지역 설정을 따름)

m (money) 해당 부동-소수점 [floating-point](#) 값이 통화 [currency](#) 금액 문자열로 변환됨 (대체로 지역 설정을 따름)

변환이 어떻게 되는지 알아보는 가장 좋은 방법은 `format`을 직접 다양하게 실험해 보는 것이다. `FormatString` 예제에는 정수 값들을 미리 정의하고, 이것들을 여러 포맷 [format](#) 문자열에 대응시켜서 변환한다.

이 프로그램의 폼 [form](#) 화면에는, 첫번째 버튼 바로 위에 에디트 [edit](#) 상자가 하나 있다. 이 에디트 상자 안에는 자리 표시들을 사용한 간단한 포맷('%d - %d - %d')이 미리 정의되어 있다. 첫번째 버튼을 누르면 에디트 상자에 보다 복잡한 포맷이 표시된다 (에디트 [edit](#) 상자의 텍스트 [Text](#)에 포맷 문자열 'Value %d, Align %4d, Fill %4.4d'를 할당하는 코드가 실행된다). 두번째 버튼은 에디트 상자에 들어있는 포맷 문자열에 미리 정의해 놓은 숫자 값들을 넣는다. 코드는 아래와 같다.

```
var
  StrFmt: string;
  N1, N2, N3: Integer;
begin
  StrFmt := Edit1.Text;
  N1 := 8;
  N2 := 16;
  N3 := 256;

  Show(Format('Format string: %s', [StrFmt]));
  Show(Format('Input data: [%d, %d, %d]', [N1, N2, N3]));
  Show(Format('Output: %s', [Format(StrFmt, [N1, N2, N3])]));
  Show(''); // 빈 줄
end;
```

먼저, 간단한 포맷 문자열을 반영하여 출력하고, 그 다음에, 복잡한 포맷 문자열을 반영된 결과를 표시한다면 (즉, 먼저 두번째 버튼을 누르고, 이어서 첫번째 버튼을 누르고, 다시 두번째 버튼을 누르면) 다음과 같은 출력이 나온다.

```
Format string: %d - %d - %d
Input data: [8, 16, 256]
Output: 8 - 16 - 256

Format string: Value %d, Align %4d, Fill %4.4d
Input data: [8, 16, 256]
Output: Value 8, Align 16, Fill 0256
```


여러분이 직접 이 프로그램 코드를 열고 직접 포맷 문자열을 편집해보고, 다양한 포맷 옵션을 직접 경험해 보기를 바란다.

문자열String의 내부 구조Structure

문자열의 내부를 많이 알지 못해도, 문자열을 일반적으로 사용할 수 있다. 하지만, 문자열 데이터 타입 뒤편의 실제 데이터 구조를 살펴보는 것은 흥미로운 일이다. 파스칼 언어 초창기에는 문자열 안에 한 문자 당 1 바이트byte인 요소를 최대 255 자까지 넣을 수 있었고, 맨 앞에 있는 바이트(즉 바이트 0)에는 문자열의 길이length가 저장되었다. 그 초기 시절이 지나고 많은 시간이 흘렀지만, 문자열에 관한 추가 정보를 문자열 데이터의 일부에 저장한다는 개념은 여전히 오브젝트 파스칼에 남아 있다 (C에서 파생된 많은 언어들이 문자열 종결자terminator 개념을 사용하는 것과는 다르다).

참고 ShortString은 파스칼에서 오래전부터 사용되던 문자열 타입의 이름이다. 이 타입에는 1 바이트 문자, 즉 AnsiChar가 최대 255자까지 들어갈 수 있다. ShortString 타입은 데스크탑 컴파일러에서는 여전히 사용할 수 있지만 모바일 컴파일러에서는 사용할 수 없다. ShortString과 비슷한 데이터 구조를 표현하려면 동적dynamic 바이트 배열array of bytes 즉, TBytes를 사용하거나, 또는 Byte 요소를 담은 일반plain 정적static 배열array을 사용한다.

이미 언급했듯이, 문자열 변수variable는 힙heap에 할당되어 있는 문자열 데이터 구조를 가리키는 포인터pointer일 뿐이다. 문자열 변수에 저장된 값은 사실 그 문자열 데이터 구조에서 가장 앞에 있는 위치를 가리키는 참조reference가 아니라 그 문자열의 첫번째 문자가 들어 있는 위치를 가리키는 참조다. 문자열에 관한 메타 데이터meta data는 문자열 변수가 가리키는 위치를 기준으로 음수만큼 떨어진 곳에 있다. 아래 표는 메모리 안에 문자열 타입 데이터가 어떻게 들어있는지를 보여준다.

-12	-10	-8	-4	문자열 참조가 가리키는 주소
코드 페이지	요소의 크기	참조 카운트	길이	문자열의 첫번째 문자

첫번째 요소(문자열의 내용이 시작되는 위치에서부터 거꾸로 가면서 셀 때)에는 정수integer가 있는데, 그 문자열의 길이length가 담긴다. 두번째 요소에는 참조 카운트reference count가 들어간다. 그 다음 필드에는 (데스크탑 컴파일러에서 사용됨) 문자열 요소의 크기가 몇 바이트인지 기록되고(1 또는 2 바이트임), 가장 먼 곳에는 코드 페이지가 기록되는데, 이전 ANSI-기반 문자열 타입을 위한 것이다.

놀랍게도, 우리는 이 필드 대부분을 접근할 수 있다. 뻔한 Length 함수 이외에도 아래와 같은 여러 저-수준 문자열 메타 데이터 함수를 사용하면 된다.

```
function StringElementSize(const S: string): Word;
function StringCodePage(const S: string): Word;
function StringRefCount(const S: string): LongInt;
```


예를 들어 StringMetaTest 예제는 아래 코드와 같이 문자열을 생성하고, 이어서 생성된 그 문자열에 대한 정보를 얻어 낸다.

```
var
  Str1: string;
begin
  Str1 := 'F' + string.Create('o', 2);

  Show('SizeOf: ' + SizeOf(Str1).ToString);
  Show('Length: ' + Str1.Length.ToString);
  Show('StringElementSize: ' + StringElementSize(Str1).ToString);
  Show('StringRefCount: ' + StringRefCount(Str1).ToString);
  Show('StringCodePage: ' + StringCodePage(Str1).ToString);
  if StringCodePage(Str1) = DefaultUnicodeCodePage then
    Show('Is Unicode');
  Show('Size in bytes: ' +
    (Length(Str1) * StringElementSize(Str1)).ToString);
  Show('ByteLength: ' + ByteLength(Str1).ToString);
```

참고 위 프로그램은 'Foo'라는 문자열을 동적으로 [dynamically](#) 생성했다. 그래야만 참조 카운트를 가진다. 그렇지 않고 'Foo'를 직접 입력하여 문자열에 할당하면, 그 상수 [constant](#) 문자열은 참조 카운트가 비활성화 된다(1로 지정됨). 데모에서 참조 카운트가 정확히 보여주기 위해 문자열을 동적으로 생성했다 (옮긴이: 위 데모의 문자열이 로컬 변수이기 때문에 필요함).

이 프로그램을 실행한 결과는 아래와 비슷하게 나온다.

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: 1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

위 결과를 잘 보자. 반환된 코드 페이지가 1200이다. 이 문자열 즉 UnicodeString 타입은 글로벌 [global/전역](#) 변수인 DefaultUnicodeCodePage에 저장된 숫자를 반환한다. 문자열 변수 [variable](#)의 크기 [SizeOf](#) (항상 4이다), 문자열의 논리적인 길이 [Length](#), 물리적인 길이 [Size in bytes](#)가 서로 다르다는 점 역시 위 결과가 분명하게 보여준다.

물리적인 길이를 얻으려면 문자 1개가 차지하는 바이트의 크기에 문자의 개수를 곱하면 된다. 또는 ByteLength 함수를 호출해도 얻을 수 있다. 하지만, 이 함수는 구형 데스탑 컴파일러의 문자열 타입 중 몇 가지는 지원하지 않는다.

문자열String 메모리 안쪽을 보기

문자열의 메타 데이터 [meta data](#)를 살펴보면, 문자열이 메모리에서 어떻게 관리되는지 특히 참조 카운팅이 어떻게 작동하는지를 더 잘 이해할 수 있다. StringMetaTest 예제에 있는 다른 코드를 더 보자.

이 프로그램 안에는 MyStr1과 MyStr2라는 글로벌/global/전역 문자열이 있다. 실행하면, 동적으로 문자열을 생성하여(위 참고에서 그 이유를 설명했다) 두 변수 중 첫번째 변수에 할당한다. 그리고 나서, 그 첫번째 변수를 두번째 변수에 할당한다.

```
MyStr1 := string.Create(['H', 'e', 'l', 'l', 'o']);
MyStr2 := MyStr1;
```

이 프로그램은 문자열을 조작하고 나서, 그 문자열의 내부 상태를 표시한다. 이때 사용되는 StringStatus 함수의 코드는 다음과 같다.

```
function StringStatus(const Str: string): string;
begin
    Result := 'Addr: ' + IntToStr(Integer(Str)) +
              ', Len: ' + IntToStr(Length(Str)) +
              ', Ref: ' + IntToStr(PInteger(Integer(Str) - 8)^) +
              ', Val: ' + Str;
end;
```

함수에서는 문자열 파라미터를 const 파라미터로 전달하고 있는데, 이점은 중요하다. 이 파라미터를 복사로 전달하면 (즉 const를 사용하지 않으면), 파라미터를 받는 함수 즉 StringStatus가 실행되는 동안 그 문자열에 대한 참조 카운트가 하나 더 증가하기 때문이다(웁긴이: StringStatus 함수에게 전달되는 파라미터 변수 역시 이 문자열을 참조하기 때문에 참조 카운트가 1 개 더 증가함). 이와 달리, 파라미터를 전달할 때, 참조로 전달(var) 또는 상수로 전달(const)을 사용하면 그 파라미터는 전달되는 문자열에 대한 참조 카운트를 증가시키지 않는다. 위 예제에서는 함수 안에서 이 문자열을 수정하지 않기 때문에 const 파라미터를 사용했다.

문자열의 메모리 주소를 얻기 위해 (문자열이 실제로 있는 위치를 확인하면, 서로 다른 문자열 변수들이 동일한 메모리 영역을 참조하고 있는지도 확인할 수 있음) 문자열 타입을 강제로 정수Integer 타입으로 캐스팅하는 코드를 이 함수에서 사용했다. 문자열 변수는 사실 참조reference이다. 즉 포인터pointer이다. 그 값에는 문자열 데이터 자체가 아니라 문자열이 저장된 메모리의 실제 위치가 담긴다.

문자열에서 어떤 일이 발생하는지 테스트한 코드는 다음과 같다.

```
Show('MyStr1 - ' + StringStatus(MyStr1));
Show('MyStr2 - ' + StringStatus(MyStr2));

MyStr1 [1] := 'a';
Show('두번째 문자 변경');
Show('MyStr1 - ' + StringStatus(MyStr1));
Show('MyStr2 - ' + StringStatus(MyStr2));
```

앞쪽에 있는 코드가 실행된 결과는, 두 문자열 모두 동일한 내용, 동일한 메모리 위치를 가리키고, 참조 카운트 역시 동일하게 2 이다.

```
MyStr1 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
MyStr2 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
```


코드의 두번째 블록은, 두 문자열 중 하나에서(둘 중 어느 문자열이든 상관없음) 값을 변경하고 나서 그 결과를 보여준다. 아래와 같이 변경되는 문자열의 메모리 위치가 다른 곳으로 변경된다. 즉 copy-on-write^{쓰는 시점에 복사} 기술의 효과가 나타난다.

두번째 문자 변경

```
MyStr1 - Addr: 51848300, Len: 5, Ref: 1, Val: Hallo
MyStr2 - Addr: 51837036, Len: 5, Ref: 1, Val: Hello
```

이 예제를 자유롭게 확장하고 사용해 보기 바란다. StringStatus 함수를 통해서, 긴 문자열이 다양한 상황에서 어떻게 동작하는 지를 들여다볼 수 있고, 문자열이 파라미터로 전달되거나 로컬 변수에 할당될 때, 참조^{reference}가 어떻게 달라지는지를 파악할 수 있을 것이다.

문자열String과 인코딩Encoding

앞서 살펴본 바와 같이, 오브젝트 파스칼의 문자열 타입은 유니코드 UTF-16 형식 format에 매핑 된다. 요소당 2-바이트^{byte}가 사용되며, BMP(다국어 기본 평면 ^{Basic Multilingual Plane}) 바깥에 있는 코드 포인트를 위해서는 surrogate pairs/대리 쌍을 관리한다.

그렇지만, 현실에서는 문자열을 파일에 저장, 파일에 있는 것을 메모리에 적재^{load}, 소켓 연결을 통한 전송, ANSI나 UTF-8과 같이 다른 표현을 사용하는 곳에 연결하여 텍스트 데이터 수신하기 등 다양한 상황이 있다.

파일 안 또는 메모리 안에 있는 문자열 데이터를 서로 다른 포맷^{format}(또는 인코딩^{encoding})으로 변환을 쉽게 할 수 있게 하는 TEncoding 클래스가 오브젝트 파스칼의 RTL에 있다. 이 클래스는 기타 상속된 클래스들과 함께 System.SysUtils 유닛에 정의되어 있다.

참고 오브젝트 파스칼의 RTL에는 텍스트 형식의 데이터를 읽고 쓰기 위해 사용할 수 있는 여러 편리한 클래스들 역시 들어 있다. 예를 들어, TStreamReader와 TStreamWriter 클래스는 텍스트 파일이 어떤 인코딩을 사용하든 모두 지원한다. 이 클래스들은 18장에서 소개한다.

아직 클래스^{class}와 상속^{inheritance} 소개까지는 가지 못했지만, 이 인코딩 클래스 세트는 사용하기가 매우 쉽다. 각 인코딩 별 글로벌^{global/전역} 오브젝트가 이미 있기 때문이다.

다시 말해, 각 인코딩 클래스의 오브젝트를 TEncoding 클래스 안에서 활용할 수 있다. 클래스 프로퍼티^{property}를 통해 얻으면 된다.

```
type
  TEncoding = class
  ..
  public
    class property ASCII: TEncoding read GetASCII;
    class property BigEndianUnicode: TEncoding
```



```

    read GetBigEndianUnicode;
class property Default: TEncoding read GetDefault;
class property Unicode: TEncoding read GetUnicode;
class property UTF7: TEncoding read GetUTF7;
class property UTF8: TEncoding read GetUTF8;

```

참고 Unicode 인코딩은 TUnicodeEncoding 클래스가 기반이다. 이 클래스는 문자열 타입에서 사용하는 UTF-16 LE(리틀 little 엔디안 endian) 표현을 사용한다. 반면에, BigEndianUnicode는 덜 흔한 빅 big 엔디안 endian 표현을 사용한다. "엔디안 endian"은 코드 포인트(또는 다른 데이터 구조)를 구성하는 2-바이트 byte 크기의 요소들 여러 개가 어떤 순서로 나열되는지를 표시하기 위해 사용되는 용어다. 리틀 엔디안은 가장 큰 자릿수를 나타내는 바이트가 맨 앞에 위치하고, 빅 엔디안은 가장 큰 자릿수가 담긴 바이트가 맨 뒤에 위치한다.
자세한 내용은 en.wikipedia.org/wiki/Endianness를 참조.

이번에도, 이런 클래스들을 쭉 훑어보기보다(지금 설명하기에는 다소 어려울 수도 있기 때문), 실용적인 예제에 집중한다. TEncoding 클래스에는 유니코드 문자열을 바이트 배열로 읽고, 쓰고, 또 알맞게 변환하는 메서드 method들이 있다.

TEncoding 클래스를 써서 UTF 형식을 변환하자. 이 EncodingsTest 예제는 파일 시스템을 다루지 않고, 단순하다. 아래 코드처럼 먼저 메모리에 UTF-8 문자열을 하나 생성하고 이 문자열을 UTF-16으로 변환하는 함수를 호출한다.

```

var
  Utf8String: TBytes;
  Utf16String: string;
begin
  // UTF-8 데이터 처리
  SetLength(Utf8String, 3);
  Utf8String[0] := Ord('a'); // 싱글 바이트 ANSI char < 128
  Utf8String[1] := $c9; // 더블 바이트, 뒤집어진 라틴어 a 자
  Utf8String[2] := $90;
  Utf16String := TEncoding.UTF8.GetString(Utf8String);
  Show('Unicode: ' + Utf16String);

```

결과와 다음과 같다.

```
Unicode: aø
```

이제, 변환을 더 잘 이해하고 표현들의 차이를 알 수 있도록 다음 코드를 추가했다.

```

Show('Utf8 bytes:');
for AByte in Utf8String do
  Show(AByte.ToString);

Show('Utf16 bytes:');
UniBytes := TEncoding.Unicode.GetBytes(Utf16String);
for AByte in UniBytes do
  Show(AByte.ToString);

```

위 코드는 메모리 덤프 memory dump를 만드는데, 같은 문자열이라도 UTF-8 표현과 UTF-

16 표현은 그 결과가 다르다. UTF-8은 a에 해당하는 코드 포인트에 1 바이트, æ에 해당하는 코드 포인트에 2 바이트를 사용하여 표현한다. 하지만, UTF-16은 두 코드 포인트 모두 각각 2 바이트를 사용한다. 각 표현을 바이트 단위로 끊어서 10진수 값으로 출력한 결과는 다음과 같다.

```
Utf8 bytes:
97
201
144
Utf16 bytes:
97
0
80
2
```

주목할 점이 있다. 문자를 바이트로 직접 변환 시, UTF-8인 경우 ANSI-7 문자들만 (값이 127 이하임) 올바르게 처리된다. 그보다 큰 ANSI 문자들은 직접 매핑할 수 없다. 따라서, 반드시 해당 인코딩을 사용해 변환해야 한다 (하지만, 멀티-바이트 UTF-8 문자라면 이 방식도 실패한다). 따라서 아래 두 가지 시도는 모두 틀렸다고 출력된다.

```
// 오류: char > 128 이면 사용할 수 없음
Utf8String[0] := Ord('à');
Utf16String := TEncoding.UTF8.GetString(Utf8String);
Show('틀림 - 너무 높은 ANSI: ' + Utf16String);
// 다른 변환 시도
Utf16String := TEncoding.ANSI.GetString(Utf8String);
Show('틀림 - 더블 바이트: ' + Utf16String);

// 결과
틀림 - 너무 높은 ANSI:
틀림 - 더블 바이트: àÉ
```

인코딩 클래스들을 사용하면 양방향으로 변환할 수 있다. 따라서, 아래 코드에서는, 먼저 UTF-16에서 UTF-8로 변환하고 나서, 그 UTF-8 문자열의 일부를 처리한다 (UTF-8의 특성 상, 문자의 길이가 다를 수 있다는 점을 고려하여 신중하게 수행해야 함). 그리고 나서 다시 UTF-16으로 변환한다.

```
var
  Utf8String: TBytes;
  Utf16String: string;
  I: Integer;
begin
  Utf16String := 'This is my nice string with à and æ';
  Show('시작: ' + Utf16String);

  Utf8String := TEncoding.UTF8.GetBytes(Utf16String);
  for I := 0 to High(Utf8String) do
    if Utf8String[I] = Ord('i') then
      Utf8String[I] := Ord('I');
  Utf16String := TEncoding.UTF8.GetString(Utf8String);
  Show('마지막: ' + Utf16String);
```


결과는 다음과 같다.

시작: This is my nice string with à and Æ
 마지막: ThIs Is my nIce strIng wItH à and Æ

문자열용 기타 타입들

비록 문자열 데이터 타입은 문자열을 표현할 때 가장 일반적이고 많이 쓰이는 타입이지만, 그 외에도 다양한 문자열 타입들이 오브젝트 파스칼 데스크탑 컴파일러에 있었고 지금도 여전히 존재한다. 이 타입들 중 몇몇은 모바일 애플리케이션에서도 사용할 수 있다. 또한 조금 전에 본 예제와 같이, 모바일 애플리케이션에서는 TBytes를 사용하여 1-바이트로 표현된 문자열을 직접 조작할 수도 있다.

오브젝트 파스칼을 오래전에 사용했던 개발자라면 유니코드 이전에 있었던 이러한 문자열 타입들이 사용된(또는 UTF-8을 직접 다루는) 코드가 많을 수 있다. 하지만, 지금의 애플리케이션은 유니코드를 완전하게 지원할 필요가 있다. 비록 이 언어가 UTF8String 등 일부 타입들을 제공하기 때문에 사용할 수 있겠지만, RTL 차원의 지원이 제한적이다. 권장 사항은 일반 표준 유니코드 문자열을 사용하는 것이다.

참고 오브젝트 파스칼 모바일 컴파일러에 AnsiString, UTF8String과 같은 네이티브 타입이 처음부터 빠졌다는 점에 대해 많은 논의와 비판이 있었다. 델파이 10.1 베를린에서는 UTF8String 타입과 저-수준 RawByteString 타입이 공식적으로 다시 도입되고, 이후 델파이 10.4에서는 모바일에서도 모든 데스크탑 문자열 타입을 사용할 수 있게 되었다. 다른 프로그래밍 언어를 보면, 기본 또는 고유 문자열 타입이 하나보다 더 많은 경우가 거의 없다는 점을 반드시 생각해 볼 필요가 있다. 여러 가지 문자열 타입을 사용하면 복잡해서 숙달하기가 더 어렵고, 원치 않는 부작용(예: 자동 변환 호출이 널리 사용하여 프로그램의 속도를 저하시킴)이 생길 수 있다. 게다가, 그 모든 문자열들의 여러 버전을 관리하고 처리하는 기능을 유지 관리하려면 비용이 많이 든다. 그러므로, 권장 사항은 특수한 경우를 제외하고는 표준 문자열 타입인 UnicodeString에 집중하는 것이다.

UCS4String 타입

흥미롭긴 하지만 거의 사용되지 않는 문자열 타입이 바로 UCS4String 타입이다. UCS4String 타입은 거의 모든 컴파일러에서 사용할 수 있다. 이것은 문자열을 UTF-32로 표현한 것일 뿐이며, UTF32Char 즉 4 바이트 문자들이 담기는 배열일 뿐이다. 앞서 언급했듯이, 유니코드의 모든 코드 포인트를 직접 표현하는 유일한 타입이다. 단 점은 명백하다. UCS4String 타입 문자열이 UTF-16 문자열보다 메모리 공간을 두 배나 많이 차지한다는 점이다 (참고로 이미 UTF-16 문자열은 ANSI 문자열보다 메모리 공간을 두 배나 더 많이 차지한다).

UCS4String 데이터 타입이 꼭 필요한 상황이라면 사용할 수 있겠지만, 일반적인 상황에는 사용하기에 매우 부적합하다. 게다가 copy-on-write^{쓰는 시점에 복사}를 지원하지 않으며, 이 타입을 다루도록 시스템 수준에서 제공하는 함수나 프로시저가 없다.

참고 UCS4String은 유니코드 코드 포인트와 UTF32Char가 1:1로 대응한다는 점을 보장하지만, grapheme 즉 "시각적 문자" 표현과 UTF32Char가 1:1로 대응한다는 보장은 하지 않는다.

이전의 문자열 타입들

앞서 언급했듯이, 오브젝트 파스칼 컴파일러는 오래된 기존 문자열 타입 몇 가지를 여전히 지원한다(텔파이 10.4부터는 모든 대상 플랫폼에서 이 타입들을 사용할 수 있다). 이 구형 문자열 타입으로는 다음과 같은 것들이 있다.

- ShortString 타입: 파스칼 언어의 원래 문자열 타입에 대응하는 타입이다. 이 문자열에는 최대 255자까지 들어갈 수 있다. 이 문자열 안에 들어가는 요소는 ANSIChar 타입이다.
- ANSIString 타입: 가변-길이^{variable-length} 문자열에 해당한다. 이 문자열은 동적으로^{dynamically} 할당되고 참조 카운팅이 적용되고, copy-on-write^{쓰는 시점에 복사} 기술이 사용된다. 이 문자열의 크기는 거의 무제한이다 (20억자까지 들어간다!). 이 문자열 역시 ANSIChar 타입이 기반이며 모바일 컴파일러에서도 사용할 수 있다. 비록 ANSI 표현이 윈도우에 특화된 표현이고 일부 특수 문자들이 플랫폼에 따라 다르게 처리될 수 있지만 말이다.
- WideString 타입: 2-바이트 유니코드 문자열과 표현 방식 면에서 비슷하다. 그 기반은 char 타입이다. 하지만, 표준 문자열 타입과 달리 copy-on-write^{쓰는 시점에 복사} 기술이 적용되지 않는다. 그래서 메모리 할당 면에서 효율성이 떨어진다. 이 타입이 굳이 언어에 들어간 이유는, 마이크로소프트사의 COM 아키텍처의 문자열 관리와의 호환성 때문이다.
- UTF8String 타입: 가변 문자 길이인 UTF-8 타입을 기반으로 하는 문자열이다. 앞서 언급했듯이, 이 타입에 대한 런타임 라이브러리의 지원이 거의 없다.
- RawByteString 타입: 코드 페이지가 지정되지 않은 문자 배열이다. 이 타입은 시스템에서 문자 변환을 수행하지 않는다(따라서 논리적으로는 TBytes 구조를 닮았다. 하지만, 현재 바이트 배열에는 없는 몇 가지 문자열 직접 연산을 할 수 있다). 라이브러리 바깥에서는 거의 이 데이터 타입을 사용해선 안 된다.
- 1-바이트 문자열을 개발자가 정의하여 특정 ISO 코드 페이지와 연결할 수 있는 문자열 구성 메커니즘: 유니코드 이전의 잔재이다.

다시 말하지만, 이 모든 문자열 타입들을 데스크탑 컴파일러에서 쓸 수 있다. 하지만 이전 버전과의 호환성을 위해서만 제공된다. 목표는 가능하면 유니코드, TEncoding 및 기타 최신 문자열 관리 기술을 사용하는 것이다.

파트 II OOP와 오브젝트 파스칼

현대 프로그래밍 언어는 객체 지향 프로그래밍(OOP, Object-Oriented Programming) 패러다임을 어떤 형태로든 지원한다. 그 중 대부분이 클래스-기반 OOP를 사용하며, 다음 세 가지 기본 개념이 그 바탕이다.

- 클래스 Class: 공개 public 인터페이스와 비공개 private 데이터 구조를 사용하는 데이터 타입이다. 그리고 자신의 인스턴스 instance 즉 오브젝트 object를 통해 캡슐화 encapsulation를 구현한다.
- 클래스 확장성 즉 상속 Class extensibility or inheritance: 데이터 타입에 새 기능을 추가하거나 확장하고 싶을 때, 기존 데이터 타입을 손대지 않고도 실현할 수 있는 능력이다.
- 다형성 즉 나중에 바인딩하기 Polymorphism or late binding: 서로 다른 클래스의 오브젝트들을 통일된 uniform 인터페이스 interface를 통해서 참조할 수 있는 능력이다. 그 오브젝트는 자신의 타입에서 정의된 방식으로 작동한다.

참고 IO, JavaScript, Lua, Rebol과 같은 언어는 프로토타입 기반 객체-지향 패러다임을 사용한다. 즉 오브젝트가 생성되는 방식에 따라 오브젝트를 생성할 때 클래스가 아니라 다른 오브젝트를 기반으로 할 수 있다. 이 언어들은 상속 inheritance의 한 형태를 제공하지만 클래스가 아니라 다른 오브젝트로부터 상속을 받는다. 또한 조금 다른 방식이긴 하지만 동적 타입 지정 dynamic typing을 사용하여 다형성 polymorphism을 구현한다.

객체 지향 프로그래밍에 대해 잘 몰라도, 오브젝트 파스칼 애플리케이션을 작성할 수 있다. 새 폼(form(양식))을 만들고, 새 컴포넌트 component들을 추가하고, 이벤트 event들을 다루면, IDE가 자동으로 관련 코드 대부분을 준비해 준다. 하지만 이 언어와 그 구현 implementation을 상세히 알면 이 개발 시스템이 하는 작업을 정확하게 이해하고 언어를 완전히 마스터하는 데 도움이 된다.

OOP를 이해하면 애플리케이션 그리고 심지어 전체 라이브러리 안에 여러분의 복잡한 아키텍처를 만드는데 도움이 된다. 그리고 이 개발 환경에서 제공하는 컴포넌트들을 담아 내고 확장하는 데에도 도움이 된다.

이 책의 두 번째 파트는 객체-지향 프로그래밍(OOP, [Object-Oriented Programming](#)) 기술의 핵심에 집중한다. 이 파트의 목적은 OOP의 기본 개념을 가르치고, 오브젝트 파스칼에서 OOP를 구현하는 방법을 자세히 설명하는 것이다. 그리고 다른 OOP 언어와 비교를 통해 이해를 돕는다.

파트 II 요약

7장: 오브젝트 [Objects](#)

8장: 상속 [Inheritance](#)

9장: 예외 다루기 [Handling Exceptions](#)

10장: 프로퍼티와 이벤트 [Properties and Events](#)

11장: 인터페이스 [Interfaces](#)

12장: 클래스 조작하기 [Manipulating Classes](#)

13장: 오브젝트와 메모리 [Objects and Memory](#)

07: 오브젝트 Objects

객체 지향 프로그래밍(OOP, Object-Oriented Programming)에 대한 자세한 지식이 없어도, 이 장에서 OOP의 핵심 개념을 소개하므로 이해할 수 있다. 그런데, OOP에 이미 능통하다면, 내용을 비교적 빠르게 볼 수 있을 것이다. 오브젝트 파스칼 언어의 세부 사항에 주로 집중하면서 이미 알고 있는 다른 언어와 비교해 볼 수 있다.

오브젝트 파스칼의 OOP 지원은 C#, Java 등의 언어와 비슷한 점이 많다. 또한 C++ 등 기타 정적 [static](#)이고 타입이 강력하게 지정되는 [strongly-typed](#) 언어와도 닮은 점이 있다. 이와 달리, 동적 [dynamic](#) 언어들은 OOP를 다르게 해석하는 경향이 있는데, 그 이유는 타입 시스템을 보다 느슨하고 유연한 방식으로 처리하기 때문이다.

참고 C#과 오브젝트 파스칼에 비슷한 개념이 많은 이유는 설계자가 같아서이다. 앤더스 헤즐스버그 [Anders Hejlsberg](#)는 터보 파스칼 컴파일러의 원 저자이며 또한 델파이의 오브젝트 파스칼 첫 버전을 개발했다. 이후 마이크로소프트로 옮겨서 C#(그 이후에는 JavaScript에서 파생한 TypeScript까지)을 설계했다. 오브젝트 파스칼 언어의 역사에 대한 자세한 내용은 부록 A에 있다.

클래스와 오브젝트 소개 Introducing Classes and Objects

클래스 [class](#)와 *오브젝트* [object](#)는 오브젝트 파스칼 및 기타 OOP 언어에서 흔히 사용되는 용어들이다. 그런데, 이 용어들은 종종 잘못 사용되기도 한다. 그러니 먼저 정의를 분명하게 확인하여 같은 바탕에서 출발하기로 하자.

- *클래스*는 사용자 정의 데이터 타입이다. 클래스에는 하나의 상태 [state](#)(즉 표현 [representation](#)) 그리고 여러 개의 연산 [operation](#)(즉 동작 [behavior](#))이 정의된다. 즉, 클래스는 여러 내부 데이터 [internal data](#)와 여러 메서드 [method](#)(프로시저 또는 함수의 모습)를 가진다.

- 일반적으로, 클래스 하나에는 비슷한 여러 오브젝트들이 가지는 특성 [characteristics](#)과 동작 [behavior](#)이 설명되어 있다. 그런데, 일부 특수 목적 클래스들은 오브젝트를 오직 하나만 가진다.
- *오브젝트*는 클래스의 인스턴스 [instance](#) 즉, 클래스로 정의된 데이터 타입의 변수 [variable](#)다. 오브젝트는 실제 엔티티 [actual entity](#)이므로, 프로그램이 실행되면, 메모리의 한 부분을 차지하고 그 안에 자신 내부의 표현을 담아 둔다.
- 오브젝트와 클래스 사이의 관계 [relationship](#)는 변수와 그 데이터 타입과의 관계와 같다. 다만, 오브젝트 변수는 인스턴스 [instance](#)라는 특별한 이름으로 부른다.

역사 OOP에서 사용하는 용어들은 객체 지향 모델을 채택한 최초의 몇 가지 언어들(Smalltalk 등)로 거슬러 올라간다. 그런데, 원래 용어들 중 대부분은 이후에 사라졌다. 그 이유는 사람들이 절차적 언어 [procedural language](#)들에서 사용하던 용어들을 더 선호했기 때문이다. 지금도 비록 클래스 [class](#), 오브젝트 [object](#) 등은 여전히 일반적으로 사용되지만, 아마 여러분은, 수신자 [receiver](#) (오브젝트)에게 메시지를 보내기 [sending a message](#)라는 원래 용어보다, 메서드를 불러내기 [invoking a method](#)라는 용어를 더 자주 들었을 것이다. OOP 전문 용어에 대해 자세히 살펴보고 시간이 흐르면서 어떻게 진화했는지 알아보는 것도 꽤 재미있을 것이다. 하지만, 이 책에 담기에는 너무 내용이 많다.

클래스 정의 [The Definition of a Class](#)

오브젝트 파스칼에서는 다음 구문 [syntax](#) 을 사용하여 새 클래스 데이터 타입(TDate)을 정의할 수 있다. 그 정의에는 로컬 데이터 필드(Month, Day, Year)와 메서드(SetValue, LeapYear)를 넣는다.

```
type
TDate = class
  FMonth, FDay, FYear: Integer;
  procedure SetValue(M, D, Y: Integer);
  function LeapYear: Boolean;
end;
```

참고 위 코드와 비슷한 구조를 레코드에서 이미 보았다. 레코드는 정의 [definition](#) 면에서 클래스와 매우 비슷하다. 하지만, 메모리 관리와 기타 영역에서는 다르다. 이 장의 뒤에서 자세히 살펴본다. 역사적으로 오브젝트 파스칼에서 이 구문 [syntax](#)이 먼저 사용된 곳은 클래스다. 레코드에는 적용된 것은 보다 나중이다.

오브젝트 파스칼 규약은 클래스 이름을 정할 때 *T*를 앞에 붙인다. 다른 모든 타입들도 마찬가지다 (실제로 *T*는 *Type*의 약자다). 이는 그저 규약일 뿐, 컴파일러에게 *T*는 다른 글자와 마찬가지로 그저 글자들 중 하나일 뿐이다. 하지만, 매우 일반적이기 때문에 이 규약을 따르면 다른 프로그래머들이 더 쉽게 그 코드를 이해할 것이다.

다른 언어와 달리, 오브젝트 파스칼에서는, 클래스 정의 [class definition](#)에 메서드의 실제 구현 [actual implementation](#)(또는 정의 [definition](#))이 들어가지 않는다. 그저 메서드의 서명 [signature](#)(즉 선언 [declaration](#))만 들어간다. 그래서, 클래스 코드가 더 간결하고 가독성이 훨씬 더 높다.

팁 메서드의 실제 구현으로 이동하려면 우리의 시간을 더 많이 쓰게 될 것 같지만, IDE 에디터 [editor\(편집기\)](#)에서 Ctrl+Shift+위 그리고 Ctrl+Shift+아래 키 조합을 사용하면 메서드 선언에서 구현으로 이동하거나 그 반대로 이동한다. 또한 메서드 정의 코드의 골격을 IDE 에디터가 만들어 주도록 할 수도 있다. 클래스 정의 [class definition](#)를 작성한 후에, 클래스 완성 [Class Completion](#)을 사용하면 된다(커서가 클래스 정의 안에 있는 상태에서 Ctrl+Shift+C 키 누르기).

또한, 명심할 점이 있다. 클래스의 정의(그 필드들과 메서드들까지)를 작성할 수 있을 뿐만 아니라, 여러분은 선언 [declaration](#)을 작성할 수도 있다. 클래스 이름만 적어도 된다.

type

```
TMyDate = class;
```

이런 선언이 필요한 이유는 클래스 두 개가 서로를 참조해야 하는 경우도 있기 때문이다. 오브젝트 파스칼에서는 심볼 [symbol](#)이 정의되기 전에는 그 심볼을 사용할 수 없다. 따라서 아직-정의되지-않은 클래스를 참조하려면 그 클래스를 선언해야 한다. 아래 코드 조각을 통해 그 구문만 파악하기 바란다. 타당한 예는 아니다.

type

```
THusband = class;
TWife = class
  FHusband: THusband;
end;
THusband = class
  FWife: TWife;
end;
```

이와 비슷한 상호 참조 [cross-reference](#)를 실제 코드에서 볼 수 있다. 따라서, 이 구문을 알아 두는 것이 중요하다. 메서드에서도 그랬지만, 클래스 역시 그 정의는 반드시 자신이 선언된 바로 그 유닛 안에 완전하게 작성되어야 한다는 점에 유의하자.

다른 OOP 언어들 안에 있는 클래스 [Classes in Other OOP Languages](#)

비교를 위해, 위 TDate 클래스를 C#과 Java 로 작성한 예문을 보자 (아래 예문은 마침 두 언어 모두 구문이 같다). 아래 코드는 더 알맞은 명명 규칙 [naming rule](#)을 사용했다. 단, 메서드 구현 코드는 생략했다.

// C# 언어와 Java 언어

```
class Date
{
    int month;
    int day;
    int year;

    void setValue(int m, int d, int y)
    {
        // 코드
    }
}
```



```

    bool leapYear()
    {
        // 코드
    }
}

```

Java 와 C#은 메서드의 코드가 클래스 정의 안에 들어간다. 오브젝트 파스칼은, 이와 달리, 클래스 안에 메서드 선언만 들어간다. 그 메서드의 정의는 그 클래스가 선언된 그 유닛 파일 안의 implementation 구현 구역에 들어간다. 즉, 오브젝트 파스칼에서 클래스 하나는 항상 하나의 유닛 안에 완전하게 정의되어야 한다 (물론 유닛 하나에 클래스 여러 개가 들어갈 수는 있다). 이와 달리, C++는, 메서드와 그 구현이 별도로 분리된다는 점이 오브젝트 파스칼과 같다. 하지만, 클래스 정의를 담고 있는 헤더 파일과 해당 메서드의 코드를 담고 있는 구현 파일이 엄격하게 일치하지는 않는다.

같은 클래스를 C++로 작성하면 다음과 같다:

```

// C++ 언어
class Date
{
    int month;
    int day;
    int year;

    void setValue(int m, int d, int y);
    BOOL leapYear();
}

```

클래스 메서드 The Class Methods

레코드와 마찬가지로, 여러분은 메서드의 코드를 정의할 때 어느 클래스에 속하는 것인지를 명시해야 한다 (아래 예에서는 TDate 클래스에 속한다). 메서드 앞에 그 클래스의 이름과 점을 붙이면 된다. 아래 코드와 같다.

```

procedure TDate.SetValue(M, D, Y: Integer);
begin
    FMonth := M;
    FDay := D;
    FYear := Y;
end

function TDate.LeapYear: Boolean;
begin
    // IsLeapYear를 호출한다 (SysUtils.pas 안에 있음)
    Result := IsLeapYear(FYear);
end;

```

대부분의 다른 OOP 언어들은 메서드 **method** 를 모두 함수 **function** 로 정의한다. 하지만, 오브젝트 파스칼의 메서드 정의는 반환 값의 존재 여부에 따라, 프로시저 **procedure** 와 함수 **function** 를 명확히 구분해서 적는다. 비록 C++에서는 메서드 구현 정의를 별도로 작성하는 것까지는 같지만, 메서드 코드는 다르다. 아래 예문을 보자.


```
// C++ 메서드
void Date::setValue(int m, int d, int y)
{
    month = m;
    day = d;
    year = y;
};
```

오브젝트 생성하기 Creating an Object

널리 사용되는 다른 언어들과 비교를 마쳤으니, 이제 오브젝트 파스칼로 돌아가서, 여러분이 어떻게 클래스를 어떻게 사용할 수 있는지 살펴보자.

클래스가 정의되면, 이제 그 타입으로 된 오브젝트를 생성하여 사용할 수 있다. 아래 코드를 보자 (아래 코드를 포함하여, 이 절의 코드는 모두 Dates1 예제에서 발췌함).

```
var
    ADay: TDate;
begin
    // 생성한다
    ADay := TDate.Create;
    // 사용한다
    ADay.SetValue(1, 1, 2020);
    if ADay.LeapYear then
        Show( '윤년: ' + ADay.Year.ToString);
```

사용된 표기법 [notation](#) 이 그다지 특별하지 않다. 하지만 강력하다. 우리는 클래스 안에 복잡한 함수(예: LeapYear)를 작성할 수 있다. 그리고 모든 TDate 오브젝트에서 그 오브젝트의 값에 접근할 수 있다. 마치 원시 데이터 타입 [primitive data type](#) 을 다루는 것과 같다. 주목할 점이 있다. ADay.LeapYear 와 ADay.Year는 표현식 [expression](#) 이 서로 비슷하다. 하지만, 앞의 것은 함수 호출이고, 뒤의 것은 데이터에 대한 직접 접근이다. 프로퍼티 [property](#) 에 접근할 때 사용하는 표기법 역시 이와 똑같다. 프로퍼티에 대해서는 10 장에서 살펴보겠다.

참고 C 언어 구문을 기반으로 하는 대부분의 프로그래밍 언어는 파라미터가 없는 메서드를 호출할 때에도 괄호가 필요하다. 즉 ADay.LeapYear() 같이 작성해야 한다. 이 구문 [syntax](#) 은 오브젝트 파스칼에서도 합법적 [legal](#) 이다. 하지만 이렇게 쓰는 경우는 거의 없다. 파라미터가 없는 메서드를 호출할 때 괄호를 적지 않는 것이 일반적이라는 점은 다른 언어들과 차이점이다. 다른 언어는 함수나 메서드를 괄호 없이 참조하면 그 함수의 주소를 반환한다. 4장에서 "프로시저 타입" 절에서 살펴봤듯이, 오브젝트 파스칼에서는 함수를 호출할 때와 주소를 읽을 때 표기법이 똑같다. 그저 문맥 [context](#) 에 의해 구분된다.

위 코드 조각의 결과 출력은 특별한 것이 없다.

```
| 윤년: 2020
```


다시 한번 다른 프로그래밍 언어와 비교해보자. 위 오브젝트를 생성하는 코드를 다른 언어로 작성하는 아래와 같다.

```
// C#과 Java 언어 (오브젝트 참조 모델, object reference model)
Date aDay = new Date();

// C++ 언어(두 가지 스타일이 있음)
Date aDay; // 로컬 할당(Local allocation)
Date* aDay = new Date(); // "수동" 참조("Manual" reference)
```

오브젝트 참조 모델 The Object Reference Model

C++ 등 일부 OOP 언어에서는, 클래스 타입의 변수 `variable` 를 선언하면 그 클래스의 인스턴스 `instance` 가 생성된다 (오브젝트 파스칼의 레코드도 이와 거의 같게 동작한다). 로컬 `local` 오브젝트는 필요한 메모리를 스택 `stack` 에서 가져온다. 그리고 함수가 끝나면 잡아 놓았던 메모리는 풀린다. 그렇기는 해도, 대부분의 경우, 포인터 `pointer` 와 참조를 명시적으로 사용해야만 오브젝트의 수명을 보다 유연하게 관리할 수 있기 때문에 더 복잡하게 된다.

이와 달리, 오브젝트 파스칼 언어는 *오브젝트 참조 모델*을 기반으로 한다. 이 점은 Java 와 C#도 똑같다. 이 모델에서는 클래스 타입인 변수에는 오브젝트 데이터 (위 예에서는 연, 월, 일)의 실제 값을 담지 않는다. 그저 그 오브젝트의 실제 데이터들이 담겨 있는 메모리의 위치를 가리키는 참조 즉 포인터만 담는다.

참고 이 언어 초기에 컴파일러 팀이 내린 최고의 설계 결정 중 하나라는 생각이 든다. 그 시절에는 프로그래밍 언어들 사이에서 이 모델은 그다지 일반적이지 않았다 (그 당시는 Java는 사용할 수 없었고, C#은 존재하지 않았던 시절이었다).

그렇기 때문에 오브젝트 참조 모델을 채택한 언어에서는, 오브젝트가 자동으로 초기화 `initialization` 되지 않기 때문에, 직접 개발자가 명시적으로 오브젝트를 생성하고 그것을 변수에 할당해야 한다. 즉, 클래스 타입 변수 선언은 메모리에 오브젝트를 생성하지 않는다. 그저 오브젝트를 가리킬 참조 `reference` 가 들어 갈 메모리 위치를 예약할 뿐이다. 오브젝트 인스턴스가 생성되려면 손수 `manually` 명시적으로 `explicitly` 작성한 코드가 있어야 한다. 직접 정의한 클래스 타입이라면 반드시 그렇다 (하지만, 오브젝트 파스칼에서는 개발자가 품 위에 올려 놓은 컴포넌트인 경우, 그 인스턴스는 런타임 라이브러리에 의해 자동으로 구축된다).

오브젝트 파스칼에서, 오브젝트의 인스턴스를 생성하려면, (생성자 `constructor` 인) 특별한 Create 메서드 또는 클래스 자체에 정의되어 있는 사용자 정의 생성자 `custom constructor` 를 호출한다. 코드는 다음과 같다.

```
| ADay := TDate.Create;
```


보다시피, 이 생성자는 오브젝트(변수)가 아니라 클래스(타입)에 적용된다. 즉 우리는 해당 타입의 새 인스턴스를 생성할 것을 클래스에게 요청한다. 그 결과로 우리는 새 오브젝트를 얻게 되고 (대체로 위와 같이) 이것을 변수에 할당한다.

이 Create 메서드는 어디에서 왔을까? 이것은 TObject 클래스의 생성자다. 모든 클래스들은 TObject로부터 상속을 받는다 (다음 장에서 이 주제를 다룬다). 그런데, 클래스에 사용자 정의 생성자를 추가하는 것은 매우 혼하다 (이 장 후반부에 설명함).

오브젝트 폐기하기 Disposing of Objects

오브젝트 참조 모델을 사용하는 언어에서는 (오브젝트를 사용하기 전에) 오브젝트를 생성 *create* 하는 방법이 있어야 한다. 또한 그 오브젝트가 (더 이상 필요하지 않을 때) 오브젝트가 차지하던 메모리를 해제 *release* 할 수 있는 방법도 있어야 한다. 오브젝트를 폐기 *dispose* 하지 않으면, 필요가 없어진 오브젝트들로 메모리가 꽉 차게 되어, 메모리 누수 *memory leak* 라는 문제가 발생한다. 이런 문제를 풀기 위해 C#, Java 와 같은 언어는 가비지 컬렉션 *garbage collection*(쓰레기 수거)을 도입하고 있다. 이 언어들은 가상 실행 환경 (즉 가상 머신)을 기반으로 하기 때문이다. 이 방식은 개발자의 삶을 편하게 해주지만, 성능과 관련된 몇 가지 복잡한 문제를 안고 있다. 흥미로운 주제이지만 오브젝트 파스칼과는 아무 관련이 없기 때문에 이 책에서 다루지는 않는다.

오브젝트 파스칼에서 오브젝트의 메모리를 해제하려면 대체로 특별한 Free 메서드를 호출한다. (다시 말하지만, TObject 의 메서드이므로 모든 클래스에서 쓸 수 있다). Free 는 오브젝트를 메모리에서 제거한다. 소멸자 *destructor*(깨끗이 지우는 특별한 코드가 있음)를 호출하기 때문이다. 따라서 위의 코드 조각을 완성하면 다음과 같다.

```
var
  ADay: TDate;
begin
  // 생성한다
  ADay := TDate.Create;

  // 사용한다 (사용하는 코드는 지금 생략)

  // 메모리를 해제한다(Free the memory)
  ADay.Free;
end;
```

지금까지 표준적인 접근 방식에 대해서 설명했다. 그런데, 컴포넌트 라이브러리에는 오브젝트 소유권 *ownership* 등의 개념이 추가되어 있어서, 수동 메모리 관리로 인한 영향을 엄청나게 줄여주기 때문에, 실제로는 다루기가 비교적 간단한 문제다.

참고 뒤에 살펴보겠지만, 오브젝트를 참조할 때 인터페이스 *interface*를 사용하면, 컴파일러는 자동 참조 카운팅(ARC, Automatic Reference Counting) 메모리 관리 방식을 채택한다. 델파이 모바일 컴파일러는, 일반 클래스 타입 변수에 대해서도 몇 년 동안 ARC 방식을 사용했었다. 하지만, 10.4 버전(시드니라고도 알려짐)부터 메모리 관리 모델이 통합되어서, 이제는 모든 타겟 플랫폼에서 고전적인 *classic* 데스크탑 델파이 메모리 관리 방식이 채택된다.

메모리 관리에 대해서는 알아야 할 것들이 훨씬 더 많다. 하지만, 중요한 주제이고 내용이 간단하지 않기 때문에, 지금은 소개만 짧게 한다. 13 장은 이 주제만 집중해서 다룬다. 우리가 사용할 수 있는 다양한 기법들도 13 장에서 자세히 볼 수 있다.

“Nil”이란 무엇인가? What is “Nil”?

앞서 언급했듯이, 변수는 (클래스의) 오브젝트를 참조할 수 있다. 하지만 그 오브젝트가 아직 초기화되지 않았을 수 있다. 또는 이미 참조되고 있었지만 그 오브젝트가 이제는 쓸 수 없는 상태가 되었을 수도 있다. 이런 경우에 nil 을 사용할 수 있다. nil 은 상수 값이며 그 변수가 아직 어떤 오브젝트에도 할당되지 않았음을 나타낸다. 다른 프로그래밍 언어에서는 null 기호를 사용해서 이 개념을 표현한다.

클래스 타입인 변수에 값이 없는 경우, 아래 방법으로 초기화하면 된다.

```
Aday := nil;
```

그 변수에 할당된 오브젝트가 있는지 확인하려면 다음 표현식 중 하나를 쓰면 된다.

```
if Aday <> nil then ...
if Assigned(Aday) then ...
```

오브젝트에 nil 을 할당해서 오브젝트를 메모리에서 제거하려는 실수를 하지 말자. 오브젝트 참조를 nil 로 지정하는 것과 그것을 해제 free 하는 것은 별개의 작업이다. 그렇기 때문에, 오브젝트 해제하기와 오브젝트 참조를 nil 로 지정하기 두 작업을 모두 해야 하는 경우가 자주 있다. 특별한 용도를 가진 프로시저인 FreeAndNil 를 호출하면 두 작업을 한꺼번에 처리할 수 있다. 다시 말하지만, 더 많은 정보와 실제 예제들은 (메모리 관리에 집중한) 13 장에 나온다.

레코드와 클래스를 메모리 관점에서 비교 Records vs. Classes in Memory

앞서 언급했듯이, 레코드와 오브젝트의 주요 차이점 중 하나는 메모리 모델과 관련이 있다. 레코드 타입 변수는 로컬 메모리를 사용하며, 함수에게 파라미터로 전달될 때는 (기본적으로 by default) 값으로 전달 pass by value 된다. 그리고 할당을 하면 "값 복사 copy-by-value" 동작이 수행된다. 이와 달리, 클래스 타입 변수는 동적 메모리인 힙 heap 에 할당되고, 참조로 전달 pass by reference 된다. 그리고, 할당 동작은 "참조 복사 copy-by-reference"이다 (따라서 같은 메모리를 가리키도록 참조를 복사하는 것이지 오브젝트의 실제 데이터를 복사하는 것이 아니다).

참고 이와 같은 메모리 관리 차이로 인해, 레코드에는 상속 inheritance 과 다형성 polymorphism 이라는 클래스의 두 가지 특성(다음 장의 주제다)이 없다.

클래스의 필드 및 메서드에 private 접근 지정자 access specifier 를 명시하면 그 클래스를 선언한 유닛 (그 소스 코드 파일) 바깥에서는 접근할 수 없다.

예를 들어, 레코드 변수를 스택에 선언하면, 즉시 사용할 수 있다. 레코드의 생성자를 호출하지 않아도 된다(단, 사용자 정의 매니지드 레코드 *custom managed records*가 아닌 경우). 즉, 레코드 변수는 일반 *regular* 오브젝트보다 메모리 매니저 입장에서 부담이 더 적고 더 효율적이다. 레코드는 동적 메모리 *dynamic memory* 관리에 참여하지 않기 때문이다. 작고 단순한 데이터 구조가 필요할 때 오브젝트 대신 레코드를 사용하는 주요 이유가 바로 이것이다.

레코드와 오브젝트는 파라미터로 전달될 때, 기본 *default* 동작이 서로 다르다. 레코드는 그 메모리 블록 전체가 복사된다(레코드의 데이터 전체 복사). 하지만, 오브젝트는 오브젝트를 가리키는 참조가 복사된다(데이터는 복사되지 않음). 물론, 레코드 파라미터에 `var` 또는 `const`를 사용해서, 레코드 타입 파라미터를 전달하는 동작을 변경하고 전체 복사를 피할 수도 있다.

비공개, 보호, 공개 *Private, Protected, and Public*

필드와 메서드에 `strict private` 접근 지정자를 붙이면, 그 클래스의 메서드가 아니면 전혀 접근할 수 없다. 같은 유닛 안이라 할지라도 다른 클래스들의 메서드에서는 접근이 안된다. 이는 대부분의 다른 OOP 언어에서 `private` 키워드가 하는 동작과 같다.

클래스에는 데이터 필드와 메서드가 개수에 상관없이 얼마든지 많아도 된다. 하지만, 좋은 객체-지향 방식이 되기 위해서는, 데이터가 자신이 들어가 있는 그 클래스 안에 숨어야 *hidden* 한다. 즉 *캡슐화* *encapsulate* 되어야 한다.

예를 들어, 날짜 오브젝트에 접근한다고 가정하자. 그 오브젝트의 값에 직접 접근해서 변경하는 것은 타당하지 않다. 날짜의 값을 직접 변경한다면, 실제로 2월 30일처럼 유효하지 않은 날짜가 되기도 한다. 메서드를 통해서 오브젝트의 내부 표현에 접근하도록 하면, 이런 오류 상황을 만드는 위험을 제한할 수 있다. 그 메서드 안에서, 유효한 날짜인지 미리 점검할 수 있고, 값이 유효하지 않은 경우 변경을 거부하도록 구현해 놓으면 된다. 올바른 캡슐화가 특히 중요한 이유가 있다. 미래에 클래스의 내부 구현 변경을 얼마든지 자유롭게 할 수 있게 된다.

캡슐화라는 개념은 꽤 단순하다. 클래스를 "블랙 박스"라고 여기면 된다. 바깥에는 아주 작은 부분만 노출한다. 이런 부분을 *클래스 인터페이스* *class interface*라고 부른다. 프로그램의 다른 곳에서 그 클래스의 오브젝트를 접근하고 사용할 수 있도록 허용하는 부분이다. 하지만, 오브젝트의 코드 대부분은 사용자가 볼 수 없도록 숨겨져 있다. 바깥에서는 오브젝트 안에 무슨 데이터가 있는지 거의 알지 못할 뿐만 아니라 오브젝트의 데이터에 직접 접근할 방법도 대체로 없다. 제공되는 메서드를 사용해야만 그 오브젝트의 데이터에 접근하거나 다룰 수 있다.

객체-지향에서 캡슐화를 하는 방식은 비공개 멤버와 보호되는 멤버를 사용하는 것이다. 이를 통해 정보 숨기기 [information hiding](#)이라는 프로그래밍의 클래식한 목표를 달성한다.

오브젝트 파스칼에서 접근 (즉 가시성 [visibility](#)) 지정자 [specifier](#) 는 기본적으로 `private` [비공개](#), `protected` [보호됨](#), `public` [공개](#) 등 다섯 가지다. 여섯 번째 접근 지정자인 `published` 는 10 장에서 설명한다. 다섯 가지 기본 접근 지정자는 다음과 같다.

- `public` 접근 지정자는 그 필드와 메서드가 자유롭게 접근할 수 있음을 명시한다. 이 필드와 메서드는 자신이 정의된 클래스뿐만 아니라 프로그램의 어느 곳에서도 접근할 수 있다.
- `protected`와 `strict protected` 접근 지정자는 그 필드와 메서드의 가시성이 제한됨을 명시한다. `protected`인 필드와 메서드는 오직 자신이 정의된 클래스 그리고 그 클래스의 자손(즉, 서브-클래스 [subclass](#))들에서만 접근할 수 있다. `strict` 제어자는 그 클래스와 같은 유닛에 있는 다른 클래스에서 접근할 수 있는지를 결정한다. `strict` 키워드는 다음 장의 "보호된 필드와 캡슐화" 부분에서 다시 살펴본다.

대체로, 클래스의 필드 [field](#) 는 `private` 또는 `strict private` 여야 하고, 메서드 [method](#) 는 주로 `public` 이다. 하지만, 항상 그런 건 아니다. 메서드가 `private` 또는 `protected` 일 수 있다. 부분적인 연산을 내부적으로만 수행하는 경우다. 필드가 `protected` 일 수도 있다. 그 필드의 정의가 앞으로 전혀 변경되지 않을 것이 확실하고 그 필드를 파생된 클래스에서 직접 다루기를 원하는 경우다 (다음 장에서 설명한다). 하지만, 이런 것들이 권장되는 상황은 거의 없다.

일반 규칙 상, 여러분은 `public` 필드를 무조건 피해야 한다. 그리고 그 데이터에 바로 접근하는 다른 방법을 외부에 노출하는 것이 일반적으로 좋다. 프로퍼티 [property](#) 를 사용하면 된다(10 장에서 자세히 살펴본다). 프로퍼티는 다른 OOP 언어의 캡슐화 메커니즘을 확장한 것으로 오브젝트 파스칼에서 매우 중요하다.

앞서 언급했듯이, `private` 접근 지정자는 클래스의 특정 멤버를 그 클래스가 선언된 유닛 밖에 있는 코드에서 접근하지 못하도록 제한할 뿐이다. 즉, `private` 필드는 같은 유닛 안에 있는 다른 클래스가 접근하는 것으로부터 보호받지 못한다. 이 경우에는 `strict private` 으로 표시해야 보호받을 수 있다. 일반적으로 이렇게 하는 것이 좋다.

참고 C++ 언어에는 `friend class` [친구 클래스](#)라는 개념이 있다. 이 클래스끼리는 상대방의 `private` 데이터에 접근할 수 있다. 이 용어를 빗대어 표현하자면, 오브젝트 파스칼에서는 같은 유닛에 있는 클래스들은 모두 자동으로 `friend class`로 간주된다고 말할 수 있다.

비공개 데이터 예시 [An Example of Private Data](#)

접근 지정자들을 사용해 캡슐화를 구현하는 예제를 보자. 접근 지정자를 써서 `TDate` 클래스의 새 버전을 만들었다. 아래와 같다.


```

type
  TDate = class
    private
      Month, Day, Year: Integer;

    public
      procedure SetValue(M, D, Y: Integer);
      function LeapYear: Boolean;
      function GetText: string;
      procedure Increase;
    end;

```

새 버전은, 필드가 `private` 으로 선언되었다. 새 메서드들도 몇 개 생겼다. 새 메서드인 `GetText` 는 함수다. 날짜 필드의 값을 문자열 `string` 로 반환한다. 다른 함수들도 추가하고 싶을 수 있을 것이다. 예를 들어, `GetDay`, `GetMonth`, `GetYear` 와 같은 함수들을 추가하여 해당 `private` 데이터를 반환하고 싶을 수 있다. 하지만, 데이터를 직접-접근하는 그런 함수가 항상 필요한 건 아니다. 모든 필드들이 직접 접근하는 함수에 의해 노출된다면, 캡슐화 `encapsulation` 가 낮아지고, 추상화 `abstraction` 가 약해진다. 나중에 클래스의 내부 구현을 변경하기도 더 어려워진다. 단순 접근을 위한 함수를 제공할 때는, 그것이 클래스의 인터페이스 한 부분이 되어 하는 논리적 이유가 있어야만 한다. 그저 필드에 맞추는 함수는 좋지 않다.

`Increase` 프로시저도 새로 추가된 메서드다. 이것은 날짜 필드를 하루 증가시킨다. 결코 단순한 동작이 아니다. 날짜의 개수는 달마다 다르고 윤년과 평년에 따라서도 다르다는 점을 고려해야 한다. 이 코드를 보다 쉽게 작성하기 위해, 이 클래스의 내부 구현을 바꾸기로 결정했다. 그 데이터 필드를 오브젝트 파스칼의 `TDateTime` 타입을 사용해 구현했다. 변경된 실제 클래스는 아래 코드와 같다 (`Dates2` 예제에서 발췌함)

```

type
  TDate = class
    private
      FDate: TDateTime;

    public
      procedure SetValue(M, D, Y: Integer);
      function LeapYear: Boolean;
      function GetText: string;
      procedure Increase;
    end;

```

위 클래스는 오직 `private` 부분만 변경되었다. 그러므로, 이 클래스를 사용하는 그 무엇도 프로그램에서 변경할 필요가 없다. 이것이 캡슐화의 장점이다!

참고 위 새 버전에서 필드 `field` 는 식별자가 문자 "F"로 시작한다. 이것은 오브젝트 파스칼에서 상당히 일반적인 규칙이다. 이 책에서도 대체로 이 규칙을 따른다.

이 클래스의 메서드들의 코드는 아래와 같다. 이 메서드들은 시스템 함수 `system function` 를 활용해 이 클래스의 내부 구조에 날짜를 또는 그 반대로 매핑 `mapping` 한다.


```

procedure TDate.SetValue(M, D, Y: Integer);
begin
    FDate := EncodeDate(Y, M, D);
end;
function TDate.GetText: string;
begin
    Result := DateToStr(FDate);
end;

procedure TDate.Increase;
begin
    FDate := FDate + 1;
end;

function TDate.LeapYear: Boolean;
begin
    // IsLeapYear(SysUtils 유닛에 있음)호출, YearOf(DateUtils 유닛에 있음)호출
    Result := IsLeapYear(YearOf(FDate));
end;

```

아래와 같이, 이 클래스를 사용하는 코드는 이제 year 필드의 값을 참조하지 못한다. 오직 TDate 클래스의 메서드에서 허용하는 오브젝트의 정보만을 받을 수 있다.

```

var
    ADay: TDate;
begin
    // 생성하기
    ADay := TDate.Create;

    // 사용하기
    ADay.SetValue(1, 1, 2020);
    ADay.Increase;

    if ADay.LeapYear then
        Show( '윤년: ' + ADay.GetText);

    // 메모리 해제(Free the memory)
    ADay.Free;

```

결과는 이전과 크게 다르지 않다.

윤년: 1/2/2020

참고로, 출력 결과는 다를 수 있다. 날짜 표현 형식은 시스템의 로케일 [locale\(지역\)](#) 설정에 의해 정해지기 때문이다.

캡슐화와 폼 Encapsulation and Forms

캡슐화의 핵심 개념 중 하나는 프로그램에서 사용하는 글로벌 [global/전역](#) 변수의 개수를 줄이는 것이다. 글로벌 변수는 프로그램의 모든 부분에서 액세스할 수 있기 때문에 글로벌 변수를 변경하면 프로그램 전체에 영향을 미친다. 이와 달리, 클래스의 필드를

변경하면, 그 클래스의 메서드만, 그 중에서도 그 필드를 참조하는 메서드만 코드를 변경하면 된다. 다른 것은 손 댈 필요가 없다. 따라서, 정보 숨김 *information hiding* 이란 *변경을 캡슐화 encapsulating changes* 하는 것을 의미한다고 볼 수 있다.

실제 예시를 통해 이 개념을 명확히 이해해 보자. 프로그램에 폼 *form* 이 여러 개 있는 경우, 유닛의 인터페이스 부분에 글로벌 변수로 선언된 데이터는 프로그램 안에 있는 모든 폼들이 사용할 수 있다.

```
var
  Form1: TForm1;
  NClicks: Integer;
```

이 방식은 잘 작동한다. 하지만 두 가지 문제가 있다. 첫째, 데이터(NClicks)가 폼의 특정 인스턴스에 연결되지 못하고, 프로그램 전체에 연결되어 있다. 같은 타입으로 폼을 두 개 생성하면, 둘 다 이 데이터를 공유한다. 같은 타입으로 된 폼들이 저마다 고유한 데이터 복사본을 갖도록 하려면 이 데이터를 폼 클래스 안에 추가하는 것이 유일한 해결책이다.

```
type
  TForm1 = class(TForm)
  public
    FNClicks: Integer;
  end;
```

두 번째 문제는, 이 데이터를 글로벌 변수 또는 폼의 *public* 필드로 정의하는 경우, 나중에 구현을 변경하기 어렵다는 점이다. 이 데이터를 사용하는 코드에 영향을 주기 때문이다. 그 대신, (다른 폼에서는 이 필드의 현재 값을 읽기만 한다면) 이 데이터를 *private* 으로 선언하고, 필드의 값을 읽을 수 있도록 메서드를 만들어 제공하면 된다.

```
type
  TForm1 = class(TForm)
    // 컴포넌트들과 이벤트 핸들러들이 들어가는 곳
  public
    function GetClicks: Integer;
  private
    FNClicks: Integer;
  end;

function TForm1.GetClicks: Integer;
begin
  Result := FNClicks;
end;
```

더 좋은 해결책은 이 폼에 프로퍼티 *property* 를 추가하는 것이다 (10 장에서 살펴본다). 위 예문은 ClicksCount 예제에서 발췌했다. 간단히 소개하면, 이 예제 프로젝트에는 폼 안에 버튼 *button* 두 개와 레이블 *label* 한 개가 있고 나머지 공간 대부분은 비어 있다. 사용자가 폼을 클릭(또는 손가락으로 탭)을 하면 그럴 때 마다, 클릭 횟수가 증가하고, 레이블에는 증가된 새 값이 업데이트 된다.


```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
begin
  Inc(FNClicks);
  Label1.Text := FNClicks.ToString;
end;

```

그림 7.1 은 애플리케이션의 모습이다. 이 폼에 있는 첫 번째 버튼은 같은 타입으로 새 폼을 만든다. 그리고 두 번째 버튼은 폼을 종료한다 (그래서 포커스를 이전 폼으로 다시 옮길 수 있다).

실행해보면, 같은 폼 타입에서 나온 인스턴스이지만, 각자 자신들만의 클릭 횟수를 가진다. 이 두 메서드는 코드가 아래와 같다.

```

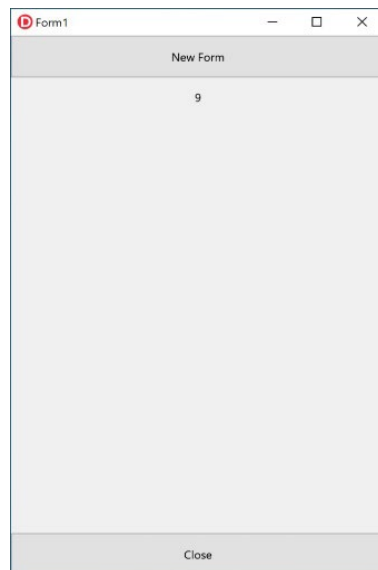
procedure TForm1.Button1Click(Sender: TObject);
var
  NewForm: TForm1;
begin
  NewForm := TForm1.Create(Application);
  NewForm.Show;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

```

그림 7.1:

ClickCount 폼 예제는 폼을 클릭(또는 탭)한 횟수를 보여준다.
(폼의 private 데이터를 사용하여 추적함)



Self 식별자 The Self Identifier

메서드가 프로시저나 함수와 매우 비슷하다는 것을 보았다. 실제 차이점은 메서드에 암시적 파라미터 [implicit parameter](#) 하나가 더 있다는 것이다. 그 파라미터는 현재 오브젝트, 즉 메서드가 적용되는 오브젝트를 가리키는 참조다. 메서드 안에서 이 파라미터 (현재 오브젝트)를 참조하려면 `Self` 식별자를 사용하면 된다 (앞서 5 장의 'Self: 레코드 뒤에 숨겨진 마법' 부분에서 이미 언급되었음)

이 숨겨진 추가 [extra hidden](#) 파라미터는 동일한 클래스로 오브젝트를 여러 개를 생성하는 경우에 요긴하다. 메서드를 적용할 때 이것이 있어야 동일한 클래스 타입 오브젝트들 중에서 다른 오브젝트에는 영향을 미치지 않고, 오직 그 메서드가 적용되어야 하는 오브젝트의 데이터에서만 동작할 수 있다.

참고 `Self` 식별자의 개념과 구현은 레코드와 클래스가 매우 비슷하다. 역사적으로, `Self`가 처음으로 도입된 곳은 클래스다. 이후 (레코드가 메서드를 가질 수 있게 되면서) 레코드로 확장되었다.

예를 들어, 앞에서 본 `TDate` 클래스의 `SetValue` 메서드에는 `Month`, `Year`, `Day` 만 적혀 있지만, 현재 오브젝트의 필드들을 참조한다. 이 부분은 실제로 아래와 같이 작성해도 마찬가지다.

```
Self.FMonth := M;
Self.FDay := D;
```

위 코드는 실제로 오브젝트 파스칼 컴파일러가 코드를 번역하는 방식이다. 하지만, 우리가 코드를 작성할 때는 `Self` 를 붙일 필요가 없다. `Self` 식별자는 컴파일러에서 사용하는 이 언어의 기반 구조 [fundamental language construct](#) 이다. 하지만, 가끔 프로그래머들이 `Self` 식별자를 사용하여, 이름 충돌을 해소하고 코드를 더 읽기 쉽게 만들기도 한다.

참고 C++, Java, C#, JavaScript 언어에는 이와 비슷한 기능이 있는데, `this`라는 키워드를 기반으로 한다. 그리고, JavaScript에서는 오브젝트 필드를 참조하는 메서드에서 이 키워드를 사용하는 것이 (C++, C#, Java와 달리) 필수이다.

`Self` 에 관해서 정말 알아야 하는 것은 메서드 [method](#) 호출과 일반 서브루틴 [generic subroutine](#) 호출이 기술적 구현 면에서 다르다는 점이다. 다시 말하지만, 메서드에는 `Self` 라는 숨겨진 파라미터가 추가로 있다. 하지만, 지금 우리는 `Self` 가 어떻게 작동하는지 알 필요가 없다. 이 모든 것들은 드러나지 않는 곳에서 수행되기 때문이다.

알아야 할 두 번째 중요한 점은 우리는 `Self` 를 명시적으로 적으면, 현재 오브젝트를 통째로 참조할 수 있다는 것이다. 예를 들어, 다른 함수에게 현재 오브젝트를 통째로 파라미터로 전달할 수 있다.

컴포넌트를 동적으로 생성하기 Creating Components Dynamically

방금 언급한 사항에 대한 사례로, 폼의 메서드들 중 어느 한 곳에서 명시적으로 현재 폼을 참조해야 하는 경우에 `Self` 식별자가 종종 사용된다.

전형적인 예를 보자. 실행 중에 컴포넌트를 생성하는 경우인데, `Create` 생성자에게 우리는 그 컴포넌트의 소유주 `owner` 를 전달해 주어야 한다. 그리고 전달한 소유자와 똑같은 값을 그 컴포넌트의 `Parent` 부모 프로퍼티에 할당해야 한다. 이 두 경우 모두, 즉 파라미터로 전달되고, `Parent` 프로퍼티의 값으로 할당되어야 하는 것은 현재의 폼 오브젝트다. 이때 가장 좋은 방법은 `Self` 식별자를 사용하는 것이다.

참고 컴포넌트의 소유권 `ownership`은 두 오브젝트 간의 수명 관계 그리고 메모리 관리 관계를 표현한다. 컴포넌트의 소유주가 해제 `free` 되면 그 컴포넌트도 해제된다. 부모 관계 `Parenthood`는 시각적 컨트롤이 자식 컨트롤을 자신의 표면 안에 담는 것을 의미한다.

`CreateComps` 예제는 이런 종류의 코드를 보여준다. 이 애플리케이션에는 간단한 폼 하나만 있다. 그 폼 안에는 컴포넌트가 전혀 없다. 그런데, 이 폼에는 `OnMouseDown` 이벤트의 핸들러가 있다. 이것은 클릭 위치를 파라미터로 받아서 해당 위치에 버튼 컴포넌트를 생성한다.

참고 이벤트 핸들러는 특별한 메서드다(10장에서 다룬다). 이 책에서 사용한 버튼의 `OnClick` 이벤트 핸들러 역시 같은 종류이다.

이 (이벤트 핸들러) 메서드의 코드는 다음과 같다.

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  Btn: TButton;
begin
  Btn := TButton.Create(Self);
  Btn.Parent := Self;
  Btn.Position.X := X;
  Btn.Position.Y := Y;
  Btn.Height := 35;
  Btn.Width := 135;
  Btn.Text := Format('At %d, %d', [X, Y]);
end;
```

이 이벤트 핸들러를 컴파일하려면 `uses` 문에 `StdCtrls` 유닛이 추가되어야 한다.

이 코드의 효과는 마우스 클릭 위치에 버튼을 생성한다. 그리고 그 버튼의 캡션에는 마우스를 클릭한 정확한 위치가 표시된다. 그림 7.2와 같다. (이 프로젝트에서는 FMX 모바일 프리뷰를 비활성화했으므로, 버튼이 네이티브 윈도우 스타일로 표시된다. 더 선명하게 버튼을 보여주기 위해서다). 위 코드에서 `Self` 식별자가 사용된 곳들을 눈여겨보자. `Create` 메서드의 파라미터와 `Parent` 프로퍼티의 값으로 사용되고 있다.

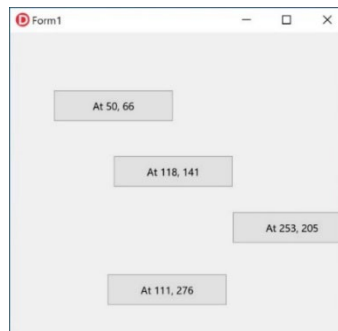
위 같은 프로시저를 작성할 때, 아마 Self가 아니라 Form1 변수를 사용하고 싶을 수 있다. 위 예제에서는, (비록 좋은 코딩 관행은 아니지만) 실제로 아무런 차이가 없다. 하지만, 폼의 인스턴스가 여러 개라면, Form1을 사용하는 건 정말 큰 실수다.

실제로, Form1 변수는 이 타입으로 생성된 폼들 중 하나를 가리킨다 (대체로 첫 번째 폼), 만약 위 프로시저에서 Form1 변수 사용했다면, 같은 폼 타입으로 인스턴스를 두 개 생성하는 경우, 두 번째 폼 위에서 클릭을 해도 버튼은 언제나 첫 번째 폼 위에 나타난다. 생성된 버튼의 Owner^{소유주}와 Parent 모두를 (사용자가 클릭한 폼이 아니라) Form1으로 명시했기 때문이다.

일반적으로, 메서드를 작성하면서 그 안에서 현재 오브젝트를 참조할 때, 이처럼 같은 클래스로 된 특정 인스턴스를 참조하는 것은 정말 나쁜 OOP 코딩 스타일이다.

그림 7.2:

CreateComps 예제를
모바일 장비에서
실행한 결과



생성자Constructors

위 코드에서 클래스의 오브젝트를 생성(즉 오브젝트를 위해 메모리를 할당)하기 위해 Create 메서드를 호출했다. 이것은 생성자 [constructor](#) 이다. 특별한 메서드이며, 클래스에 적용하여 해당 클래스의 새 인스턴스를 메모리에 할당한다.

```
ADay := TDate.Create;
```

생성자는 인스턴스를 반환한다. 우리는 그 인스턴스를 변수에 할당해서 저장해 두고 사용할 수 있다. 오브젝트를 생성하는 시점에는 해당 메모리가 초기화 [initialized](#) 된다. 새 인스턴스의 모든 데이터는 0 (또는 nil, 빈 문자열 또는, 데이터 타입에 맞는 알맞은 "기본 [default](#)" 값)으로 설정된다.

인스턴스 데이터가 0 이 아닌 값을 가지고 시작하도록 하려면 (특히, 0 이 기본 [default](#) 값으로 의미가 없는 경우라면) 사용자 정의 생성자 [custom constructor](#) 를 작성해야 한다. 새 생성자의 이름은 Create 라고 지어도 되고, 다른 이름을 써도 된다. 생성자 메서드가 되도록 결정하는 것은 이름이 아니라 constructor 키워드의 사용 여부다.

참고 다시 말해서, 많은 OOP 언어에서는 생성자 이름이 반드시 클래스 자체의 이름이어야 하지만, 오브젝트 파스칼은 이름 지정 생성자(named constructor)를 지원한다. 따라서 우리는 똑같은 파라미터를 가지는 생성자를 둘 이상 만들 수 있다 (Create 심볼을 오버로드하는 것 외에도 - 오버로드는 다음 절에서 설명한다). 이 언어에는 다른 OOP 언어에서 보기 힘든 매우 특별한 특징이 하나 더 있다. 바로 생성자가 가상(virtual)이 될 수도 있다는 점이다. 몇몇 예제를 통해서 이 멋진 기능에서 비롯되는 결과를 이 책의 뒷부분에서 설명한다. 그리고 가상 메서드(virtual method) 개념은 다음 장에서 소개한다.

사용자 정의 생성자를 클래스에 추가하는 경우, 주된 이유는 오브젝트의 데이터를 초기화(initialize) 하기 위해서다. 오브젝트를 생성할 때 초기화를 하지 않고 생성하면 나중에 메서드를 호출할 때, 이상하게 동작하거나 런타임 에러가 발생할 수 있다. 이러한 에러가 나타날 때까지 기다리지 말고, 예방 기술(preventive technique)을 써야 한다. 즉, 처음부터 방지하는 것이 옳다. 이러한 기법 중 하나는 생성자를 항상 일관되게 사용하여 오브젝트의 데이터를 초기화 하는 것이다. 예를 들어, 오브젝트를 생성한 다음 반드시 TDate 클래스의 SetValue 프로시저를 호출하는 것이다. 또다른 대안은, 사용자 정의 생성자를 제공하는 것이다. 그 생성자 안에서 오브젝트를 생성하고 초기값(initial value)도 제공한다.

```
constructor TDate.Create;
begin
    FDate := Date; // 오늘 날짜
end;

constructor TDate.CreateFromValues(M, D, Y: Integer);
begin
    FDate := SetValue(M, D, Y);
end;
```

위 생성자들을 사용하는 방법은 아래와 같다 (Date3 예제에 있는 버튼들 각각에서 반영하는 코드에서 발췌함).

```
Aday1 := TDate.Create;
Aday2 := TDate.CreateFromValues(12, 25, 2015);
```

비록, 일반적으로, 생성자에 어떤 이름이든 사용할 수 있지만 Create 이외의 이름을 사용하면 기반 클래스인 TObject 의 Create 생성자 역시 계속 사용할 수 있게 된다는 점을 주의하자. 우리가 개발한 코드를 다른 사람들이 사용하도록 배포한다고 생각해 보자. 코드를 받은 프로그래머는 기본(default) Create 생성자를 호출할 수 있을 텐데, 그러면 우리가 제공한 초기화 코드를 그냥 넘어가게 된다.

Create 이름으로 생성자를 정의하면서 파라미터를 가지도록 하면 (또는 파라미터가 없도록 하면, 위 코드 참조), 기본(default) 정의된 Create 를 새로 만든 정의가 대체하고 새 정의가 사용되도록 강제할 수 있다.

클래스가 사용자 정의 생성자를 가질 수 있는 것과 마찬가지로, 클래스는 사용자 정의 소멸자(custom destructor), 즉 destructor 키워드로 선언되는 메서드를 가질 수 있다. 그

이름은 항상 Destroy 이다. 소멸자 메서드는 오브젝트가 파괴되기 전에 리소스 정리 작업을 수행할 수 있다. 하지만, 사용자 정의 소멸자가 필요 없는 경우가 많다.

생성자 호출이 오브젝트에 메모리를 할당 allocate 하는 것처럼, 소멸자 호출은 메모리를 해제 free 한다. 사용자 정의 소멸자는 오직 그 오브젝트가 생성자에서 획득했거나 살아 있는 동안에 획득한 다른 리소스(또다른 오브젝트 등)가 있는 경우에만 필요하다.

기본 default Create 생성자와 달리, 기본 default Destroy 소멸자는 가상 virtual 메서드다. 그리고 개발자는 가상 소멸자를 오버라이드 override(재정의) 하는 것이 좋다 (가상 메서드는 다음 장에서 다룬다).

그 이유는 소멸자를 직접 호출하여 오브젝트를 해제하는 것보다, 오브젝트 파스칼 프로그래밍의 공통 관행에 따라 TObject 에 있는 특별한 Free 메서드를 호출하는 것이 좋기 때문이다. 이 특별한 Free 메서드는 오직 오브젝트가 존재하는 경우에만 (즉 그 오브젝트가 nil 이 아닌 경우에만) Destroy 를 호출한다. 따라서 소멸자를 다른 이름으로 정의하면, Free 가 그 소멸자를 호출되지 않는다. 다시 말하지만, 이 주제는 13 장에서 메모리 관리에 집중할 때 자세히 살펴보겠다.

참고 다음 장에서 다루겠지만, Destroy는 가상 메서드다. 우리는 상속된 클래스에서 Destroy를 새로 정의하고 override 키워드를 붙여서 기반 base Destroy 메서드를 대체할 수 있다. 그건 그렇고, 정적 static 메서드에서 가상 virtual 메서드를 호출하는 것은 매우 흔한 프로그래밍 스타일이다. 소위 템플릿 패턴 template pattern 이라고 한다. 소멸자 안에는 오직 리소스를 깨끗이 청소하는 코드만 넣는 것이 일반적이다. 더 복잡한 동작은 피하도록 노력하라. 예외 exception 가 발생할 가능성이 있거나 시간이 오래 걸리는 연산이 있으면 좋지 않다. 또한 프로그램 종료 시점에는 많은 소멸자들이 호출되기 때문이다. 가능한 빠른 속도를 지키는 것이 좋다.

로컬 클래스 데이터를 관리하기 (생성자와 소멸자 사용)

이 책의 뒷부분에서 좀 더 복잡한 시나리오를 다루겠지만, 지금 간단한 리소스 보호 사례를 통해 생성자 constructor 와 소멸자 destructor 가 어떻게 사용되는지 보자. 소멸자가 사용되는 가장 흔한 경우다.

다음과 같은 구조의 클래스가 있다고 가정하자 (이것 역시 Date3 예제의 일부임).

```
type
  TPerson = class
  private
    FName: string;
    FBirthdate: TDate;
  public
    constructor Create(const Name: string);
    destructor Destroy; override;
    // 일반 메서드들이 나열되는 곳
    function Info: string;
  end;
```


이 클래스는 또 다른 오브젝트에 대한 참조를 가지고 있다. `FBirthdate` 라는 내부 오브젝트다. `TPerson` 클래스의 인스턴스가 생성되면, 그 내부(즉 자식) 오브젝트 역시 생성되어야 하고, 그 인스턴스가 소멸되면 그 내부(즉 자식) 오브젝트 역시 폐기되어야 한다.

다음은 생성자 코드 작성과 소멸자를 오버라이드 `override` 하는 코드다. 내부 오브젝트가 존재한다고 믿고 작성되는 내부 메서드 코드도 있다.

```
constructor TPerson.Create(const Name: string);
begin
    FName := Name;
    FBirthdate := TDate.Create;
end;

destructor TPerson.Destroy;
begin
    FBirthdate.Free;
    inherited;
end;

function TPerson.Info: string;
begin
    Result := FName + ' ' + FBirthdate.GetText;
end;
```

참고 위에서, 소멸자 정의에 붙인 `override` 키워드와 그 정의 안의 `inherited` 키워드를 이해하려면, 다음 장까지 기다려야 한다. 지금은 간단히 보자. `override`는 기반 `Destroy` 소멸자를 대체하는 새 정의가 클래스 안에 있음을 나타낸다. `inherited`는 그 기반 `base` 클래스의 소멸자를 호출한다. 또한, `override`는 메서드 선언에는 사용되지만, 메서드 구현 코드에는 사용되지 않는다.

이제, 우리는 외부에서 이 클래스의 오브젝트를 사용할 수 있다. 아래 코드와 같다. `TPerson` 클래스의 내부 오브젝트는 `TPerson` 오브젝트가 생성될 때 올바르게 생성되고, `TPerson` 오브젝트가 파괴될 때 함께 파괴한다 (이것 역시 `Date3` 예제에서 발췌함).

```
var
    Person: TPerson;
begin
    Person := TPerson.Create('홍길동');
    // 이 클래스와 그 내부 오브젝트를 사용한다
    Show(Person.Info);
    Person.Free;
end;
```

오버로드되는 메서드와 오버로드되는 생성자 Overloaded Methods and Constructors

오브젝트 파스칼은 함수와 메서드의 오버로드 `overload` 를 지원한다. 파라미터가 다르다면 이름이 같은 메서드를 여러 개 가져도 된다. 오버로드가 작동하는 방식은 우리가 이미 보았다. 글로벌 `global` 함수와 프로시저에 적용되는 규칙은 메서드에도 그대로 적용된다. 컴파일러는 파라미터를 보고, 메서드의 어떤 버전을 호출하려고 하는지 알아낸다.

다시 말하지만, 오버로드에는 두 가지 기본 규칙이 있다.

- 메서드의 모든 버전마다 뒤에 `overload` 키워드가 붙어야 한다.
- 반드시 파라미터의 개수 또는/그리고 타입이 달라야 한다. 반환 타입 `return type`으로는 함수를 구분할 수 없다.

만약, 클래스의 모든 메서드에 오버로드가 적용되는 경우가 있다면, 특히 생성자가 여기에 해당할 것이다. 왜냐하면 생성자를 여러 개 가질 수 있는데, 그 이름이 모두 (외우기 쉽도록) `Create` 일 것이기 때문이다.

역사 역사적으로, 오버로딩은 C++에서 특별히 생성자를 여러 개 쓸 수 있게 하려고 추가되었다. C++는 생성자의 이름이 모두 같아야(그 클래스의 이름) 하기 때문이다. 이 기능은 오브젝트 파스칼에서 불필요하다고 볼 수 있었다. 생성자들에 서로 다른 이름을 붙이면 되기 때문이다. 하지만, 어쨌든 이 언어에도 이 기능이 추가되었고, 많은 다른 상황에서도 유용하게 쓰이고 있다.

오버로딩 예제를 보자. `TDate` 클래스의 `SetValue` 메서드가 두 개의 버전을 가지도록 추가했다.

```
type
  TDate = class
  public
    procedure SetValue(Month, Day, Year: Integer); overload;
    procedure SetValue(NewDate: TDateTime); overload;

  procedure TDate.SetValue(Month, Day, Year: Integer);
  begin
    FDate := EncodeDate(Year, Month, Day);
  end;

  procedure TDate.SetValue(NewDate: TDateTime);
  begin
    FDate := NewDate;
  end;
```

그런 다음, 아래와 같이 `Create` 생성자를 두 개 추가했다. 하나는 파라미터가 없다. 이것은 기본 `default` 생성자를 숨긴다 `hide`. 그리고 다른 하나는 초기화 값을 파라미터로 받는다. 파라미터가 없는 생성자는 오늘 날짜를 기본 값으로 사용한다.

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create(Month, Day, Year: Integer); overload;

  constructor TDate.Create(Month, Day, Year: Integer);
  begin
    FDate := EncodeDate(Year, Month, Day);
  end;

  constructor TDate.Create;
  begin
    FDate := Date; // 오늘 날짜
  end;
```


생성자가 두 개이므로, 새 TDate 오브젝트를 정의하는 방법도 두 가지다.

```
var
  Day1, Day2: TDate;
begin
  Day1 := TDate.Create(2020, 12, 25);
  Day2 := TDate.Create; // 오늘 날짜
```

위 코드는 Dates4 예제에서 발췌했다.

TDate 클래스 전체 The Complete TDate Class

이 장 전체에서, 소스 코드를 나누어 TDate 클래스의 다양한 버전들의 보여 주었다. 첫 버전은 정수 3 개를 사용하여 연, 월, 일을 클래스 안에 저장했다. 두 번째 버전은 TDateTime (RTL 에서 제공함) 타입으로 된 필드 하나를 사용했다. 아래 코드는 TDate 클래스를 정의한 유닛의 전체 인터페이스 부분이다.

```
unit Dates;

interface

type
  TDate = class
  private
    FDate: TDateTime;
  public
    constructor Create; overload;
    constructor Create(Month, Day, Year: Integer); overload;
    procedure SetValue(Month, Day, Year: Integer); overload;
    procedure SetValue(NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase(NumberOfDays: Integer = 1);
    procedure Decrease(NumberOfDays: Integer = 1);
    function GetText: string;
end;
```

위에 새로 들어간 메서드인 Increase 와 Decrease(둘 다 기본값이 지정된 파라미터를 사용함)의 목적은 쉽게 이해할 것이다. 이 메서드를 파라미터 없이 호출하면 필드의 날짜 값을 다음 날짜 또는 이전 날짜로 변경한다. 만약 파라미터인 NumberOfDays 가 함께 전달되면, 그 숫자만큼 앞 또는 뒤에 있는 날짜로 변경한다.

```
procedure TDate.Increase(NumberOfDays: Integer = 1);
begin
  FDate := FDate + NumberOfDays;
end;
```

GetText 메서드는 날짜 필드의 값을 표현하는 문자열을 반환하는데, 그 문자열에는 날짜 형식이 반영된다. 날짜를 문자열로 변환하기 위해 DateToStr 함수를 사용하기 때문이다.

```
function TDate.GetText: string;
```

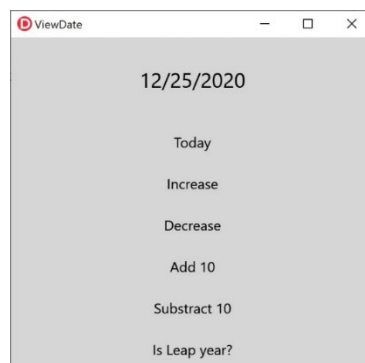


```
begin
  GetText := DateToStr(FDate);
end;
```

이미 앞에서 메서드 대부분을 살펴봤으니 전체 목록을 나열하지 않겠다. 전체 코드는 ViewDates 예제에 있다. 이 예제의 폼은 이 책에 있는 다른 것들보다 조금 더 복잡한 모습이다. 날짜를 표시하는 레이블 `Label` 하나와 버튼 여섯(6) 개가 있다. 각 버튼들은 날짜를 변경할 때 사용된다. 이 예제를 실행하면 나타나는 메인 폼 `main form` 은 아래 그림 7.3 과 같다. 레이블 컴포넌트는 더 잘 보이도록 하기 위해, 큰 글꼴을 지정하고, 너비를 폼의 너비에 맞추고, Alignment 프로퍼티를 `taCenter` 로 지정하고 (굵긴이: `TextSettings.HorzAlign = Center`), `AutoSize` 프로퍼티를 `False` 로 지정했다.

그림 7.3:

ViewDates 애플리케이션을
기동 `start-up` 하면 나타나는 결과



프로그램이 시작되는 코드는 폼의 `OnCreate` 이벤트 핸들러 안에 있다. 아래와 같이, 폼이 생성되는 이벤트에 대응하는 이 메서드는 `TDate` 클래스의 인스턴스를 생성하고, 생성된 오브젝트를 초기화하고, 레이블의 `Text` 에 그 날짜를 보여준다 (그림 7.3).

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  ADay := TDate.Create;
  LabelDate.Text := ADay.GetText;
end;
```

`ADay` 는 이 폼의 클래스인 `TDateForm` 안에 있는 `private` `비공개` 필드이다. 참고로, 개발 환경에서 폼의 `Name` 프로퍼티를 `DateForm` 으로 바꾸면, 클래스 이름도 새 이름에 맞게 자동으로 변경된다.

위와 같이, 날짜 오브젝트는 폼이 생성될 때 생성된다 (앞에서 본 `TPerson` 클래스와 그 하위 오브젝트인 날짜 사이의 관계와 같다). 그리고 폼이 파괴되면 함께 파괴될 수 있도록 하는 아래 코드가 있다.

```
procedure TDateForm.FormDestroy(Sender: TObject);
begin
  ADay.Free;
end;
```


사용자가 버튼 여섯 개 중 하나를 클릭하면, 거기에 대응하는 메서드를 통해 ADay 오브젝트에 변경을 반영한다. 그리고 변경된 새 날짜를 레이블에 표현한다.

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
    ADay.SetValue(Today);
    LabelDate.Text := ADay.GetText;
end;
```

위 메서드는 다른 방법으로 작성할 수 있다. 현재 오브젝트를 파괴하고 오브젝트를 새로 만들어도 같은 효과가 있다.

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
    ADay.Free;
    ADay := TDate.Create;
    LabelDate.Text := ADay.GetText;
end;
```

그런데, 여기에서, 위 방식은 그다지 좋지 않다 (새 오브젝트 생성과 기존 오브젝트 파괴는 부담 [overhead](#) 이 크다. 여기에서는 그저 오브젝트의 값만 바꾸면 되기 때문에 굳이 부담이 큰 방식을 선택할 이유가 없다). 하지만, 이 코드를 통해 오브젝트 파스칼 기술 몇 가지를 설명하려고 한다. 이 코드에서 새 오브젝트를 할당하기 전에 기존 오브젝트 파괴를 먼저 한다는 점을 가장 눈여겨보아야 한다. 할당 [assignment](#) 연산은 여기에서 참조 [reference](#) 를 바꾸는 동작일 뿐이므로 (비록 그 오브젝트를 참조하는 포인터 [pointer](#) 가 없더라도) 참조되고 있던 기존 오브젝트는 메모리에 그대로 남게 되기 때문이다. 오브젝트 하나를 다른 오브젝트에게 할당하는 코드는, 컴파일러에 의해 새로 할당되는 오브젝트를 가리키는 참조가 새 오브젝트 참조에 들어갈 뿐이다.

다른 부수적인 문제 하나는 오브젝트에서 데이터를 복사하여 다른 오브젝트에 넣는 방법이다. 위 경우는 매우 간단하다. 왜냐하면, 필드가 하나뿐이고 그것을 초기화하는 메서드도 하나이기 때문이다. 일반적으로, 기존 오브젝트 안에 있는 데이터를 변경하고 싶으면, 필드 각각을 복사하거나 또는 내부 데이터 전체를 복사하는 작업을 하는 특정 메서드를 제공해야 한다. 일부 클래스들은 Assign 메서드를 제공한다. Assign 메서드는 깊은-복사 [deep-copy](#) 작업을 수행한다.

참고 더 정확히 말하자면, 런타임 라이브러리 안에서 TPersistent를 상속하는 모든 클래스에는 Assign 메서드가 있다. 하지만 TComponent를 상속하는 클래스들은 대부분 이 메서드를 구현하지 않고 있어서 이 메서드를 호출하면 예외가 발생한다. 그 이유는 런타임 라이브러리에서 지원하는 스트리밍 메커니즘, 그리고 TPersistent 타입의 프로퍼티에 대한 지원과 관련 있다. 하지만, 이 책에서 자세히 설명하기에는 너무 복잡하다.

중첩된 타입과 중첩된 상수 Nested Types and Nested Constants

오브젝트 파스칼은 인터페이스 [interface](#) 구역에서 새 클래스를 선언할 수 있다. 그러면, 그 새 클래스를 프로그램의 다른 유닛들이 참조할 수 있다. 또한, 구현 [implementation](#) 구역에서도 새 클래스를 선언할 수 있다. 그러면, 오직 같은 유닛 안에 있는 다른 클래스들의 메서드, 또는 그 유닛 안에서 해당 클래스 정의보다 뒤에 구현된 글로벌 루틴에서만 그 새 클래스를 접근할 수 있다.

또한, 클래스(또는 다른 데이터 타입)를 다른 클래스 안에 선언할 수 있는 기능도 그 이후에 추가되었다. 클래스의 다른 멤버들이 그렇듯이, 중첩된 타입에도 가시성 제한(예: `private` [비공개](#) 또는 `protected` [보호됨](#))을 지정할 수 있다. 중첩된 타입과 관련된 예는 동일한 클래스에서 사용하는 열거 [enumeration](#) 와 구현-지원 클래스 등이 있다.

이와 관련된 구문 [syntax](#) 를 사용하면, 중첩된 상수 즉 클래스에 소속된 상수를 정의할 수 있다 (다시 말하지만, `private` 이면 오직 내부에서만 사용할 수 있고, `public` 이면 프로그램의 나머지에서도 사용할 수 있다). 예를 들어, 아래 코드는 중첩된 클래스를 선언하고 있다 (NestedTypes 예제의 NestedClass 유닛에서 발췌함).

```
type
  TOne = class
    private
      FSomeData: Integer;
    public
      // 중첩된 상수 (Nested constant)
      const Foo = 12;
      // 중첩된 타입 (Nested type)
      type
        TInside = class
          type TInsideInside = class
            procedure Two;
          end;
          public
            procedure InsideHello;
          private
            FMsg: string;
            FInsIns: TInsideInside;
          end;
          public
            procedure Hello;
          end;

        procedure TOne.Hello;
      var
        Ins: TInside;
      begin
        Ins := TInside.Create;
        Ins.FMsg := '안녕';
        Ins.InsideHello;
        Show('상수(Constant): ' + IntToStr(Foo));
        Ins.Free;
      end;
```



```

procedure TOne.TInside.InsideHello;
begin
  FMsg := '새 메시지';
  Show( '내부(Internal) 호출' );
  if not Assigned(FInsIns) then
    FInsIns:= TInsideInside.Create;
  FInsIns.Two;
end;

procedure TOne.TInside.TInsideInside.Two;
begin
  Show( '중첩되고 또 중첩된 클래스의 메서드입니다' );
end;

```

(위 코드에 표시된 대로) 중첩된 클래스는 클래스 안에서 직접 사용할 수 있다. 또는 클래스 바깥에서 사용될 수도 있다 (단, 공개 구역에 선언된 경우). 이때는 전체 이름 *fully qualified name* 즉 TOne.TInside 처럼 적어주어야 한다. 중첩된 클래스의 메서드를 정의할 때에도 *클래스의 전체 이름*이 사용된다. 위 경우에는 TOne.TInside (옮긴이: TOne.TInside.InsideHello 가 맞는 것으로 생각됨)가 그것이다. 중첩된 클래스를 담고 있는 클래스는 클래스 타입을 선언한 뒤에 즉시 그것을 자신의 필드로 쓸 수 있다 (NestedClass 예제의 코드 참조).

우리가 만든 중첩된 클래스를 담고 있는 클래스를 사용하는 코드는 아래와 같다.

```

var
  One: TOne;
begin
  One := TOne.Create;
  One.Hello;
  One.Free;

```

위 코드의 결과는 아래와 같다.

```

내부(Internal) 호출
중첩되고 또 중첩된 클래스의 메서드입니다
상수(Constant): 12

```

오브젝트 파스칼 언어에서 중첩 클래스를 사용해서 얻는 이점은 무엇일까? Java 에서는 이 개념을 주로 이벤트 핸들러 델리게이트 *event handler delegate* 를 구현하는데 사용한다. C#은 클래스를 유닛 안에 숨길 수 없는 언어이기 때문에 이 개념이 의미가 있다. 오브젝트 파스칼에서는, 사적으로 사용하는 *private* 다른 클래스 (또는 내부 *inner* 클래스) 타입을 필드로 가질 수 있고, 그 클래스를 글로벌 *global/전역* 네임스페이스 *namespace* (이름 공간)에 추가하지 않고도 전역에서 볼 수 있도록 하는 유일한 방법은 중첩된 클래스다.

만약 그 내부 클래스를 사용하는 곳이 오직 메서드 하나라면, 그 내부 클래스를 유닛의 구현 *implementation* 구역에서 선언해도 같은 효과를 얻을 수 있다. 하지만, 그 내부 클래스가 유닛의 인터페이스 *interface* 구역에서 참조되어야 한다면(예: 필드 또는

파라미터에서 사용되는 경우), 그 내부 클래스는 같은 인터페이스 구역에 선언되어야 한다. 그러므로, 외부에서 볼 수 있다. 트릭 즉 제네릭 [generic\(일반\)](#) 타입을 기반 [base](#) 으로 하는 필드 하나를 선언하고 나서, 타입 변환 [casting](#) 을 통해 그것을 (사적으로 사용하는) 특정한 타입으로 바뀌서 쓰는 방식보다는 중첩된 클래스를 사용하는 것이 훨씬 더 깔끔하다.

참고 10장에는 중첩된 클래스를 활용하는 실용적인 예제가 있다. 그 예제는 `for-in` 루프를 지원할 수 있도록 사용자 정의 반복어 [iterator](#) 를 구현하는 부분에서 볼 수 있다.

08: 상속 Inheritance

클래스 `class`를 작성하는 주 이유가 캡슐화 `encapsulation`라면, 클래스 사이에서 상속 `inheritance`을 사용하는 주 이유는 유연성 `flexibility`이다. 이 두 개념을 결합하면, 변경된 버전을 만들 수 있어서 원래 버전은 바뀌지 않는 데이터 타입을 사용할 수 있다. 소위 "개방-폐쇄 원칙 `open-closed principle`"이라고 알려진 원칙이다.

"소프트웨어 엔티티 `entity`(독립 존재) (클래스, 모듈, 함수 등)는 확장 `extension`에는 개방적 `open` 이어야 한다. 하지만, 수정 `modification`에는 폐쇄적 `closed` 이어야 한다." - Bertrand Meyer, *Object-Oriented Software Construction*, 1988 년

상속은 매우 강력한 바인딩이라서 서로 단단히 결합되는 `tightly coupled` 코드를 유발한다는 점은 사실이다. 단단하게 결합되는 코드를 요즘 꺼려하지만, 상속이 굉장한 능력을 개발자에게 제공한다는 점 역시 사실이다. (물론 그만큼 더 많은 책임이 따른다).

상속에 대한 논쟁을 유발하고 싶지는 않다. 우리의 목표는 타입 상속이 어떻게 작동하는지, 특히 오브젝트 파스칼 언어에서 어떻게 작동하는지를 설명하는 것이다.

기존 타입으로부터 상속하기 Inheriting from Existing Types

(내가 예전에 만들었던 다른 사람이 만들어서 준 것이든) 이미 있는 클래스인데 살짝 다른 버전을 사용해야 하는 경우가 종종 있다.

예를 들면, 메서드를 새로 추가하거나 기존 메서드를 약간 변경해야 하는 경우다. 이럴 때는 원본의 코드를 바꾸면 쉽다. 그 클래스가 서로 다른 두 버전이 되고, 각각 다른 상황에 사용되기를 원하지 않는다면 말이다. 또한, 만약 다른 사람이 만든 클래스라면 (그리고 여러분이 라이브러리에서 찾은 것이라면) 여러분은 아마 여러분이 변경한 사항을 따로 잘 보관하고 싶어할 것이다.

전형적인 옛날 방식은 클래스 하나를 비슷한 버전 두 개로 만들 때, 원본 타입의 정의를 복사해 새 복사본을 만들고, 코드를 변경해 새 기능을 추가하고 그 타입에 새 이름을 지정한다. 이 방법은 효과가 있다. 하지만, 문제도 만든다. 코드를 복사하면 버그도 그대로 복사된다. 복사본 중 하나에서 버그를 수정할 때는 다른 복사본에서도 똑같이 수정해야 한다는 점을 잊지 말아야 한다. 또한 새 기능을 추가하고 싶을 때 다른 복사본에서도 그 작업을 해야 한다는 점도 마찬가지다. 복사본이 모두 몇 개든 그 개수만큼 반복해야 한다. 이런 코드를 처음 작성할 때는 불편하지 않을 수 있지만, 소프트웨어 관리에서는 재앙이다. 게다가 이 방식으로 만든 데이터 타입은 서로 전혀 다른 타입이다. 따라서 컴파일러는 이 두 타입 사이에 있는 공통점을 활용하지 못한다.

클래스 사이의 공통점을 표현할 때 생기는 이런 문제를 해결하기 위해, 오브젝트 파스칼에서는 기존 클래스에서 직접 새 클래스를 정의할 수 있도록 한다. 이 기법은 **상속 inheritance**(또는 **하위클래스 만들기 subclassing** 또는 **타입 파생 type derivation**)이라고 알려져 있다. 그리고 이는 객체 지향 프로그래밍 언어의 기본 요소 **fundamental element** 중 하나다.

기존 클래스에서 상속을 받으려면 하위클래스 선언의 시작 부분에 기존 클래스를 명시하면 된다. 실제로 새 폼 **form** 을 만들면, 그때마다 자동으로 아래와 같이 된다.

```
type
  TForm1 = class(TForm)
  ...
end;
```

위에 있는 클래스 정의는 TForm1 클래스가 TForm 클래스의 모든 메서드, 필드, 프로퍼티, 이벤트를 상속한다는 의미이다. 타입이 TForm1 인 오브젝트에는 TForm 클래스의 모든 공개 **public** 메서드를 적용할 수 있다. TForm 클래스 역시 자신이 상속받은 다른 클래스로부터 메서드들을 상속받는다. 이렇게 계속 거슬러 올라가면, 결국 TObject 클래스(모든 클래스의 기반 **base** 클래스)까지 간다.

이에 비해 C++, C#, Java 구문은 다음과 같다.

```
class Form1: TForm
{
  ...
}
```

상속 사례로, 앞 장의 ViewDate 예제를 바꿔보자. TDate 에서 새 클래스를 파생시키고, 상속받은 함수인 GetText 를 수정하자 (DerivedDates 예제의 Dates.pas 파일에서 발췌함).

```
type
  TNewDate = class(TDate)
  public
    function GetText: string;
end;
```

위에서, TNewDate 는 TDate 에서 파생된 클래스다. 일반적으로 TDate 는 TNewDate 의 **조상 ancestor** 클래스, **기반 base** 클래스, **부모 parent** 클래스라고 부른다. 그리고 TNewDate 는 TDate 의 **하위클래스 subclass**, **자손 descendant** 클래스, **자식 child** 클래스라고 부른다.

GetText 함수의 새 버전에서는 아래 코드처럼 구현에서 FormatDateTime 함수를 사용했다. FormatDateTime 은 미리 정의된 월 이름을 사용할 수 있다 (사용할 수 있는 기능은 그것 말고도 더 있음). 아래 코드에서 GetText 메서드를 보면, 'dddddd'가 있는데, 이것은 긴 날짜 형식을 사용하라는 옵션이다.

```
function TNewDate.GetText: string;
begin
    Result := FormatDateTime('dddddd', FDate);
end;
```

새 클래스를 정의했으니, 이제 이 새 데이터 타입을 사용할 수 있다. TNewDate 타입으로 ADay 오브젝트를 정의하고, FormCreate 메서드에서 TNewDate 클래스의 생성자를 호출하기만 하면 된다 (DerivedDates 예제의 폼 [form](#) 코드에서 발췌함).

```
type
    TDateForm = class(TForm)
    ...
    private
        FDay: TNewDate; // 변경된 선언 (Updated declaration)
    end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
    FDay := TNewDate.Create; // 변경된 코드 (Updated line)
    DateLabel.Text := FDay.GetText;
end;
```

더 이상 다른 변경을 하지 않아도, 새 애플리케이션은 잘 작동한다.

TNewDate 클래스는 날짜를 하루 올리기, 여러 일 올리기 등의 메서드를 상속받는다. 또한, 이 메서드들을 호출하던 예전 코드도 여전히 작동한다. GetText 메서드의 새 버전을 호출하기 위해, 소스 코드를 변경할 필요가 없다! 오브젝트 파스칼 컴파일러는 기존 호출 코드가 새 메서드에 바인딩되도록 자동으로 반영한다.

다른 이벤트 핸들러 [event handler](#) 들도 모두 똑같이 그대로이다. 그런데도, 그 의미는 상당히 변경되었다. 새로 출력된 결과는 아래와 같다 (그림 8.1 참조).

그림 8.1:

DerivedDates 프로그램의

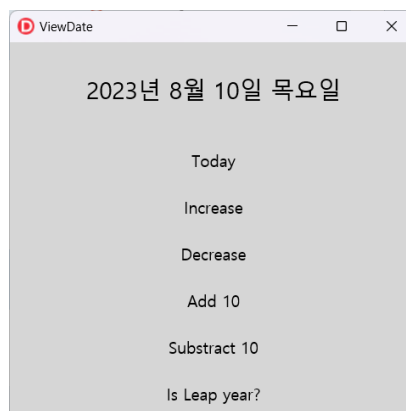
결과는 월을 표현할 때

숫자가 아니라 문자 이름이

보여지고 있다.

이 표현은 윈도우 지역 설정에

의해 결정된다



공통 기반 클래스 A Common Base Class

앞에서 기반 [base](#) 클래스를 지정하여 상속하는 경우를 보았다.

```
type
  TNewDate = class(TDate)
    ...
  end;
```

그런데, 만약 위 구문에서 기반 클래스를 적지 않으면 어떻게 될까?

```
type
  TNewDate = class
    ...
  end;
```

이 경우 새 클래스는 TObject 라는 기반 클래스를 상속받는다. 즉, 오브젝트 파스칼 클래스 계층 구조에서는 뿌리가 오직 하나다. 모든 클래스는 직접적이든 간접적이든 이 공통 조상 클래스로부터 상속받는다. TObject 에 많은 메서드들이 있고, 이 책에서도 사용되고 있지만 그 중 가장 일반적으로 사용되는 것은 Create, Free, Destroy 다. TObject 라는 기반 [fundamental](#) 클래스에 대한 전체 설명(언어 영역에도 해당되고, 런타임 라이브러리의 일부이기도 하다)과 사용할 수 있는 메서드 전체 목록은 17 장에 있다.

참고 공통 조상 클래스라는 개념은 C#과 Java 언어에도 존재한다. 그 언어들에서 공통 조상의 이름은 Object이다. 이와 반대로, C++ 언어에는 이러한 개념이 없다. 그리고 C++ 프로그램 안에는 대체로 독립적인 클래스 계층 구조 여러 개가 들어 간다.

보호된 필드와 캡슐화 Protected Fields and Encapsulation

위 TNewDate 클래스의 GetText 메서드 코드는 TDate 클래스와 동일한 유닛 안에서 작성된 경우에만 컴파일이 된다. 실제로, 이 메서드는 조상 클래스의 [private](#) [비공개](#) 필드인 FDate 에 접근한다. 만약 이 자손 클래스를 정의할 때, 부모 클래스와 같은 유닛이 아니라 다른 유닛 안에 작성하고 싶다면, 부모 클래스인 TDate 클래스에서 FDate 필드를 [protected](#) [보호됨](#)(또는 [strict protected](#) [엄격하게 보호됨](#))으로 선언하거나, 이 [private](#) 필드의 값을 읽는 간단한 메서드를 (가능하면 [protected](#) 로) 추가해야 한다.

일부 개발자들은 첫 번째 해법이 언제나 최선이라고 믿는다. 필드 대부분을 [protected](#) 로 선언하면 클래스를 확장하기가 더 좋아지고, 하위클래스 [subclass](#) 를 작성하기도 더 쉬워지기 때문이다. 하지만, 그건 캡슐화 [encapsulation](#) 라는 개념을 위반하는 것이다. 방대한 클래스 계층 구조 안에서, 기반 [base](#) 클래스의 [protected](#) 필드 몇개의 정의를 변경하는 것은 글로벌 [global/전역](#) 데이터 구조를 변경하는 것만큼이나 어렵다. 만약 파생된 클래스 10 개가 그 데이터를 접근하고 있다면, 그 데이터의 정의를 변경한 경우, 그 10 개 자손 클래스마다 해당 코드를 변경해야 할 수도 있다.

이처럼, 유연성 [flexibility](#), 확장성 [extension](#), 캡슐화 [encapsulation](#) 의 목적은 종종 서로 상충된다. 그럴 때면, 캡슐화를 우선해야 한다. 만약 유연성을 희생하지 않고 해낼 수 있다면 더욱 좋다. 가상 메서드 [virtual method](#) 를 사용하면 그렇게 해낼 수 있다. 가상 메서드에 대해서는 "나중에 바인딩하기와 다형성" 절에서 자세히 설명한다. 만약 하위클래스 코드를 더 빠르게 작성하게 하고 싶어서 캡슐화를 사용하지 않겠다는 선택을 한다면, 객체 지향 원칙을 따르지 않는 설계가 될 수 있다.

또한 `protected` 필드도 `private` 필드 접근 규칙과 똑같은 규칙을 공유한다. 따라서, 같은 유닛 안에 있는 클래스들은 언제든지 다른 클래스의 `protected` 멤버에 접근할 수 있다. 앞 장에서 언급했듯이, 더 강력한 캡슐화를 사용하려면 `strict protected` 접근 지정자를 사용할 수 있다.

"보호된 멤버 해킹" 사용하기 [Using the "Protected Hack"](#)

만약, 오브젝트 파스칼과 OOP 를 처음 접했다면, 이 절은 다소 수준이 높은 내용이므로, 이 책을 처음 읽을 때는 건너 뛰는 것도 좋다. 꽤 혼란을 느낄 수 있기 때문이다.

유닛 수준의 보호가 작동하는 방식을 볼 때, 현재 유닛 안에서 선언된 클래스의 기반 클래스에 있는 `protected` 멤버는 (`strict protected` 키워드가 사용되지 않았다면) 직접 접근될 수 있다. 이러한 논리적 바탕 위에서 소위 "보호된 멤버 해킹 [protected hack](#)" 이 가능하다. 즉, 기반 클래스와 똑같은 파생 클래스를 정의하면, 그 기반 클래스에 있는 `protected` 멤버에 접근할 수 있는 능력을 얻게 된다. 이제 어떻게 작동하는 것인지 살펴보자.

이미 우리는 클래스에 있는 `private` 과 `protected` 데이터는 그 클래스가 들어 있는 유닛 안에 있는 모든 함수와 메서드에서 접근할 수 있다는 점을 보았다. 예를 들어, 아래에 있는 간단한 클래스를 보자 (Protection 예제에서 발췌함)

```
Type
TTest = class
protected
  FProtectedData: Integer;
public
  PublicData: Integer;
  function GetValue: string;
end;
```

GetValue 메서드는 문자열 하나를 반환하는데, 거기에는 정수 값 두 개가 들어 있다.

```
function TTest.GetValue: string;
begin
  Result := Format('Public: %d, Protected: %d',
    [PublicData, FProtectedData]);
end;
```


만약 위 클래스가 자체 유닛 안에 들어 있다면, 다른 유닛 안에서는 그 클래스의 `protected` 부분에 직접 접근할 수 없다. 이런 맥락에서, 아래 코드를 다른 유닛 안에 작성했다고 생각해보자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  Obj.FProtectedData := 20; // 컴파일 되지 않는다.
  Show(Obj.GetValue);
  Obj.Free;
end;
```

컴파일러는 다음과 같은 에러 메시지를 만든다. *Undeclared identifier(선언되지 않은 식별자)*: “*FProtectedData.*” 이 메시지를 보고 어찌면 클래스의 `protected` 데이터는 다른 유닛에서 접근할 방법이 없다고 생각할 수 있다. 하지만, 돌아가는 방법이 있다.

다음과 같이 겉보기에는 쓸모없는 파생 클래스를 만들면 어떻게 될지 생각해 보자.

```
type
  TTestAccess = class(TTest);
```

이제, 이 클래스가 선언된 유닛 안에서는, `TTestAccess` 클래스의 어떤 `protected` 메서드이든 호출이 가능하다. 같은 유닛 안에 있는 클래스끼리는 서로 다른 클래스의 `protected` 메서드를 호출할 수 있다는 사실 때문이다.

그렇다면, 위 코드는 `TTest` 클래스의 오브젝트를 사용하는데 무슨 도움이 될까? 두 클래스의 메모리 레이아웃 [layout\(배치\)](#)이 똑같은 점을 고려하면, (메모리 레이아웃에 차이가 없기 때문에) 한 클래스의 오브젝트를 다른 클래스의 오브젝트로 취급하도록 컴파일러에게 강제할 수 있다. 일반적으로는 타입-불안전한 [type-unsafe](#) 캐스트 [cast](#) 다.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TTestAccess(Obj).FProtectedData := 20; // 컴파일 된다!
  Show(Obj.GetValue);
  Obj.Free;
end;
```

위 코드는 컴파일이 되고 작동도 잘된다. `Protection` 예제를 실행해보면 알 수 있다. 다시 말하지만, `TTestAccess` 클래스는 기반 [base](#) 클래스인 `TTest`에 있는 `protected` 필드들을 자동으로 상속받았다. 그런데, `TTestAccess`가 상속받은 필드의 데이터를 접근하는 위 코드는 `TTestAccess` 클래스가 있는 그 유닛 안에 있기 때문에 그 `protected` 데이터에도 접근할 수 있는 것이다.

이제 방법을 보여주었으니, 경고를 하겠다. 이런 식으로 클래스 보호 메커니즘을 위반하면 (접근해서는 안 되는 데이터에 접근함에 따라) 프로그램에서 에러가 발생할 수 있다. 또한 좋은 OOP 방법론에 위배된다. 하지만 거의 없기는 하지만 이 기법을 사용하는 것이 최선의 해결책이 되는 경우도 있다. 그런 경우는, 라이브러리 소스 코드 그리고 여러 컴포넌트의 코드를 보면 알 수 있을 것이다.

전반적으로, 이 기법은 *해킹*이므로 가능하면 피해야 한다. 물론 이 효과 역시 이 언어 사양 *specification* 의 한 부분이라고 보기도 한다. 또한, 오브젝트 파스칼에서 현재 버전뿐만 아니라 예전 버전에도 있었고, 모든 플랫폼에서 사용할 수 있기도 하다.

상속부터 다형성까지 From Inheritance to Polymorphism

상속은 코드 중복을 피하고 서로 다른 클래스 간에 코드 메서드를 공유할 수 있다는 점에서 멋진 기술이다. 그러나 상속의 진정한 힘은 서로 다른 클래스들의 오브젝트를 단일화된 방식을 사용해서 다룰 수 있는 능력에서 나온다. 이런 방식은 종종 객체 지향 프로그래밍 언어에서 *다형성* *polymorphism* 이라는 용어를 사용하거나 또는 *나중에 바인딩 하기* *late binding*이라고 부른다.

이 기능을 완전히 이해하려면, 몇 가지 요소를 살펴봐야 한다. 파생 클래스 간의 타입 호환성 *type compatibility*, 가상 메서드 *virtual method* 등이 해당되는데, 지금부터 살펴보자.

상속과 타입 호환성 Inheritance and Type Compatibility

앞에서 어느 정도 살펴본 것처럼, 오브젝트 파스칼은 엄격하게 타입이 지정되는 언어 *strictly typed language* 다. 즉, 예를 들어, 불리언 *bool* 변수에는 정수 값을 할당할 수 없다. 적어도 명시적인 형 변환이 없이는 그렇다. 두 값 *value* 이 타입-호환되려면 *type-compatible*, 데이터 타입이 같아야 한다는 것이 기본 규칙이다. (보다 정확히 말하자면) 그 값들의 데이터 타입은, 이름 *name* 이 같아야 하고 정의 *definition* 가 작성된 유닛도 같아야 한다.

이 규칙에 중요한 예외가 있다. 바로 클래스 타입인 경우다. 클래스 하나를 선언하고 (예: TAnimal), 그 클래스에서 새 클래스를 파생시킨 (예: TDog) 경우, TDog 타입인 오브젝트를 TAnimal 타입인 변수에 할당할 수 있다. 개는 동물이기 때문이다! 그러니, 아마 놀랄 수 있지만, 아래에 있는 생성자 호출 두 개는 모두 합법적 *legal* 이다.

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
  MyAnimal1 := TAnimal.Create;
  MyAnimal2 := TDog.Create;
```

보다 정확하게 말하자면, 조상 클래스의 오브젝트를 기대하는 곳에는 자손 클래스의 오브젝트를 언제나 사용할 수 있다. 하지만 그 반대는 합법적이지 않다. 즉 자손

클래스의 오브젝트를 기대하는 곳에는 조상 클래스의 오브젝트를 사용할 수 없다. 간단히 설명하기 위해 코드로 적어보면 다음과 같다.

```
MyAnimal := MyDog; // 문제없다.
MyDog := MyAnimal; // 오류다!!!
```

사실, 개는 동물이라고 언제나 말할 수 있다. 하지만, 어떤 동물이 주어졌을 때, 그것이 개라고 단정할 수는 없다. 어떨 때는 사실이겠지만, 언제나 그런 것은 아니기 때문이다. 상당히 논리적이다. 이 언어의 타입 호환성 규칙 역시 같은 논리를 따른다.

이 중요한 언어 기능의 의미를 살펴보기 전에, Animals1 예제를 사용해 보기 바란다. 이 예제에는 TAnimal과 TDog 클래스가 있는데, 부모와 자식 클래스로 정의되어 있다.

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    FKind: string;
  end;

  TDog = class(TAnimal)
  public
    constructor Create;
  end;
```

위에서 Create 메서드 두 개는 그저 FKind 값을 지정하는 일만 한다. 그렇게 지정된 FKind 값은 GetKind 함수가 반환한다.

그림 8.2:

Animals1 예제의 폼(form)을
개발 환경에서 본 모습

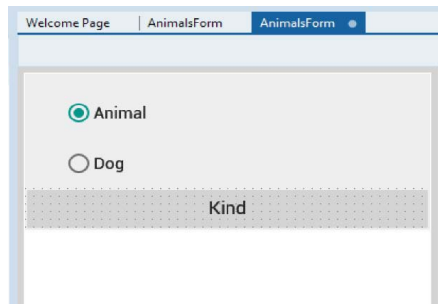


그림 8.2에 있는, 이 예제의 폼(form)에는 라디오 버튼(radioButton) 두 개가 있어서 (판넬(panel) 안에 들어 있음), 두 클래스 중 어느 하나의 오브젝트를 선택하도록 한다. 오브젝트가 선택되면 TAnimal 타입으로 된 private 필드인 FMyAnimal에 저장된다. 이 클래스의 인스턴스(instance)가 생성되고 created 초기화되는 initialized 시점은 폼이 생성되는 시점이다. 또한, 라디오 버튼 중 하나가 클릭될 때마다 다시 생성된다 (두 라디오 버튼의 코드 중 아래쪽 버튼의 코드만 아래 코드 조각에 적어 놓았다).


```

procedure TFormAnimals.FormCreate(Sender: TObject);
begin
    FMyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RadioButton2Change(Sender: TObject);
begin
    FMyAnimal.Free;
    FMyAnimal := TDog.Create;
end;

```

마지막으로, 그 아래에 있는 Kind 버튼은 GetKind 메서드를 호출하여, 현재 동물을 알아내고 그 결과를 (이 폼의 아래 부분을 차지하고 있는) 메모 [memo](#) 안에 표시한다.

```

procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
    Show(FMyAnimal.GetKind);
end;

```

나중에 바인딩하기와 다형성 Late Binding and Polymorphism

오브젝트 파스칼에서 함수와 프로시저는 일반적으로 *정적 바인딩* static binding 기반이다. *일찍 바인딩하기* early binding 라고도 한다. 그 의미는 메서드 호출이 컴파일러나 링커에 의해 해소된다 resolved 는 뜻이다. 즉 그 함수 호출 요청은 특정 메모리 주소로 교체된다. 거기에는 해당 함수나 프로시저가 컴파일 되어 있다(함수의 주소라고도 한다). 객체 지향 프로그래밍 언어에는 또다른 형태의 바인딩도 있다. *동적 바인딩* dynamic binding, 즉 *나중에 바인딩하기* late binding 라는 것이다. 이 경우, 호출되는 함수의 실제 주소가 실행 중에 정해진다. 그 호출을 하기 위해 사용된 인스턴스의 타입에 따라 달라지기 때문이다.

이 기법의 장점을 *다형성* polymorphism 이라고 부른다. 다형성이란, 여러분이 메서드 호출을 작성할 때, 그 메서드를 변수에 적용하게 되는데, 그 변수가 참조하는 오브젝트의 타입에 따라 (텔파이가 호출하는) 실제 메서드가 달라지는 것을 말한다. 변수가 참조하는 오브젝트의 실제 클래스가 무엇인지는 런타임 runtime(실행 중) 이 되기 전까지 텔파이는 알 수 없다. 클래스에 적용되는 타입 호환성 규칙 때문인데, 앞에서 설명했다.

참고 오브젝트 파스칼 메서드는 기본 default 바인딩 방식이 일찍 바인딩하기 early binding 이다. C++과 C#도 그렇다. 그 이유 중 하나는 이 방식이 더 효율적이기 때문이다. 이와 달리, Java는 기본 바인딩 방식이 나중에 바인딩하기 late binding 이다 (그리고, 일찍 바인딩하기 early binding 를 사용하도록 표시하여 컴파일러로 하여금 메서드를 최적화하도록 지시하는 방법이 제공된다).

이름이 같은 메서드를 어느 클래스와 그 클래스의 서브클래스(예: TAnimal 과 TDog) 둘 다 정의하고 있고, 그 메서드가 나중에 바인딩 late binding 되는 상황을 보자. 여러분이 그 메서드를 일반 generic 변수인 FMyAnimal 에 적용했다면, FMyAnimal 변수는 런타임에 TAnimal 클래스의 오브젝트와 TDog 클래스의 오브젝트 중 무엇이든 참조할 수 있다. 실제로 호출되는 메서드는 현재 오브젝트의 클래스에 따라 런타임에 정해진다.

이 기법을 보여주기 위해 Animals1 프로젝트를 확장해 준다 Animals2 예제를 만들었다. 새 버전에서는, TAnimal 과 TDog 클래스 모두 새 메서드인 Voice 를 추가했다. 선택한 동물이 내는 소리를 텍스트와 소리로 출력하는 메서드다. TAnimal 클래스에는 virtual 로 정의했고, TDog 클래스의 정의에서는 이 메서드를 오버라이드 [override \(덮어쓰기\)](#) 한다. 즉, virtual 과 override 키워드가 사용되었다.

```
type
  TAnimal = class
  public
    function Voice: string; virtual;
  TDog = class(TAnimal)
  public
    function Voice: string; override;
```

물론, 두 메서드 모두 구현이 정의되어야 한다. 아래 코드는 간단한 구현을 정의한다.

```
function TAnimal.Voice: string;
begin
  Result := 'AnimalVoice';
end;

function TDog.Voice: string;
begin
  Result := 'ArfArf';
end;
```

이제 FMyAnimal.Voice 를 호출하면 어떤 효과를 얻게 될까? 상황에 따라 다르다. FMyAnimal 변수가 현재 TAnimal 클래스의 오브젝트를 참조한다면 TAnimal.Voice 메서드를 호출한다. 그런데, TDog 클래스의 오브젝트를 참조하고 있다면 TDog.Voice 메서드를 호출한다. 그 이유는 이 함수가 virtual [가상](#)이기 때문이다.

FMyAnimal.Voice 에 대한 호출은 TAnimal 클래스의 모든 자손 클래스의 인스턴스인 오브젝트에서 작동한다. 심지어 이 메서드 호출 이후 즉 호출 범위 밖에서 정의된 클래스일지라도 작동한다. 컴파일러는 이 호출이 모든 자손 클래스에서 호환될 수 있도록 하지만, 그렇다고 해서 그 자손 클래스를 일일이 알 필요가 없다. 그저 조상 클래스만 가지고 작업할 수 있다. 즉, TAnimal 의 하위클래스가 되는 모든 클래스들은 FMyAnimal.Voice 호출과 호환된다.

재사용성 [reusability](#) 을 객체 지향 프로그래밍 언어에서 선호하는 중요한 기술적 이유가 바로 이것이다. 우리는 계층 구조 안에 있는 클래스를 사용하여 코드를 작성할 때, 그 계층 구조 안에서 있는 클래스들 각각을 구체적으로 알지 못해도 된다. 다시 말해서, 계층 구조(와 그 프로그램)를 여전히 확장할 수 있다 [extensible](#). 심지어 그 계층 구조를 사용하는 코드가 수천 줄에 달하더라도 말이다. 물론 거기에는 한 가지 조건이 있다. 바로 계층 구조 상의 조상 클래스들을 매우 신중하게 설계해야 한다.

Animals2 예제는 이 새로운 클래스를 사용하는 것을 보여준다. 폼 [form](#) 화면은 이전 예제와 비슷하다. 버튼을 클릭하면 그 결과를 텍스트로 표시하고 소리도 낸다.


```
begin
  Show(FMyAnimal.Voice);
  MediaPlayer1.FileName := SoundsFolder + FMyAnimal.Voice + '.wav';
  MediaPlayer1.Play;
end;
```

참고 이 애플리케이션은 MediaPlayer 컴포넌트를 사용하여 사운드 파일 두 개 중 하나를 재생한다 (사운드 파일은 애플리케이션과 함께 제공되며, 파일의 이름은 실제 사운드 즉 Voice 메서드가 반환하는 값을 사용했다). 일반 동물의 소리로는 좀 무작위(random) 소음을, 개의 소리로는 짖는 소리를 파일에 담아 놓았다. 윈도우에서 실행하고 음성 파일이 적절한 폴더에 있는 한, 이 코드는 쉽게 작동한다. 하지만, 모바일 플랫폼에 배포하려면 노력 조금 필요하다. 제공되는 실제 데모를 잘 살펴보면, 배포와 폴더 구조가 어떻게 구성되어 있는지를 알 수 있을 것이다.

메서드를 오버라이딩, 다시 정의하기, 다시 도입하기 Overriding, Redefining, Reintroducing

앞에서 봤듯이, 자손 클래스에서 override 키워드를 사용하면, 나중에 바인딩되는 메서드를 오버라이드(override)한다. 메서드 오버라이드는 조상 클래스에서 virtual 이라고 정의된 메서드인 경우에만 가능하다는 점을 알아 두자. 그런데, dynamic 이라고 정의되어 있는 메서드인 경우에도 가능하다. 이 키워드는 좀 더 뒤에 가서 다룬다. 이런 키워드가 없는 메서드는 정적 메서드로 간주되므로, 나중에 바인딩하기를 통해 변경될 수 없다. 변경하고 싶으면, 조상 클래스의 코드를 변경하는 수밖에 없다.

참고 override 키워드는 앞 장의 예제에서 이미 사용했었다. 기본(default) Destroy 소멸자를 오버라이딩 할 때 사용했다. 기본 Destroy는 기반 클래스인 TObject로부터 상속된다.

규칙은 간단하다. 정적(static)으로 정의된 메서드는 모든 하위클래스에 상속되어서도 정적으로 유지된다 (단, 하위클래스에서 같은 메서드 이름으로 새 가상(virtual) 메서드를 만들어 조상의 정적 메서드를 숨기는(hide) 경우에는 그렇지 않다). Virtual로 정의된 메서드는 모든 하위클래스에 상속되어서도 나중에 바인딩(late-bound) 되도록 유지된다. 이 규칙을 바꿀 방법은 없다. 나중에 바인딩되는 메서드는, 컴파일러가 작동하는 방식이 다르고 생성하는 코드도 다르기 때문이다.

정적 메서드를 다시 정의하려면, 하위클래스에 이름이 같은 메서드를 추가하면 된다. 파라미터는 원래 메서드와 같아도 되고 달라도 된다. 더 지정해야 하는 것은 없다. 이와 달리, virtual 메서드를 오버라이드 하려면, 반드시 파라미터가 같아야 하고 override 키워드도 사용해야 한다.

```
type
  TMyClass = class
    procedure One; virtual;
    procedure Two; // 정적 메서드 (Static method)
  end;
  TMySubClass = class(TMyClass)
    procedure One; override;
    procedure Two;
  end;
```


위 메서드 `Two` 는 다시 정의 `redefined` 된 것이고, 나중에 바인딩되는 메서드가 아니다. 즉, 이 메서드를 기반 클래스 타입인 변수에 적용하면, 기반 클래스의 `Two` 메서드가 항상 호출된다 (심지어, 그 변수가 참고하고 있는 것이 파생 `derived` 클래스의 오브젝트이고, 그 파생 클래스에 `Two` 메서드의 다른 버전이 있는 경우에도 그렇다).

메서드를 오버라이드 `override` 하는 전형적인 방법은 크게 두 가지다. 하나는 조상 클래스의 메서드와 다르도록 완전히 새 버전으로 바꾸는 것이다. 다른 하나는 기존 메서드에 코드를 조금 더 추가하는 것이다. 두 번째 방법을 쓰려면 `inherited` 키워드를 사용하면 된다. 이 키워드는 조상 클래스에 있는 그 메서드를 호출한다. 예를 들어 다음과 같이 작성할 수 있다.

```
procedure TMySubClass.One;
begin
    // 새 코드
    ...
    // 상속된 프로시저인 TMyClass.One을 호출한다
    inherited One;
end;
```

`override` 키워드를 사용해야 하는 이유가 궁금할 수 있다. 다른 언어에서는, 가상 메서드를 하위클래스에서 다시 정의하면, 원래 메서드를 자동으로 오버라이드 한다. 이와 달리, 명확히 키워드를 쓰면, 조상 클래스의 메서드 이름과 하위 클래스의 메서드 이름이 대응하는지를 컴파일러가 확인하게끔 한다 (다시 정의하는 함수의 철자를 틀리는 것은 다른 OOP 언어에서 흔히 생기는 실수다). 또한 조상 클래스에 그 메서드가 `virtual` 로 되어 있는지 등 필요한 확인 작업들도 같이 수행하게끔 한다.

참고 널리 사용되는 OOP 언어 중에 `override` 키워드를 사용하는 언어가 또 있는데 바로 C#이다. 언어를 설계한 사람이 같다는 사실을 안다면 놀랍지 않다. Anders Hejlsberg는 왜 `override` 키워드가 버전을 다루는 근본적인 도구로써 라이브러리 설계에 활용되는 지를 설명했다. 내용은 <http://www.artima.com/intv/nonvirtual.html> 에 있다. 보다 최근에는 애플이 Swift 언어에서 `override` 키워드를 도입해 파생 클래스에서 메서드를 변경할 때 사용한다.

이 키워드의 장점은 더 있다. 여러분이 라이브러리의 클래스를 상속받아 클래스를 만들고 그 안에 정적 메서드를 정의하는 경우, 앞으로 문제가 없다고 확신할 수 있다. 심지어 우리가 정의한 메서드와 이름이 똑 같은 메서드가 그 라이브러리 안에 새로 생겨도 문제가 없다. 우리가 정의한 메서드는 `override` 키워드가 붙어있지 않기 때문에, 별개의 메서드로 간주된다. 즉, 라이브러리에 있는 메서드의 새 버전으로 취급되지 않는다 (만약 그렇게 취급된다면 우리의 기존 코드가 깨질 수도 있을 것이다).

메서드에는 오버로드 `overload` 가 지원되므로, 이 그림이 보다 복잡해진다. 하위클래스는 메서드의 새 버전을 추가하기 위해 `overload` 키워드를 사용할 수 있다. 새 메서드의 파라미터가 기반 클래스에 있는 버전의 파라미터와 다른 경우, 오버로드 된 메서드가 된다. 하지만, 그렇지 않다면, 기반 클래스의 메서드를 교체한다. 예문을 보자.


```

type
  TMyClass = class
    procedure One;
  end;

  TMySubClass = class(TMyClass)
    procedure One(S: string); overload;
  end;

```

위에서, 기반 클래스에는 메서드에 `overload` 를 표시할 필요가 없다. 그런데, 만약 기반 클래스의 메서드가 `virtual` 메서드라면, 컴파일러는 경고를 표시한다. 경고 메시지는 *Method 'One' hides virtual method of base type 'TMyClass.'* 다.

그런 경우, 컴파일러에게 숨기기를 바란다는 내 의도를 더 정확하게 알려주어서 경고 메시지가 표시되지 않게 하려면, `reintroduce` 재도입 지시어를 사용하면 된다.

```

type
  TMyClass = class
    procedure One; virtual;
  end;

  TMySubClass = class(TMyClass)
    procedure One(S: string); reintroduce; overload;
  end;

```

이 코드는 `ReintroduceTest` 예제에 있으니 찾아서 더 실험해 보기 바란다.

참고 `reintroduce` 키워드가 사용되는 상황으로는, 컴포넌트 클래스에 사용자 정의 [custom Create](#) 생성자를 추가하고 싶을 때를 들 수 있다. 컴포넌트는 이미 가상 `Create` 생성자를 기반 클래스인 `TComponent`로부터 상속받고 있기 때문이다.

상속과 생성자 Inheritance and Constructors

지금까지 살펴본 것처럼 `inherited` 키워드를 사용하면, 파생 클래스의 메서드에서 기반 클래스에 있는 이름이 같은 (또는 이름이 다른) 메서드를 호출할 수 있다. 생성자에서도 역시 이 사실은 다르지 않다. C++, C#, Java 와 같은 다른 언어에서는 기반 클래스의 생성자 호출이 암묵적이고 (파라미터를 기반 클래스의 생성자에게 전달해야 할 때에는) 강제적이다. 이와 달리, 오브젝트 파스칼에서는, 기반 클래스의 생성자 호출은 필수 사항이 아니다.

하지만, 대부분의 경우, 기반 클래스의 생성자를 수작업으로 호출하는 것은 너무나 중요하다. 아래의 경우가 그렇다. 모든 컴포넌트 클래스들은 그 컴포넌트의 초기화 [initialization](#) 가 실제로 조상인 `TComponent` 클래스 수준에서 수행되기 때문이다:

```

constructor TMyComponent.Create(Owner: TComponent);
begin
  inherited Create(Owner);
  // 특정 코드...
end;

```


컴포넌트의 경우, Create 는 가상 메서드이므로 특히 더 중요하다. 이와 비슷하게 모든 클래스의 경우, Destroy 소멸자는 가상 메서드다. 따라서, 여러분은 Destroy 안에서 반드시 inherited 를 호출해야 한다는 점을 명심해야 한다.

한 가지 의문이 남는다. 우리가 클래스 하나를 생성하는데, 그 클래스가 TObject 를 직접 상속받는 것이어도 그 생성자 안에서 TObject.Create 즉 기반 클래스의 생성자를 호출해야 할까? 기술적인 관점에서 보면, 정답은 "아니오"이다. 기반 생성자는 실제로 속이 비어 있기 때문이다. 하지만, 어떤 경우이든지 항상 기반 생성자를 호출한다면 그것은 좋은 습관이라고 생각한다. 하지만, 이 습관때문에 코드가 불필요하게 느려진다는 점을 인정할 수 밖에...전혀 눈치챌 수 없는 마이크로 초 수준이지만.

이런 농담은 제쳐 두고, 두 가지 접근 방식 모두 합당한 이유가 있다. 하지만, 특히 이 언어에 초보자라면 항상 기반 클래스의 생성자를 호출하는 것이 좋은 프로그래밍 습관이고, 더 안전하게 코드를 작성하는 습관이기도 하므로 추천한다.

가상 메서드와 동적 메서드 Virtual versus Dynamic Methods

오브젝트 파스칼에서는, 나중에 바인딩하기를 활성화하는 방법은 두 가지다. 앞서 살펴본 것처럼 메서드를 virtual 또는 dynamic 이라고 선언할 수 있다. 이 두 키워드의 구문 [syntax](#) 은 완전히 똑같다. 그 결과 역시 똑같다. 다른 점은 내부 메커니즘이다. 즉 나중에 바인딩하기를 컴파일러에서 구현하는 방식만 다를 뿐이다.

가상 [virtual](#) 메서드는 가상 메서드 테이블 (VMT^{Virtual Method Table}, 흔히 *vtable* 이라고 함)을 기반으로 한다. 가상 메서드 테이블은 메서드 주소들이 담긴 배열이다. 컴파일러는 가상 메서드가 호출되면, 그 오브젝트의 가상 메서드 테이블 안에 있는 n번째 위치에 저장되어 있는 메모리 주소로 바로 가는 코드를 생성한다.

가상 메서드 테이블을 사용하면 메서드 호출 실행이 빨라질 수 있다. 주요 단점은 각 자손 클래스 [descendant class](#)마다 각 가상 메서드를 위한 항목이 필요하다는 것이다. 심지어 그 하위클래스에서 오버라이드 하지 않은 메서드일지라도 항목으로 가지고 있어야 한다. 가끔은 가상 메서드 테이블 항목들이 클래스 계층 구조 전체에 전파되는 결과를 낳기도 한다 (다시 정의되지 않는 메서드들까지도). 이 방식은 많은 메모리가 필요하다. 똑같은 메서드 주소를 여러 번 저장하기 때문이다.

이와 반대로, 동적 [dynamic](#) 메서드 호출은 그 메서드를 가리키는 고유 번호를 사용하여 찾아낸다. 대응되는 함수를 찾는 작업 속도는 테이블을 조회하여 한 번에 가상 메서드를 찾는 방식보다 대체로 더 느리다. 장점은 오직 자손이 그 메서드를 오버라이드 한 경우에만 동적 [dynamic](#) 메서드 호출 항목이 전파된다는 점이다. 오브젝트 계층이 크고 깊은 경우에는, 가상 메서드 대신 동적 메서드 호출을 사용하는 편이 메모리를 크게 절약할 수 있다. 단지 속도 면에서 손해가 아주 조금 있다.

프로그래머의 관점에서 보면, 이 두 접근 방식은 그저 내부 구현이 다를 뿐이다. 그로 인해, 속도와 메모리 사용이 조금 다르다. 그 점을 빼면, 가상 방식과 동적 방식은 똑같다.

이제 이 두 모델의 차이점을 설명했으니, 중요한 점을 강조하고 싶다. 대부분의 경우에서, 애플리케이션 개발자는 `dynamic` 이 아니라 `virtual` 을 사용한다.

윈도우에서 메시지 다루기 Message Handlers on Windows

윈도우용 애플리케이션을 구축한다면, 나중에 바인딩되는 특수 목적 메서드를 써서 윈도우 시스템 메시지를 다룰 수 있다. 이런 목적을 위해, 오브젝트 파스칼에서는 메시지 처리 메서드를 정의할 수 있도록 `message` 지시어를 제공한다. `message` 지시어를 사용하는 프로시저는 반드시 알맞은 타입으로 된 `var` 파라미터 하나를 가져야 하고, `message` 지시어 바로 뒤에는 그 메서드에서 처리하려는 윈도우 메시지의 번호를 붙여야 한다. 예를 들어, 사용자-정의 메시지를 처리하는 코드는 아래와 같다. 지시어 뒤에 있는 `WM_USER` 윈도우 상수는 그 윈도우 메시지의 번호를 나타내는 숫자 값이다.

```
type
  TForm1 = class(TForm)
    ...
  procedure WmUser(var Msg: TMessage); message WM_USER;
end;
```

프로시저의 이름 그리고 파라미터의 실제 타입은 개발자가 정하기 나름이다. 단, 그 물리적인 데이터 구조는 해당 윈도우 메시지의 구조와 맞아야 한다. 윈도우 API 와 접속 interface 하는 데 사용되는 유닛들이 있는데 그 안에는 다양한 윈도우 메시지에 맞게 미리 정의된 여러 가지 레코드 타입들이 들어있다. 이 기법은 윈도우 메시지와 API 함수들 모두에 대해 잘 알고 있는 베테랑 윈도우 프로그래머에게 크게 유용할 수 있다. 그러나, 당연히, 이 기술은 다른 운영체제(예: 맥 OS, iOS, 안드로이드)와는 전혀 호환되지 않는다.

메서드 추상화와 클래스 추상화 Abstracting Methods and Classes

클래스의 계층 구조를 만들 때, 어느 클래스를 기본 클래스로 할지를 결정하기 어려울 때가 있다. 기반 클래스 경우에는 실제 엔티티 entity(독립 존재)를 표현하지는 않고 그저 공유하려는 행위 shared behavior 몇 개 정도만 가지고 있을 수도 있기 때문이다. 예를 들어, 고양이 클래스나 개 클래스와 같은 것들의 기반 클래스로 동물 클래스가 있을 수 있다. 동물 클래스처럼 오브젝트를 생성하지 않는 클래스라면 추상 abstract 클래스가 되도록 표시하는 경우가 많다. 구현 implementation 구체적이지도 완전하지도 않기 때문이다. 추상 클래스는 추상 메서드를 가질 수 있다. 추상 메서드에는 실제 구현이 없다.

추상 메서드 Abstract Methods

`abstract` 키워드는 가상 메서드 중에서 현재 클래스가 아니라 그 하위클래스에서만 정의할 수 있는 메서드를 선언하는 데 사용된다. `abstract` 지시어가 붙은 메서드는 그것 만으로 이미 정의가 완료된 메서드다. 즉 포워드 선언 [forward declaration](#) 이 아니다. 추상 메서드에게 정의 [definition](#) 를 제공하려고 시도하면 컴파일러가 불평할 것이다.

오브젝트 파스칼에서는, 추상 메서드가 있는 클래스의 인스턴스를 생성하는 것이 가능하다. 하지만, 그렇게 하면 컴파일러는 경고 메시지를 표시한다. *Constructing instance of <클래스 이름> containing abstract methods*. 그리고, 실행 중에 추상 메서드를 호출하면, 델파이는 특정 런타임 예외 [runtime exception](#) 를 발생시킨다.

참고 C++, Java 및 기타 언어는 더 엄격한 접근 방식을 사용한다. 이러한 언어에서는 추상 클래스의 인스턴스를 만드는 것을 전혀 허용하지 않는다.

`abstract` 메서드를 사용하는 이유가 궁금할 수 있다. 그 이유는 다형성 [polymorphism](#) 을 사용하기 때문이다. `TAnimal` 클래스에 있는 `Voice` 가 virtual `abstract` 메서드라면, 모든 하위클래스들은 `Voice` 메서드를 다시 정의할 수 있다.

그렇게 하면, 우리는 `Voice` 메서드를 호출할 때 일반 [generic](#) `FMyAnimal` 오브젝트를 사용하여 하위클래스에 정의된 각 동물을 참조할 수 있다는 이점이 있다. 그런데, 이 메서드가 `TAnimal` 클래스의 인터페이스 [interface](#) 구역에 없다면, 컴파일러는, 정적 타입 검사를 수행하기 때문에, 이런 호출을 허용하지 않게 되고, 일반 [generic](#) `FMyAnimal` 오브젝트를 사용하는 경우, 바로 그 클래스 즉, `TAnimal` 에 정의된 메서드만 호출된다.

부모 클래스는 자신의 하위클래스가 제공하는 메서드를 호출하지 못한다. 부모 클래스 안에 최소한 그 메서드에 대한 선언이 있어야 가능하다. 그러므로 `abstract` 메서드로 선언해 놓으면 된다. 새 `Animals3` 예시는 `abstract` 메서드 사용하기와 추상 호출의 에러를 보여준다. 이 예시에 있는 클래스들의 인터페이스 구역은 다음과 같다.

```
type
  TAnimal = class
  private
    FKind: string;
  public
    constructor Create;
    function GetKind: string;
    function Voice: string; virtual; abstract;
  end;

  TDog = class(TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;

  TCat = class(TAnimal)
```



```

public
  constructor Create;
  function Voice: string; override;
  function Eat: string; virtual;
end;

```

위에서 가장 흥미로운 부분은 TAnimal 클래스의 정의 안에 있는 virtual abstract 메서드인 Voice 이다. 또한 각 파생 클래스에서 이 메서드를 오버라이드 [override](#) 하고 있다는 점 그리고 새 가상 메서드인 Eat 가 추가된 점을 눈여겨보자. 이 두 방식은 차이가 무엇일까? Voice 메서드를 호출할 때, 우리는 MyAnimal.Voice 를 사용할 수 있었다.

```
Show(MyAnimal.Voice);
```

그런데, Eat 메서드를 호출하려고 하면 어떻게 하면 될까? 우리는 TAnimal 클래스의 오브젝트에 Eat 메서드를 적용하지 못한다.

```
Show(MyAnimal.Eat);
```

위 구문은 에러가 발생한다. 메시지는 *Field identifier expected* 이다.

이 문제를 해결하려면, 동적이면서 안전한 타입 캐스팅을 사용해 TAnimal 오브젝트를 TCat 또는 TDog 오브젝트로 처리하면 된다. 하지만, 그 방식은 매우 번거롭고 에러가 발생하기도 쉽다.

```

begin
  if MyAnimal is TDog then
    Show(TDog(MyAnimal).Eat)
  else if MyAnimal is TCat then
    Show(TCat(MyAnimal).Eat);

```

이 코드는 나중에 "안전한 타입 캐스트 연산자" 절에서 설명한다. 위 문제를 해결하는 전형적인 해결책은 TAnimal 클래스 정의에 가상 메서드로 Eat 을 추가하는 것이다. 즉, 모든 동물은 먹기 때문이고, abstract 키워드가 있기 때문에 이 해결책을 훨씬 더 선호한다. 위와 같은 보기 흉한 코드를 피하는 것이 바로 다형성을 쓰는 이유다. 또한 위 경우에, 우리가 구현하는 모델은 진짜 세상을 잘 표현한다.

마지막으로, 추상 메서드를 가진 클래스는, 종종 추상 클래스로 간주된다는 점을 알아두자. 그런데, 클래스에 직접 abstract 지시어를 붙여서 추상 클래스임을 명확히 표시할 수도 있다 (그러면 추상 메서드가 없더라도 추상 클래스로 간주된다). 다시 말하지만, 오브젝트 파스칼에서는, 추상 클래스도 인스턴스를 생성할 수 있도록 허용한다. 따라서, 클래스 자체를 추상 클래스라고 선언하는 건 그다지 쓸모가 없다.

봉인된 클래스와 최종 메서드 Sealed Classes and Final Methods

이미 말했듯이, Java 는 동적 `dynamic` 접근 방식을 취한다. 그리고, 나중에 바인딩하기 (또는 가상 함수)가 기본 `default` 이다. 그러므로, Java 에는 상속이 더 이상 안되는 (봉인된 `sealed`) 클래스라는 개념, 그리고 파생 클래스가 오버라이드 `override` 하지 못하는 메서드(최종 메서드 `final method` 또는 비-가상 `non-virtual` 메서드)라는 개념이 도입되었다.

봉인된 `sealed` 클래스는 더 이상 상속이 안되는 클래스다. 이 클래스가 적합한 경우를 예로 들면, (소스 코드는 없이) 컴포넌트 또는 런타임 패키지를 배포하고, 다른 개발자들이 코드를 수정할 수 없도록 제한하고 싶을 때다. 이 두 개념의 원래 목표 중 하나는 런타임 `runtime(실행 중)` 보안 강화이다. 하지만, 오브젝트 파스칼처럼 완전히 컴파일되는 언어에서는 이런 보안 조치가 일반적으로 필요 없다.

최종 메서드 `final method` 는 가상 메서드 중에서 상속받은 클래스에서 더 이상 오버라이드 하지 못하는 것이다. 다시 말하지만, Java 에서는 의미가 있다 (모든 메서드가 가상인 것이 기본 `by default` 이고, 최종 메서드를 사용하면 최적화 효과가 크기 때문이다). 그런데, 이 메서드는 C#에서도 채택하고 있는데, 다만 가상 메서드를 만들려면 명시적으로 표시해야, 그 중요성 역시 훨씬 더 적다. 마찬가지로 오브젝트 파스칼에서도 채택하고 있다. 하지만 거의 사용되지 않는다.

구문 `syntax` 으로 보면, 봉인된 클래스 `sealed class` 의 코드는 다음과 같다.

```
type
  TDeriv1 = class sealed(TBase)
    procedure A; override;
  end;
```

위 클래스를 상속하려고 시도하면, 에러가 발생한다. 메시지는 *“Cannot extend sealed class TDeriv1”*이다. 최종 메서드 `final method` 의 코드 구문은 다음과 같다.

```
type
  TDeriv2 = class(TBase)
    procedure A; override; final;
  end;
```

위 클래스에서 상속한 다음 그 클래스에서 A 메서드를 오버라이드 하려고 시도하면 컴파일러 에러가 발생한다. 메시지는 *“Cannot override a final method”*다.

안전한 타입 캐스트 연산자 Safe TypeCast Operators

앞에서 봤듯이, 언어 타입 호환성 규칙에 따르면, 조상 `ancestor` 클래스를 기대하는 곳에 자손 `descendant` 클래스를 사용할 수 있다. 그 반대의 경우는 불가능하다.

Eat 메서드가 TDog 클래스에는 있는데, TAnimal 클래스에는 없다고 가정하자. 만약 FMyAnimal 변수가 개를 참조하고 있다면, 우리는 그 변수에서 Eat 함수를 호출하고

싶을 것이다. 그런데, 만약 `FMyAnimal` 변수가 참조하고 있는 것이 개가 아니라 다른 클래스라면, `Eat` 함수를 호출할 때 에러가 발생한다. 명시적 타입 캐스트 [cast\(변환\)](#)를 하면, 고약한 실행 중 에러 (또는 더 나쁘게는, 메모리를 덮어쓰기라는 미묘한 문제)를 일으킬 수 있다. 왜냐하면 컴파일러는 그 오브젝트의 타입이 맞는지 그리고 호출하려는 메서드가 실제로 그 오브젝트에 존재하는지 확인할 수 없기 때문이다.

이 문제를 해결하기 위해, 런타임 타입 정보 [runtime type information](#)에 기반한 기술을 사용할 수 있다. 근본적으로 [essentially](#), 실행 중에는 각 오브젝트가 자신의 타입과 자신의 부모 클래스를 "알고" 있기 때문에, 우리는 이 정보를 요청할 수 있다. `is` 연산자 또는 `TObject`에 있는 메서드들을 사용하면 된다. `is` 연산자의 파라미터는 오브젝트와 클래스 타입이다. 그리고 반환 값은 불리언 [Boolean](#)이다.

```
if FMyAnimal is TDog then
    ...
```

위에 있는 `is` 표현식이 참 [true](#)을 반환하려면, `FMyAnimal` 오브젝트가 현재 `TDog` 클래스 또는 `TDog`의 자손 클래스의 오브젝트 (즉, `TDog`과 타입 호환이 되는 오브젝트)를 참조하고 있어야 한다. 즉, `TAnimal` 변수에 저장된 `TDog` 오브젝트가 정말로 있는지를 테스트하는 위 구문은 정말 `TDog` 오브젝트가 있을 때 성공한다. 따라서, 이 표현식이 `True`를 반환한다면, 우리는 이 오브젝트(`FMyAnimal`)를 그 데이터 타입(`TDog`)으로 된 변수에 안전하게 할당할 수 있다.

참고 `is` 연산자의 실제 구현은 `TObject` 클래스의 `InheritsFrom` 메서드에서 제공한다. 따라서 `FMyAnimal.InheritsFrom(TDog)`이라고 작성해도 같은 표현식 [expression](#)이다. `InheritsFrom` 메서드를 직접 사용하는 이유는 `is` 연산자를 지원하지 않는 클래스 참조와 특수 목적 타입에서도 적용할 수 있기 때문이다.

이제 우리는 그 동물이 개라는 것을 확실히 알기 때문에, 직접 타입을 캐스트 [cast](#)하는 (일반적으로는 안전하지 않은) 코드를 사용해도 된다. 그 방법은 아래와 같다.

```
if FMyAnimal is TDog then
begin
    MyDog := TDog(FMyAnimal);
    Text := MyDog.Eat;
end;
```

또다른 타입 변환 연산자인 `as`를 직접 사용하면, 위와 똑같은 동작을 하는 코드를 작성할 수 있다. `as` 연산자는 요청된 클래스가 그 오브젝트의 실제 클래스와 호환이 되는 경우에만 변환 [convert](#)한다. 만약 호환되지 않으면, 런타임 예외를 내고 실패한다. `as` 연산자의 파라미터는 오브젝트와 클래스 타입이다. 그리고 그 결과는 새 클래스 타입으로 "변환된" 오브젝트다. 코드는 아래와 같다.

```
MyDog := FMyAnimal as TDog;
Text := MyDog.Eat;
```


만약, Eat 함수 호출만 한다면, 더 짧게 아래와 같이 작성할 수 있다.

```
(FMyAnimal as TDog).Eat;
```

위 표현식의 결과는 TDog 클래스 데이터 타입의 오브젝트다. 따라서 그 클래스가 가진 모든 메서드를 이 오브젝트에 적용할 수 있게 된다. 기존 타입 캐스트 방식과 as 를 사용한 타입 캐스트의 차이는, as 를 사용한 방식은 그 오브젝트의 실제 타입을 미리 확인해서, 혹시라도 확인한 현재 타입과 요청된 새 타입이 호환되지 않으면 EInvalidCast 라는 예외를 발생시킨다는 점이다 (예외 [exception](#) 는 다음 장에서 설명한다).

경고 이와 반대로, C# 언어의 as 표현식이 오브젝트가 타입 호환이 되지 않는 경우 nil을 반환한다. 그리고, 직접 타입 캐스트 방식이 예외를 발생시킨다. 따라서, 기본적으로, 이 두 연산은 C#과 오브젝트 파스칼에서 서로 정반대이다.

이 예외를 피하려면 is 연산자를 써서 테스트가 성공한 경우에만, 일반 타입 캐스트를 수행한다. (사실 is 와 as 를 나란히 쓸 이유는 없다. 타입 검사를 두 번 수행할 필요가 없기 때문이다. 그렇지만 종종 우리는 is 와 as 가 함께 사용된 코드를 보게 된다).

```
if FMyAnimal is TDog then  
    TDog(MyAnimal).Eat;
```

이 두 타입캐스트 연산자 모두 오브젝트 파스칼에서 매우 유용하다. 우리는 종종 일반화된 [generic](#) 코드를 작성해두고, 그 코드를 (타입 같거나 심지어 타입이 달라도) 여러 가지 컴포넌트를 대상으로 사용할 수 있기를 원한다. 예를 들어, 이벤트 [event](#) 에 응답하는 메서드는 컴포넌트를 파라미터로 전달하게 되는데, 각종 컴포넌트를 일반 [generic](#) 타입(TObject)으로 전달한다. 그래서, 종종 우리는 전달된 컴포넌트를 다시 원래 타입으로 되돌려 놓고 쓰기도 한다. 코드는 아래와 같다.

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    if Sender is TButton then  
        ...  
end;
```

이것은 흔히 쓰이는 기법이다. 이 책에서도 몇몇 예제에서 나온다. (이벤트 [event](#) 에 대한 소개는 10 장에서 한다).

이 두 타입 캐스트 연산자 즉, is 와 as 는 엄청나게 강력하다. 따라서 프로그램을 만드는 표준 구조로 받아들이고 싶을 수 있다. 비록 정말로 강력하긴 하지만, 특수한 경우에만 사용하는 것이 좋다. 여러 클래스가 관련되는 복잡한 문제를 해결해야 할 때는 다형성 [polymorphism](#) 사용을 먼저 시도해야 한다. 그리고 다형성만 가지고 적용하기 힘든 특수한 경우에만 타입 변환 연산자를 사용하여 보완해야 한다.

참고 타입캐스트 연산자를 사용하면, 성능에 약간의 부정적인 영향을 미친다. 왜냐하면 클래스 계층 구조를 타고 가 봐야만 올바른 타입캐스트인지 아닌지를 확인할 수 있기 때문이다. 이미 봤듯이, 가상 메서드 호출은 메모리에서 바로 찾을 수 있다. 따라서 훨씬 더 빠르다.

비주얼 폼 상속 Visual Form Inheritance

상속 inheritance 은 라이브러리 클래스 그리고 사용자가 작성한 클래스에만 사용되는 것이 아니라, 오브젝트 파스칼을 기반으로 하는 개발 환경 전체에 상당히 널리 퍼져 있다. 지금까지 본 것처럼, 우리가 IDE 에서 폼 하나를 생성하면, 그 폼은 TForm 으로부터 상속받은 클래스의 인스턴스이다. 따라서 시각적 애플리케이션이라면 모두 상속을 기반으로 하는 구조를 가진다. 비록 우리가 그저 이벤트 핸들러 안에 들어가는 코드만 작성한다고 해도 그렇다.

하지만, 숙련된 개발자에게도 잘 알려지지 않은 것이 있다. 바로 이미 생성한 폼에서 새로운 폼을 상속할 수 있다는 점이다. 이는 일반적으로 *시각적 폼 상속*이라고 불리는 기능이다 (그리고 오브젝트 파스칼 개발 환경에서 매우 두드러지게 나타나는 특징이다).

시각적 폼 상속은 우리가 상속의 힘을 시각적으로 직접 확인하고 그 규칙을 직접 파악할 수 있기 때문에 더 흥미롭다! 이것이 실제로도 유용할까? 글쎄, 주로 그것은 구축하려는 애플리케이션의 종류에 따라 다르다. 만약 폼이 여러 개이고, 모습이 매우 비슷한 폼들이 있거나, 또는 공통 요소들을 담고 있다면, 우리는 공통된 컴포넌트들과 공통된 이벤트 핸들러들을 기반 base 폼 안에 넣는다. 그리고 하위클래스에서는 각자의 특정 동작과 특정 컴포넌트를 추가하는 방식을 쓸 수 있다. 흔한 사례로, 시각적 폼 상속을 사용하여 애플리케이션의 폼 몇 개를 커스터마이징하는 경우가 있다. 그렇게 해서 각 고객사마다 다른 폼을 제공하면서도, 어떤 소스 코드도 중복으로 유지할 필요가 없다 (상속을 사용하는 핵심 이유 중 으뜸이 이것이다).

또한, 시각적 폼 form 을 상속해서 다양한 운영체제 operating system 와 폼 팩터 form factor 에 맞게 (예: 휴대폰에서 태블릿으로) 애플리케이션을 커스터마이징 할 수 있다. 그러면, 소스 코드나 폼을 정의하는 코드 어느 것도 중복하지 않는 것이 가능하다. 그저 표준 폼으로부터 원하는 클라이언트에 맞는 특정 버전을 상속받기만 하면 된다.

시각적 요소 상속의 가장 큰 장점을 잊지 말자. 원본 폼을 나중에 변경하면, 파생된 폼들도 모두 자동으로 업데이트된다. 상속의 이런 장점은 객체-지향 프로그래밍 언어에서 매우 잘 알려진 것이다. 게다가, 동반되는 유용한 효과로 다형성 polymorphism 을 활용할 수 있다. 즉, 우리는 기반 폼에 가상 메서드를 추가해 놓고, 하위클래스 폼에서 오버라이드 할 수 있기 때문에, 두 폼 모두를 참조하지만, 각자의 메서드를 호출하도록 할 수 있다.

참고 동일한 요소로 구성된 폼 form 을 구축하는 또 다른 접근 방식은 프레임 frame 을 사용하는 것이다. 프레임은 폼 판넬 form panel 들이 조합되어 시각적으로 구성되는 방식이다. 두 방식 모두 디자인을 할 때 그 폼의 두 버전을 대상으로 작업할 수 있다. 그런데, 시각적 폼 상속 방식에서 우리는 서로 다른 두 개의 클래스 (부모와 자식)를 정의하게 된다. 이와 달리, 프레임 사용 방식에서 우리는 프레임 클래스 하나와 폼 안에 담긴 그 프레임의 인스턴스를 가지고 작업하게 된다.

기반 클래스로부터 상속하기 Inheriting From a Base Form

시각적 폼 상속을 지배하는 규칙들은 상당히 단순하다. 상속이 무엇인지 그 개념을 명확히 알고 있다면 말이다. 기본적으로, 하위클래스 폼 안에는 부모 폼에 있는 컴포넌트들이 똑같이 들어 있다. 또한 거기에 새 컴포넌트들을 더 추가할 수도 있다. 하지만, 기반 클래스의 컴포넌트를 제거할 수는 없다. 그래도 (만약 시각적 컨트롤이라면) 보이지 않도록 할 수 있다. 중요한 점이 있다. 상속받은 컴포넌트들의 프로퍼티들을 우리가 쉽게 바꿀 수 있다.

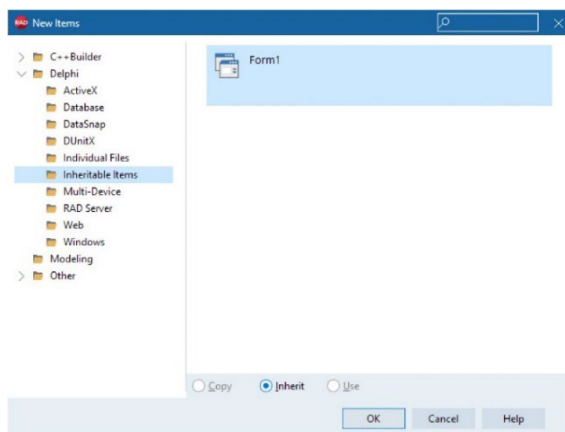
컴포넌트의 프로퍼티를 상속된 폼에서 변경하는 경우, 그 프로퍼티는 부모 폼에서 어떻게 변경하더라도, 그 하위 폼에 변경 사항이 전달되지 않는다는 점을 잘 알아야 한다. 하지만, 그 외의 프로퍼티들은 여전히 부모의 변경 사항이 하위클래스 버전에 그대로 반영된다. 컴포넌트의 프로퍼티를 자식에서 변경했는데, 다시 일치시키고 싶을 수도 있다. 그럴 때는 오브젝트 인스펙터에서 해당 프로퍼티를 선택하고 "Revert to Inherited" 메뉴 명령을 사용하면 된다. 또한 두 프로퍼티의 값을 똑같이 지정하고 코드를 다시 컴파일 해도 된다. 하나의 컴포넌트 안에서 여러 프로퍼티를 수정했는데, 그 컴포넌트의 프로퍼티 전체를 다시 기본 버전에 일치시키고 싶을 때는, 폼 디자이너에서 그 컴포넌트를 선택하고, 마우스 오른쪽을 클릭하면 나타나는 메뉴 중에서 "Revert to Inherited"를 사용하면 된다.

상속되는 것은 컴포넌트만이 아니다. 새 폼은 기반 폼의 (이벤트 핸들러까지 포함하여) 모든 메서드들을 상속받는다. 그리고 우리는 새 이벤트 핸들러를 추가하거나 기존 핸들러를 오버라이드 [override](#) 할 수 있다.

VisualInheritTest 예제를 보면 시각적 폼 상속이 어떻게 작동하는지를 볼 수 있다. 이 예제를 직접 구축할 수 있는데, 그 방법은 다음과 같다. 먼저 개발 환경에서 multi-device project 하나를 새로 만들고, blank project 를 선택하고, 그 프로젝트의 메인 폼에 버튼 두 개를 추가한다. 그런 다음 File > New > Others 를 선택하고, New Items 대화 상자 안에서 "Inheritable Items" 페이지를 선택한다 (아래 그림 8.3). 그리고 그 페이지에서 아래와 같이 상속하고 싶은 폼을 선택하면 된다.

그림 8.3:

New Items 대화 상자에서
상속된 폼을 생성할 수 있다



새 폼에도 그 버튼 두 개가 똑같이 들어 있다. 새 폼을 텍스트로 열어 보면 (View as Text 메뉴), 아래와 같다 (앞부분만 발췌함).

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
end
```

그리고, 이 새 클래스의 선언 코드는 아래와 같다. 여기에서 기반 클래스를 잘 보자. 일반적인 TForm 이 아니라 실제 클래스가 기반 클래스다.

```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
  public
    { Public declarations }
end;
```

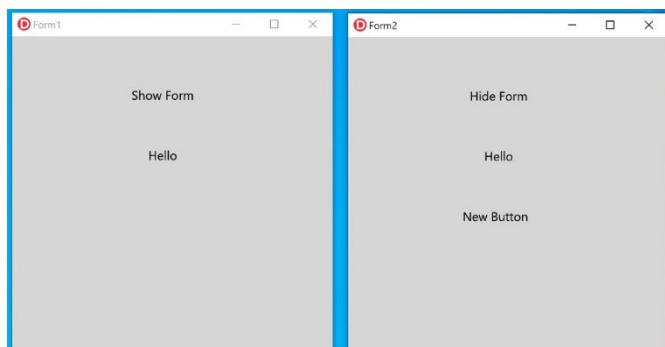
폼을 텍스트로 봤을 때, 그 안에 `inherited` 키워드가 있다는 점을 눈여겨보자. 또한, 이 폼 텍스트에서는 기반 클래스 폼에서 정의된 컴포넌트들도 몇 개가 들어 있다. 우리가 만약 상속받은 버튼 중 하나의 캡션 `caption` 을 변경하고, 새로 버튼 하나를 더 추가하면 그 디자인 작업은 이 텍스트에 그대로 반영된다.

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
  inherited Button1: TButton
    Text = 'Hide Form'
  end
  object Button3: TButton
    ...
    Text = 'New Button'
    OnClick = Button3Click
  end
end
```

위 텍스트에는 오직 값이 다른 프로퍼티만 나열된다. 그 외 다른 것들은 그대로 상속되기 때문이다.

그림 8.4:

VirtualInheritTest
예제를 실행하면 볼 수 있는
폼 두 개를 나란히 본 모습



첫 번째 폼의 버튼 두 개에는 각각 OnClick 핸들러가 있다. 그 안의 코드는 간단하다. 첫 번째 버튼은 두 번째 폼을 보여주기 위해 두 번째 폼의 Show 메서드를 호출한다. 두 번째 버튼은 간단한 메시지를 표시한다.

상속받은 폼은 어떤 동작을 할까? 먼저 Show 버튼의 동작을 변경해서 Hide 버튼이 되도록 구현했다. 즉, 기반 클래스의 이벤트 핸들러를 실행하지 않도록 했다 (그러기 위해, 기본으로 들어간 `default inherited` 호출을 주석으로 처리해서 빼내었다). 이와 달리, Hello 버튼은 기반 클래스의 메시지 표시를 수행하도록 그대로 두고, 이어서 두 번째 메시지를 수행하도록 추가하기 때문에 `inherited` 호출을 그대로 남겨두었다.

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    // Inherited;
    Hide;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    inherited;
    ShowMessage('Hello from Form2');
end;
```

명심해야 할 점이 있다. 상속된 메서드인 경우, 우리가 `inherited` 키워드를 사용하면 기반 클래스의 메서드들 중에서 이름이 같은 것을 호출한다. 그런데, 이벤트 핸들러인 경우에는, `inherited` 키워드가 있으면 부모 폼 `form` 의 그 이벤트에 연결된 핸들러를 호출한다 (이벤트 핸들러 메서드의 이름과는 무관하다).

물론, 기반 폼에 있는 메서드들도 역시 현재 폼의 메서드라고 생각하고 자유롭게 호출할 수 있다. 이 예제는 시각적 폼 상속의 여러 기능들을 살펴보는데 도움이 된다. 하지만, 그 진정한 힘을 확인하려면 이 책에서 미처 다루지 못한 더 복잡한 실제 예제를 살펴봐야 할 것이다.

09: 예외 다루기 Handling Exceptions

클래스에 있는 다른 기능들을 다루기 전에, 오브젝트 파스칼 언어에서 에러 조건을 다루는 오브젝트 집단인 *예외* *exception* 를 집중해서 살펴볼 필요가 있다.

예외 처리 *exception handling* 는 프로그램을 더 견고하게 만들기 위한 아이디어다. 소프트웨어 또는 하드웨어에서 발생하는 에러(및 기타 모든 유형의 에러)를 단순하고 일원화된 방식으로 다루는 능력을 추가한다. 프로그램은 이러한 에러에도 살아남거나 우아하게 종료한다. 즉, 사용자는 종료 전에 데이터를 저장할 수 있다. 예외를 사용하면, 여러분은 에러를 다루는 코드와 일반 코드를 분리할 수 있어서 코드가 서로 얽히지 않는다. 덕분에 작성하는 코드는 더 간결하다. 또한 실제 프로그래밍 목적과 무관한 유지 관리 작업으로 인해 코드가 지저분해지는 상황도 더 적다.

또 다른 장점도 있다. 예외는 일원화 *uniform* 되고 보편적인 *universal* 에러-보고 메커니즘을 정의한다. 델파이 컴포넌트 라이브러리도 이 메커니즘을 사용한다. 개발 시스템은 실행 중에 문제가 생기면 예외를 발생시킨다. 여러분이 작성한 적절한 코드가 있으면, 시스템은 그 문제를 인식하고 해결하려고 한다. 적절한 코드가 없으면, 그 코드를 호출한 코드에게 차례대로 예외가 넘어간다. 끝까지 가도, 여러분의 코드가 그 예외를 다루지 않으면, 개발 시스템이 일반적으로 처리한다. 즉 표준 에러 메시지를 표시하고 프로그램을 계속 진행하려고 시도한다. 비정상적인 상황에서 여러분의 코드가 어떠한 예외 처리 블록에도 잡히지 못한다면, 예외 발생은 프로그램 종료 *terminate* 로 이어진다.

오브젝트 파스칼에서 예외 처리 메커니즘 전체는 아래 5 개의 키워드가 바탕이 된다.

- **try**: 보호되는 코드 블록의 시작 부분이라고 등록 *register* 한다.
- **except**: 보호되는 코드 블록의 끝임을 등록하고, 예외 처리 코드의 시작을 알린다.
- **on**: 개별 예외 처리 문장 *statement* 이라고 표시한다. 특정 예외 클래스와 연결한다. (각 예외 클래스 별로 이런 구문 *syntax* 을 사용한다: **on** 예외-타입 **do** 문장)
- **finally**: (예외가 발생해도) 반드시 항상 실행되어야 하는 블록을 지정한다.

- **raise**: 예외를 발생시키는 문장이다. 발생시키려는 예외 오브젝트를 파라미터로 전달한다 (다른 언어들은, **throw** 키워드를 이 동작에 사용한다)

아래 표는 오브젝트 파스칼의 예외 처리 키워드를 (C#, Java 와 같은) C++ 예외 구문과 비교한 것이다.

try	try
except on	catch
finally	finally
raise	throw

C++ 언어 용어를 써서 말하자면, 우리는 예외 오브젝트를 던진다 **throw**. 그리고, 타입 별로 해당 예외를 잡는다 **catch**. 이 표현을 오브젝트 파스칼 용어로 말하면, 우리는 예외 오브젝트를 **raise** 문 **statement** 에게 전달한다. 그리고, 전달되는 예외를 파라미터로 받는 곳은 **except on** 문이다.

Try-Except 블록 Try-Except Blocks

간단한 try-except 예제부터 시작해보자. 아래 코드에는 일반 **general** 예외 처리 블록이 하나 있다 (ExceptionsTest 예제에서 발췌함).

```
function DividePlusOne(A, B: Integer): Integer;
begin
  try
    // B가 0이면 예외를 발생시킨다 (Raises exception if B equals 0)
    Result := A div B;
    Inc(Result);
  except
    Result := 0;
  end;
  // 코드가 더 있음
end;
```

참고 프로그램을 델파이 디버거에서 실행하는 경우에는, 예외 핸들러 **exception handler**가 있다고 해도, 예외가 발생하면 프로그램은 디버거 **debugger**에 의해 중지되는 것이 기본이다 **by default**. 물론 보통 우리가 원하는 바이다. 우리는 그 예외가 발생한 곳을 알고 싶고, 예외 핸들러 호출을 단계별로 넘기면서 보기를 원하기 때문이다. 만약, 예외가 알맞게 처리되는 경우라면 (사용자가 보는 것과 똑같이) 프로그램이 계속 실행되기를 바란다면, “Run without debugging” 명령을 사용하거나, 디버거 옵션에서 전체 (또는 일부 유형의) 예외를 비활성화하면 된다.

위 코드는 예외를 "침묵"시킨다. 그리고 그 결과에 영 0 을 지정하는 것으로 대신한다. 실제로 사용되는 애플리케이션에서는 사실 위 방식이 그다지 타당하지 않다 (이렇게 사용자로부터 에러를 덮는 것은 주로 나쁜 방식이다). 위 코드는 에러 처리의 핵심 메커니즘을 이해할 수 있도록 돕기 위해 간단한 상황을 만든 것이다.

위 함수를 호출하는 이벤트 핸들러 `event handler`의 코드는 아래와 같다.

```
var
  N: Integer;
begin
  N := DividePlusOne(10, Random(3));
  Show(N.ToString);
```

위 코드에서 프로그램은 임의의 값을 생성하여 사용한다. 따라서 사용자가 버튼을 클릭하면 유효한 상황 (세 번 중에 두 번) 또는 유효하지 않는 상황에 처하게 된다. 따라서 서로 다른 두 가지 프로그램 흐름이 있을 수 있다.

- B가 0이 아닌 경우: 프로그램은 나눗셈을 수행하고, 1을 더한 다음, 예외 블록을 건너 뛰고 예외 블록의 `end` 뒤에 이어지는 문장(`// 코드가 더 있음`)으로 간다.
- B가 0인 경우: 나눗셈에서 예외가 발생한다. 예외가 발생한 곳보다 뒤에 있는 문장은 모두 (위 코드에는 해당 문장이 하나뿐임) 그냥 넘어간다. 대신, 처음 만나는 `try-except` 블록이 실행된다. 프로그램은 예외를 발생시킨 원래 위치로 돌아가지 않고, `except` 블록 뒤에 있는 문장 (`// 코드가 더 있음` 부분)을 실행한다.

이 예외 모델을 설명할 때 우리는 재개하지 않음 `non-resumption` 방식을 따른다고 말한다. 에러가 발생한 경우, 에러 조건을 처리한 다음 다시 에러를 일으킨 문장으로 돌아가는 것은 매우 위험하다. 그 시점에는 프로그램의 상태가 정의되지 않았을 가능성이 높기 때문이다. 예외는 실행 흐름을 크게 바꾼다. 그 뒤에 나오는 문장들을 건너 뛰고, 메모리 스택을 거슬러 올라가면서 적절한 에러 처리 코드를 찾는다.

위에 있는 `except` 블록은 매우 간단하다. 옆에 `on` 문이 없다. 만약, 여러 가지 타입의 예외(즉 여러 가지 예외 클래스 타입)를 다루어야 하거나, 이 블록에게 전달된 예외 오브젝트를 직접 접근하고 싶으면, `on` 문을 하나 또는 그 이상 사용하면 된다.

```
function DividePlusOneBis(A, B: Integer): Integer;
begin
  try
    Result := A div B; // Error if B equals 0
    Result := Result + 1;
  except
    on E: EDivByZero do
      begin
        Result := 0;
        ShowMessage(E.Message);
      end;
    end;
end;
```

위 예외 처리 문장은 `EDivByZero` 예외를 잡아 낸다. `EDivByZero`는 런타임 라이브러리에 정의된 예외다. 이런 런타임 문제(0으로 나누기, 잘못된 동적 캐스팅 등), 시스템 문제(메모리 소진 등), 컴포넌트 에러 (유효하지 않은 인덱스 등)를 참조하는 예외 타입들이 많다. 그 예외 클래스들 모두 최상위 기반 클래스인 `Exception`의 자손이다. `Exception` 클래스에는 최소한의 기능 몇 가지가 있다. 위 코드의 `Message` 프로퍼티도 그 중 하나다. 이 클래스들은 실제로 논리적으로 구성된 계층으로 되어 있다.

참고 오브젝트 파스칼에서, 타입은 일반적으로 문자 T로 시작된다. 이와 달리, 예외 클래스는 일반적으로 문자 E로 시작한다. 즉 이 규칙에서 예외다.

예외의 계층 구조 [The Exceptions Hierarchy](#)

다음은 런타임 라이브러리의 `System.SysUtils` 유닛에 정의된 핵심 예외 클래스들 중 일부다 (아래 핵심 목록에 더해, 대부분의 다른 시스템 라이브러리들은 자체 예외 타입을 가지고 있다).

```
Exception
  EArgumentException
    EArgumentOutOfRangeException
    EArgumentNilException
  EPathTooLongException
  ENotSupportedException
  EDirectoryNotFoundException
  EFileNotFoundException
  EPathNotFoundException
  EListError
  EInvalidOpException
  ENoConstructException
  EAbort
  EHeapException
    EOutOfMemory
    EInvalidPointer
  EInOutError
  EExternal
    EExternalException
  EIntError
    EDivByZero
    ERangeError
    EIntOverflow
  EMathError
    EInvalidOp
    EZeroDivide
    EOverflow
    EUnderflow
  EAccessViolation
  EPrivilege
  EControlC
  EQuit
  EInvalidCast
  EConvertError
  ECodesetConversion
  EVariantError
  EPropReadOnly
  EPropWriteOnly
  EAssertionFailed
  EAbstractError
  EIntfCastError
  EInvalidContainer
  EInvalidInsert
  EPackageError
  ECFError
  EOSError
```



```

ESafecallException
EMonitor
  EMonitorLockException
  ENoMonitorSupportException
EProgrammerNotFound
ENotImplemented
EObjectDisposed
EJNIException

```

참고 내가 볼 때 가장 이상한 예외 클래스는 EProgrammerNotFound다. 이 웃긴 예외가 정확히 무슨 상황에 사용되는지 여전히 개인적으로 알아내고 싶다! 델파이의 라이브러리들 안에는 숨겨져 있는 이스터 에그 [Easter egg](#) 들이 몇 개 있다. 이것도 그 중 하나다.

핵심 예외 계층 구조를 봤으니, except-on 문을 알아보자. 이 문장들은 발생한 예외 오브젝트에 맞는 예외 클래스가 나올 때까지 차례대로 평가된다. 타입 일치 규칙은 타입 호환성 규칙이 적용된다(앞 장에서 설명): 예외 오브젝트의 타입은 그 특정 타입의 모든 기반 [base](#) 타입들과 호환된다 (예: TDog 오브젝트는 TAnimal 클래스와 호환된다).

즉, 여러분은 해당 예외에 맞는 여러 가지 예외 핸들러 [exception handler](#) 를 사용할 수 있다. 보다 세부적인 [granular](#) 예외(예외 계층 구조 상 아래쪽 클래스)를 다루면서, 일반 [generic](#) 예외도 함께 다루려면, 핸들러 블록을 나열할 때 더 구체적인 것에서 더 일반적인 것 순서로 적어야 한다. 즉, 하위 예외에서 그 예외의 부모 클래스 순서로 나열해야 한다.

또한, Exception 타입을 처리하는 핸들러는 모든 예외를 받아 낸다. 따라서 이 타입은 순서 상 가장 뒤에 있어야 한다.

아래 코드에는 핸들러 두 개가 블록 하나에 있다.

```

function DividePlusOne(A, B: Integer): Integer;
begin
  try
    Result := A div B; // B가 0이면 에러 발생
    Result := Result + 1;
  except
    on EDivByZero do
      begin
        Result := 0;
        MessageDlg('Divide by zero error', mtError, [mbOK], 0);
      end;
    on E: Exception do
      begin
        Result := 0;
        MessageDlg(E.Message, mtError, [mbOK], 0);
      end;
  end; // 예외 블록의 끝
end;

```

위 코드에는 예외 핸들러 두 개가 같은 try 블록 하나 안에 있다. 이렇게 핸들러의 개수는 얼마든지 많아도 된다. 앞에서 설명했듯이, 순서대로 하나씩 평가된다.

명심해야 할 점이 있다. 가능한 모든 예외를 처리하는 핸들러를 사용하는 것은 일반적으로 좋은 선택이 아니다. 알 수 없는 `unknown` 예외는 시스템에게 맡기는 것이 더 좋다. 시스템의 기본 `default` 예외 핸들러는 일반적으로 해당 예외 클래스의 예러 메시지를 메시지 상자에 표시하고 나서 프로그램을 다시 정상 가동한다.

팁 이런 일반 예외 핸들러를 실제로 변경할 수 있다. `Application.OnException` 이벤트를 처리하는 메서드를 제공하면 된다. 예를 들면, 예외 메시지를 파일에 기록하기 `log`만 하고, 사용자에게는 표시하지 않을 수 있다.

예외 발생시키기 `Raising Exceptions`

오브젝트 파스칼 프로그래밍에서 우리가 만나는 예외들은 대부분 시스템에서 생성된 것들이다. 하지만, 우리가 코드 안에서 직접 예외를 발생시킬 수도 있다. 유효하지 않거나 일관되지 않은 데이터를 실행 중에 탐지하는 경우에 활용한다.

대부분의 경우, 사용자 정의 예외 `custom exception` 를 위해서는, 여러분이 직접 예외 타입을 정의한다. 그 방법은 기본 `default` 예외 클래스 또는 그 예외 클래스의 하위클래스들 (위 목록에서 있는 것들) 중 하나로부터 상속받아 새 하위 클래스를 만들기만 하면 된다.

type

```
EArrayFull = class(Exception);
```

대부분의 경우, 우리는 새 예외 클래스에 메서드나 필드를 추가할 필요가 없다. 빈 파생 클래스 `empty derived class` 를 선언하는 것만으로 충분하다.

위 예외 타입은 메서드에서 배열에 요소를 추가하려고 시도하고 하는데 그 배열이 이미 꽉 찬 상황에 사용될 수 있다. 그런 상황이 되면, 위에서 우리가 정의한 예외 오브젝트를 생성하고 `raise` 키워드 뒤에 넣어 전달하면 된다. 아래 코드와 같다.

```
if MyArray.IsFull then
```

```
    raise EArrayFull.Create('배열이 가득 찼습니다');
```

위 `Create` 메서드에는 (기반 `base` 클래스 `Exception` 으로부터 상속받은 것임) 파라미터 하나가 있는데, 여기에는 사용자에게 그 예외를 설명할 때 사용할 문자열을 넣는다.

참고 예외 오브젝트를 우리가 직접 생성하는 경우라 해도, 그것을 파괴할 걱정을 하지 않아도 된다. 예외 오브젝트는 자동으로 제거되기 때문이다. 예외-핸들러 메커니즘 자체가 그렇게 되어 있다.

`raise` 키워드를 사용하는 두 번째 경우가 있다. `except` 블록 안에서 우리가 원하는 처리를 직접 하지만 거기에서 끝내지 않고, 예외를 다음 예러 핸들러에게 넘겨주고 싶을 수 있다. 그럴 때는, `except` 블록 안에서 원하는 조치를 한 다음 그저 `raise` 를 호출하기만 하면 된다. 파라미터를 붙일 필요가 없다. 이런 작업을 *예외 다시 발생시키기* `reraise` 라고 부른다.

예외와 스택 Exceptions and the Stack

프로그램에서 예외가 발생했는데 현재 루틴이 그 예외를 처리하지 않으면, 그 메서드와 호출 스택 [call stack](#) 에 무슨 일이 생길까? 프로그램은 스택 상에 있는 함수들 중에서 핸들러를 찾기 시작한다. 즉, 프로그램은 현재 함수를 빠져나간다. 그 함수의 나머지 부분을 건너뛰고 해당 예외 핸들러로 넘어간다는 의미이다. 그 방식을 이해하려면, 디버거를 사용하거나 또는 간단한 출력 라인을 넣어서 특정 소스 코드 문장이 언제 실행되는지 출력하도록 한다. 다음 예제인 `ExceptionHandler` 는 두 번째 방식을 사용한다.

예를 들어, `ExceptionHandler` 예제의 폼에서 `Raise1` 버튼을 누르면, 예외가 발생한다. 그런데, 처리되지 않는다. 그 결과, 아래 코드는 마지막 부분이 절대 실행되지 않는다.

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
    // 보호 장치가 없는 호출
    AddToArray(24);
    Show( '프로그램은 절대로 여기에 도달하지 못한다' );
end;
```

위 메서드는 `AddToArray` 프로시저를 호출한다. `AddToArray` 프로시저는 항상 예외를 일으킨다. 예외가 처리되면, 프로그램 흐름이 해당 핸들러 뒤부터 다시 시작된다. 그 예외를 발생시킨 코드 뒤가 아니다. 아래에 있는 수정된 코드를 고려할 필요가 있다.

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
    try
        // 이 프로시저는 예외를 발생시킨다
        AddToArray(24);
        Show( '프로그램은 절대로 여기에 도달하지 못한다' );
    except
    on EArrayFull do
        Show( '예외를 처리합니다' );
    end;
    Show( 'ButtonRaise1Click 호출이 완료되었다' );
end;
```

마지막에 있는 `Show` 호출은 두 번째 `Show` 가 호출된 직후에 실행된다. 첫 번째 `Show` 호출은 항상 무시된다. 여러분이 이 프로그램을 실행해보고, 코드를 바꾸면서 실험해 본다면, 예외 발생시의 프로그램 흐름을 완전히 이해하는데 도움이 될 것이다.

참고 예외를 처리하는 코드 위치와 예외를 발생시키는 코드 위치와 다르다는 점을 생각하면, 실제로 어떤 메서드가 그 예외를 발생시키는지 알 수 있어야 한다. 예외가 발생되었을 때 스택 추적 [stack trace](#) 을 얻고 그 정보를 핸들러 안에서 사용하도록 코드를 작성할 수 있다. 하지만, 정말 수준 높은 주제라서 이 책에서는 다루지 않을 것이다. 대부분의 경우, 오브젝트 파스칼 개발자들은 써드-파티에서 제공하는 라이브러리와 도구들(Jedi Component Library에서 제공하는 `JclDebug`, `madExcept`, `EurekaLog` 등)을 사용한다. 게다가, 컴파일러가 만드는 MAP 파일(애플리케이션 안에 있는 메서드와 함수 각각의 메모리 주소가 나열됨)을 가져와서 [generate](#) 코드에 포함시켜야 한다.

Finally 블록 The Finally Block

이제 예외 처리 키워드 중 네 번째 키워드인 `Finally` 를 보자. 앞에서 언급은 했지만, 사용한 적은 없다. `finally` 블록은 작업 중 항상 실행되어야 하는 것들(주로 정리 [clean-up](#) 작업들)을 수행하는데 사용된다. 실제로, `finally` 블록 안에 있는 문장은 예외 발생 여부와 관계없이 항상 처리된다. `try` 블록 뒤에 있는 평범한 코드는 예외가 발생하지 않거나, 예외가 발생하더라도 처리가 된 경우에만 실행된다. 바꿔 말해, `finally` 블록 안의 코드는 `try` 블록 코드 다음에 항상 실행된다. 예외가 발생한 경우에도 그렇다.

아래 메서드를 보자 (`ExceptFinally` 예제에서 발췌함). 오래 걸리는 연산을 진행하면서, 그 상태를 폼 [form](#) 의 캡션 [caption](#) 에 표시한다.

```
procedure TForm1.BtnWrongClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := '계산 중';
  J := 0;
  // 오래 걸리는 (그리고 잘못된) 컴퓨터 계산...
  for I := 1000 downto 0 do
    J := J + J div I;

  Caption := '계산 완료';
  Show( '합계: ' + J.ToString );
end;
```

위 코드는 알고리즘에 에러가 있기 때문에(변수가 0 값에 도달할 수 있는데, 그대로 나눗셈에 사용되기 때문에), 프로그램은 멈춘다. 그런데, 폼의 캡션은 재설정되지 않는다. 이런 때 사용하는 것이 `try-finally` 블록이다.

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := '계산 중';
  J := 0;
  try
    // 오래 걸리는 (그리고 잘못된) 컴퓨터 계산...
    for I := 1000 downto 0 do
      J := J + J div I;
    Show( '합계: ' + J.ToString );
  finally
    Caption := '계산 완료';
  end;
end;
```

프로그램은 이 함수를 실행하면서, (어떤 종류의) 예외이든, 발생 여부와 관계없이 항상 커서를 (옳긴: '캡션'이 맞는 것 같음) 재설정한다. 위 함수의 현재 단점은 예외 처리를 하지 않고 있다는 것이다.

Finally 와 Except

흥미롭게도, 오브젝트 파스칼 언어에서는 try 블록 뒤에 except 나 finally 문 중 하나가 올 수 있다. 하지만, 그 두 개가 동시에 올 수는 없다. 종종 두 블록을 모두 넣고 싶을 때가 있는데, 해결책은 try 블록 두 개를 중첩해서 사용하는 것이다. 안쪽 블록은 finally 문에 연결하고, 바깥쪽 블록은 except 문에 연결한다. 또는 상황에 맞게 반대로 연결한다. 아래 코드와 같다(ExceptFinally 예제에 있는 세 번째 버튼에 연결된 코드에서 발췌함).

```
procedure TForm1.BtnTryTryClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := '계산 중';
  J := 0;
  try try
    // 오래 걸리는 (그리고 잘못된) 컴퓨터 계산...
    for I := 1000 downto 0 do
      J := J + J div I;
      Show( '합계: ' + J.ToString);
  except
    on E: EDivByZero do
      begin
        // 새 메시지를 담아서 예외를 다시 발생시킨다
        raise Exception.Create( '알고리즘에 오류가 있음' );
      end;
    end;
  finally
    Caption := '계산 완료';
  end;
end;
```

Finally 블록을 사용해 커서 회복하기 Restore the Cursor with a Finally Block

try-finally 블록이 사용되는 일반적인 사례는 리소스 할당 [allocation](#) 및 해제 [release](#) 이다. 그런데, 또 다른 사례로는 작업을 완료한 후에, (그 작업이 예외를 발생시키더라도) 임시로 구성했던 것을 재설정해야 하는 경우가 있다.

임시로 구성한 것을 복원할 필요가 있는 경우의 예로 모래시계 커서를 들 수 있다. 긴 작업 중에는 모래시계가 표시되지만, 작업이 끝나면 원래대로 복원되어야 한다. 아무리 간단한 코드라도 예외가 발생할 가능성은 있다. 따라서, 언제나 try-finally 블록을 사용하는 것이 좋다.

RestoreCursor 애플리케이션 예제(VCL 애플리케이션임, 파이어몽키는 커서 관리가 다소 복잡하기 때문)에서 사용한 아래 코드는 현재 커서를 저장해 두고 임시로 모래시계 커서를 설정했다가, 마지막에 가서 원래 커서로 복원한다.

```
var CurrCur := Screen.Cursor;
Screen.Cursor := crHourGlass;
```



```

try
    // 약간 오래 걸리는 연산
    Sleep(5000);
finally
    Screen.Cursor := CurrCur;
end;

```

매니지드 레코드를 사용해 커서 회복하기 Restore the Cursor with a Managed Record

리소스 할당 보호 또는 임시 구성 복원 정의를 하고 싶을 때, try-finally 블록을 명시적으로 사용하는 방법 말고도, 매니지드 레코드 managed(관리되는) record 를 사용할 수 있다. 매니지드 레코드는 컴파일러에 의해 자체 intrinsic finally 블록이 추가된다. 그 결과 리소스 보호나 구성 복원을 하는 코드가 더 간결해진다. 비록 처음에 레코드를 정의하는 수고가 조금 더 들어가지만 말이다.

아래에 있는 매니지드 레코드는 앞에서 본 코드와 똑같은 동작을 한다. 현재 커서를 필드에 저장하는 코드는 Initialize 메서드 안에 있고, 그것을 재지정하는 코드는 Finalize 메서드 안에 있다.

```

type
    THourCursor = record
    private
        FCurrCur: TCursor;
    public
        class operator Initialize(out ADest: THourCursor);
        class operator Finalize(var ADest: THourCursor);
    end;

class operator THourCursor.Initialize(out ADest: THourCursor);
begin
    ADest.FCurrCur := Screen.Cursor;
    Screen.Cursor := crHourGlass;
end;

class operator THourCursor.Finalize(var ADest: THourCursor);
begin
    Screen.Cursor := ADest.FCurrCur;
end;

```

위에서 매니지드 레코드를 정의했으니, 이제 아래와 같이 사용하면 된다.

```

var HC: THourCursor;
    // 약간 오래 걸리는 연산
    Sleep(5000);

```

참고 매니지드 레코드를 통한 리소스 보호에 대한 보다 광범위한 예는 다음 블로그 게시물 (<https://blog.grijjy.com/2020/08/03/automate-restorable-operations-with-custom-managed-records/>)에서 볼 수 있다. 이 블로그는 매니지드 레코드에 대한 매우 상세한 블로그 시리즈 중 하나다.

실제 세상에 있는 예외 Exceptions in the Real World

예외 [exception](#) 는 에러 보고 및 에러 처리를 큰 틀에서 할 수 있는 훌륭한 메커니즘이다 (개별 코드 조각이 아니라 더 큰 아키텍처 안에서 수행된다). 일반적으로, 예외는 로컬 [local](#)(한 위치에 국한된) 에러 조건 확인을 대신하기 위한 용도가 아니다. (일부 개발자들이 예외를 이런 식으로 사용하기도 하지만).

예를 들어, 파일 이름이 분명하지 않다면, 파일이 정말 존재하는지 먼저 확인한 후에 파일을 여는 것이 더 좋은 방식이라고 여겨진다. 일단 파일 열기부터 하고 나서 예외를 사용하여 파일이 없는 경우를 처리하는 것보다 말이다. 이와 반대로, 파일에 쓰기 작업을 한다면, 그때마다 어디서나 디스크 공간이 충분한지 미리 확인하는 조건 검사 방식이 타당하다고 보기 어렵다. 공간이 부족한 경우는 극히 드물기 때문이다.

한 가지 기준을 정한다면, 프로그램에서는 일반적인 [common](#)(흔한) 에러 조건을 검사하고, 비정상적이고 예상치 못한 에러는 예외 처리 메커니즘에 맡기는 것이 좋다. 물론, 종종 이 두 가지 사이의 경계가 모호하고 개발자마다 판단하는 방식이 다를 수 있다.

그런데, 일상적일 코딩에서 너무나 중요하고, 매우 흔하게 사용되는 것이 있다. 바로 `finally` 블록을 사용하여 예외 상황에서 리소스를 보호하는 것이다. 리소스를 참조하는 블록은 항상 `finally` 문으로 보호해야 한다. 그래야 예외가 발생된 경우에 리소스 누출 [leak](#) 을 피할 수 있다. 궁극적으로, `finally` 문은 프로그램을 안정적으로 유지해준다. 에러가 발생하더라도, 사용자가 애플리케이션 사용을 지속할 수 있고 또는 (더 심각한 이슈인 경우에는) 질서정연하게 종료할 수 있다.

글로벌(전역) 수준에서 예외 다루기 Global Exceptions Handling

이벤트 핸들러 [event handler](#) 에서 예외가 발생되어서 표준 실행 흐름이 중단되었는데, 만약 예외 핸들러 [exception handler](#) 를 찾지 못한다면 프로그램이 종료될까? 콘솔 애플리케이션 또는 기타 특수 목적 코드 구조라면 종료되지만, (VCL 또는 파이어몽키 [FireMonkey](#) 라이브러리 기반 애플리케이션 등) 시각적 애플리케이션이라면 대부분 글로벌 [global/전역](#) 메시지 처리 루프가 있기 때문에 각 실행이 `try-except` 블록으로 감싸지고, 그 결과 이벤트 핸들러에서 예외가 발생하면 그 예외는 잡힌다 [trap](#).

참고 알아 둘 점이 있다. 기동 [start-up](#) 코드 안에서, 예외가 발생하면, 해당 메시지 루프가 활성화되기 전이므로 일반적으로 라이브러리가 그 예외를 잡지 못한다. 그래서 프로그램은 종료되고 [terminate](#) 에러가 발생한다. 이러한 문제 동작을 다소 완화하려면, 메인 프로그램 코드에서 사용자 지정 `try-except` 블록을 직접 추가한다. 하지만, 이런 수준에서 보호를 해도, 라이브러리 초기화 코드가 메인 프로그램이 실행되기보다 먼저, 즉 사용자 지정 `try-except` 블록에 진입하기 전,에 수행될 수도 있다. 따라서 처리되지 않은 예외가 그 전에 생길 수 있는 가능성은 여전히 있다.

실행 중에 예외가 발생하는 일반적인 경우, 어느 라이브러리를 사용하는지에 따라 다르지만, 대체로 프로그래밍 방식으로 예외 가로채기 또는 에러 메시지 표시하기 등을 할 수 있다. 세부 사항은 조금 다르지만, VCL과 FireMonkey 모두 해당된다.

앞에서 본 데모들은, 예외 발생시, 에러 메시지를 표시한다. 이 동작을 바꾸고 싶다면, 글로벌 [global/전역](#) 오브젝트인 Application에 있는 OnException 이벤트를 처리하면 된다. 이는 애플리케이션에서 시각적 라이브러리와 이벤트 처리 쪽에 가깝긴 하지만, 예외 처리과도 관련이 있으므로 지금 살펴보기로 하자.

앞에서 사용한 예제를 그대로 가져와 이름을 ErrorLog라고 바꾼 다음 다음과 같이 진행했다. 메인 폼에 새 메서드 하나를 아래와 같이 추가했다.

```
public
procedure LogException(Sender: TObject; E: Exception);
```

OnCreate 이벤트 핸들러 안에는, 글로벌 이벤트인 OnException 이벤트에 연결되는 메서드를 가로채는 코드를 추가했다. 그리고 나서, 새로 들어갈 글로벌 [\(전역\)](#) 핸들러의 실제 코드를 구현했다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Application.OnException := LogException;
end;

procedure TForm1.LogException(Sender: TObject; E: Exception);
begin
    Show( '예외: ' + E.Message);
end;
```

참고 메서드 포인터를 이벤트에 할당하는 방법(위 코드와 같다)에 대한 자세한 내용은 다음 장에서 알아본다.

이제 글로벌 예외 핸들러 안에 새 메서드가 있으므로, 프로그램은 에러 메시지를 출력 공간 [output](#)에 기록한다. 이제는 에러 대화 상자를 표시하느라 애플리케이션을 멈추지 [stop](#) 않아도 된다.

예외와 생성자 Exceptions and Constructors

예외를 둘러싼 조금 더 수준 높은 문제, 즉 오브젝트 생성자 [constructor](#) 안에서 예외가 발생하면 어떻게 될까? 이 상황에서 해당 오브젝트의 소멸자 [destructor](#)가 (사용 가능한 경우) 호출된다는 사실을 모든 오브젝트 파스칼 프로그래머들이 아는 것은 아니다.

그 사실을 아는 것은 중요하다. 일부만 초기화된 오브젝트를 위해서 소멸자가 호출될 수도 있다는 의미이기 때문이다. 내부 오브젝트들이 생성자 안에서 생성된다고 해서

당연히 소멸자 안에도 있다고 생각한다면, 실제 오류 발생 시 위험한 상황에 처할 수 있다(즉, 첫 번째 예외가 처리되기 전에 또다른 예외가 발생하는 상황).

또한, 오브젝트 생성은 try-finally 블록 바깥에서 하는 것이 적절한 순서라는 것을 뜻하기도 한다. 오브젝트 생성은 컴파일러가 자동으로 보호한다. 따라서 생성자에서 실패한 오브젝트를 Free 할 필요는 없다. 이런 점들로 인해, 오브젝트 파스칼에서 오브젝트를 보호하는 표준 코딩 스타일은 아래와 같다.

```
AnObject := AClass.Create;
try
    // 생성된 오브젝트를 사용한다...
finally
    AnObject.Free;
end;
```

참고 비슷한 일이 TObject 클래스의 두 메서드 즉 AfterDestruction과 BeforeConstruction에서 발생한다. 이 특수 메서드들은 의사-생성자와 의사-소멸자로써 C++ 호환성을 위해 도입되었다 (하지만, 오브젝트 파스칼에서는 거의 쓰이지 않음). AfterConstruction 메서드가 예외를 발생시키면, BeforeDestruction 메서드(와 일반 소멸자)가 호출된다는 점을 알아 두자.

소멸자 안에서 오브젝트를 바르게 처리하지 못하는 실수를 본 경험이 많았기 때문에, 그 문제를 더 명확히 할 수 있도록, 실제 데모를 통해서 그 문제와 실제 해결책을 보여주고자 한다. 아래 예제 코드는 스트링리스트 하나를 가지는 클래스와 그 클래스의 생성자와 소멸자다 (ConstructorExcept 프로젝트에서 발췌함).

```
type
    TObjectWithList = class
    private
        FStringList: TStringList;
    public
        constructor Create(Value: Integer);
        destructor Destroy; override;
    end;

constructor TObjectWithList.Create(Value: Integer);
begin
    if Value < 0 then
        raise Exception.Create('음수 값은 허용되지 않습니다');

    FStringList := TStringList.Create;
    FStringList.Add(Value.ToString);
end;

destructor TObjectWithList.Destroy;
begin
    FStringList.Clear;
    FStringList.Free;
    inherited;
end;
```


언뜻 보면, 올바른 코드다. 생성자는 하위-오브젝트를 할당하고 소멸자는 이를 적절히 폐기하고 있다. 또한, 이 클래스를 호출하는 아래 코드에서는 예외가 생성자 이후에 발생하면 Free 메서드를 호출하고, 예외가 생성자 안에서 발생하면 아무 일도 하지 않도록 잘 작성되어 있다.

```
var
  Obj: TObjectWithList;
begin
  Obj := TObjectWithList.Create(-10);
  try
    // 뭔가를 한다...
  finally
    Show( '오브젝트를 해제하는 중 ');
    Obj.Free;
  end;
```

그래서, 이게 잘 될까? 절대 아니다! 이 코드는 진행되다가, 생성자 코드에서 예외가 발생한다. 즉, 내부 스트링리스트를 생성하기 전에 예외가 생긴다. 시스템은 즉시 소멸자를 불러낸다. 그 소멸자 코드는 스트링리스트를 제거하려고 한다. 그런데, 스트링리스트가 생성되지 않았으니 존재하지 않는다. 따라서 액세스 위반 [access violation](#) 또는 그와 비슷한 에러가 발생한다.

왜 이런 일이 생길까? 다시 말하지만, 생성자 코드에서 순서를 바꾸면 (스트링리스트를 먼저 생성한 다음에 예외를 발생시키면) 모든 것이 올바르게 작동한다. 그렇게 되면, 소멸자가 스트링리스트를 해제 [free](#) 해야 하는 것이 맞기 때문이다. 하지만, 이는 제대로 된 수정 [real fix](#)이라고 할 수 없다. 그저 회피 방법 [workaround](#) 일 뿐이다. 언제나 소멸자 코드 보호를 명심해야 한다. 즉, 생성자 실행이 완전하다는 가정을 하지 말고 소멸자 코드를 작성해야 한다. 아래 예문처럼 작성하자.

```
destructor TObjectWithList.Destroy;
begin
  if Assigned(FStringList) then
  begin
    FStringList.Clear;
    FreeAndNil(FStringList);
  end;
  inherited;
end;
```

예외의 고급 기능들 [Advanced Features of Exceptions](#)

이 부분은, 이 언어에 대한 충분한 지식이 없다면, 처음 읽을 때는 건너뛰고 싶은 부분 중 하나일 것이다. 충분한 지식이 생기기 전까지는 다음 장으로 넘어갔다가 나중에 이 부분으로 다시 돌아와서 봐도 좋다.

이 장의 마지막 부분에서는, 예외 처리와 관련된 좀 더 수준 높은 주제를 다루겠다. 중첩된 예외(RaiseOuterException)와 클래스의 예외 가로채기(RaisingException)가 그것들이다.

이러한 기능들은 오브젝트 파스칼 언어의 초기 버전에는 포함되지 않았던 기능으로, 개발 시스템에 상당한 능력을 더해주는 것들이다.

중첩된 예외와 내부 예외 메커니즘 Nested Exceptions and the Inner Exception Mechanism

예외 핸들러 안에서 예외를 발생시키면 어떻게 될까? 새로 발생한 예외가 기존 예외를 대체한다는 것이 전통적인 대답이다. 그래서, 최소한 에러 메시지들이라도 결합하는 관행이 널리 퍼져 있다. 예를 들면, 아래와 같은 코드처럼 결합한다 (실제 코드는 생략하고 예외-관련 문장만을 남겨두었다).

```
procedure TFormExceptions.ClassicReraise;
begin
  try
    // 뭔가를 한다...
    raise Exception.Create('안녕');
  except on E: Exception do
    // 수정 조치를 시도한다...
    raise Exception.Create('하나 더: ' + E.Message);
  end;
end;
```

아래와 같이, 메시지를 호출하고 그 예외를 처리하면, 여러분에게는 단 하나의 예외가 남는다. 그런데, 그 예외의 메시지에는 두 예외의 메시지가 합쳐져 있을 것이다 (AdvancedExcept 예제에서 발췌함).

```
procedure TFormExceptions.BtnTraditionalClick(Sender: TObject);
begin
  try
    ClassicReraise;
  except
    on E: Exception do
      Show('메시지: ' + E.Message);
    end;
  end;
end;
```

(아주 당연하게도) 출력 결과는 다음과 같다.

메시지: 하나 더: 안녕

현재의 오브젝트 파스칼은, 중첩된 예외를 개발 시스템 전반에서 지원한다. 예외 핸들러 안에서 새 예외를 생성 해서 `create` 발생시켜도 `raise`, 현재 예외 오브젝트를 그대로 유지할 수 있다. 그리고, 그것을 새 예외와 연결할 수 있다. 그러려면, Exception 클래스의 InnerException 프로퍼티에서 이전 예외를 참조하도록 하고, BaseException

프로퍼티에서는 이어지는 그 예외들 중 가장 첫 번째 예외를 접근하도록 하면 된다 (예외 중첩은 재귀 반복이 될 수 있기 때문임). 중첩된 예외들을 관리하는 것과 관련된 Exception 클래스의 요소들은 다음과 같다.

```

type
  Exception = class(TObject)
  private
    FInnerException: Exception;
    FAcquireInnerException: Boolean;
  protected
    procedure SetInnerException;
  public
    function GetBaseException: Exception; virtual;
    property BaseException: Exception read GetBaseException;
    property InnerException: Exception read FInnerException;
    class procedure RaiseOuterException(E: Exception); static;
    class procedure ThrowOuterException(E: Exception); static;
  end;

```

참고 정적 클래스 메서드 `static class method`는 클래스 메서드의 특수한 형태다. 이 언어 기능은 12장에서 설명한다.

사용자 관점에서, 예외를 발생시키면서, 동시에 기존 예외는 보관 `preserve` 하고 싶으면, 여러분은 클래스 메서드인 `RaiseOuterException`(또는 `ThrowOuterException`: C++식 이름일 뿐 실제로는 같다)를 호출해야 한다. 여러분이 이와 비슷한 예외를 처리할 때, 위의 새 프로퍼티들을 사용하면 추가 정보에 접근할 수 있다. `RaiseOuterException` 메서드는 오직 예외 핸들러 안에서만 쓸 수 있다. 소스 코드 도움말에는 이 메서드를 다음과 같이 설명하고 있다.

예외 핸들러 안에서 새 예외 인스턴스를 발생시키고 `raise` 싶을 때 이 메서드를 사용하면, 활성화된 예외를 "획득"하고 그것을 새로 생성한 예외와 연결하고, 그 컨텍스트 `context(문맥)`를 보존할 수 있다. 현재 작동하고 있는 예외는 `FInnerException` 필드 안에 지정 `set` 된다.

이 프로시저는 오직 `except` 블록 안에서 사용해야 한다. 그리고, 이 새 예외를 처리하는 곳이 다른 어딘가에 있다고 예상되는 경우에 사용해야 한다.

실제 예제를 보려면 `AdvancedExcept` 예제를 보면 된다. 아래에서 나는 메서드 하나를 추가했다. 새로운 방식으로 중첩된 예외를 발생시킨다. (앞에서 본 `ClassicReraise` 메서드와 비교해 보기 바란다).

```

procedure TFormExceptions.MethodWithNestedException;
begin
  try
    raise Exception.Create('안녕');
  except
    Exception.RaiseOuterException(Exception.Create('하나 더'));
  end;
end;

```


이제 이 바깥쪽 예외를 처리하는 핸들러 안에서는 두 예외 오브젝트를 모두 접근할 수 있다 (이번에도 새 ToString 메서드를 호출해서 어떤 결과가 나오는지 보자).

```

try
  MethodWithNestedException;
except
  on E: Exception do
    begin
      Show( '메시지: ' + E.Message);
      Show( 'ToString: ' + E.ToString);
      if Assigned(E.BaseException) then
        Show( 'BaseException 메시지: ' + E.BaseException.Message);
      if Assigned(E.InnerException) then
        Show( 'InnerException 메시지: ' + E.InnerException.Message);
    end;
  end;

```

이 호출이 출력하는 결과는 다음과 같다.

```

메시지: 하나 더
ToString: 하나 더
안녕
BaseException 메시지: 안녕
InnerException 메시지: 안녕

```

위에서 두 가지 요소를 주목해야 한다. 첫째, (위와 같이) 중첩된 예외가 하나인 경우, BaseException 프로퍼티와 InnerException 프로퍼티가 모두 동일한 예외 오브젝트, 즉 원본 예외를 참조한다. 둘째, 새 예외의 메시지 프로퍼티에는 그 예외에 해당되는 실제 메시지만 들어가지만, ToString 을 호출하면 중첩된 모든 예외의 메시지가 합쳐 나온다(각 예외마다 sLineBreak 로 구분됨). Exception.ToString 메서드가 출력하는 결과를 보면 알 수 있다.

줄 바꿈을 사용했기 때문에 출력된 결과가 이상하게 보일 것이다. 하지만, 줄 바꿈에 대해 배우고 나면 원하는 심볼로 줄 바꿈을 교체하는 등 원하는 대로 형식을 반영할 수 있다. 또한, 전체 메시지를 스트링리스트의 Text 프로퍼티에 할당할 때도 좋다.

조금 더 깊은 예제를 보자. 이번에는 두 번 중첩되는 예외를 발생시키면 어떤 일이 생기는지를 보여주는 새 메서드다.

```

procedure TFormExceptions.MethodWithTwoNestedExceptions;
begin
  try
    raise Exception.Create( '안녕' );
  except
    try
      Exception.RaiseOuterException(Exception.Create( '하나 더' ));
    except
      Exception.RaiseOuterException(Exception.Create( '세 번째' ));
    end;
  end;
end;

```


앞에서 사용한 것과 똑같은 메서드를 사용해서 위 코드를 호출하면 다음과 같은 결과가 출력된다.

```
메시지: 세 번째
ToString: 세 번째
하나 더
안녕
BaseException 메시지: 안녕
InnerException 메시지: 하나 더
```

이번에는 `BaseException` 프로퍼티와 `InnerException` 프로퍼티가 서로 다른 오브젝트를 참조하고 있으며, `ToString`의 출력은 한 줄이 더 생겨 총 세 줄이 되었다.

예외 가로채기 Intercepting an Exception

오브젝트 파스칼 언어의 원래 예외 처리 시스템에 새로 추가된 또 다른 고급 기능은 바로 아래에 있는 메서드다.

```
procedure RaisingException(P: PExceptionRecord); virtual;
```

이것에 대해 소스 코드 도움말에는 아래와 같이 설명되어 있다.

이 가상 `virtual` 함수는 예외가 발생하기 직전에 호출된다. 외부 `external`의 델파이가 아닌 예외인 경우에는, 그 오브젝트가 생성된 후에 곧 호출된다. "발생 `raise`" 조건이 이미 진행 중이기 때문이다.

`Exception` 클래스 안에 있는 이 함수의 구현은 그 내부 예외 `inner exception`를 관리한다 (그러기 위해 내부적으로 `SetInnerException`이라는 메서드를 호출한다). 이는 이 함수가 왜 내부 예외 메커니즘 `Inner Exception Mechanism`과 동시에 가장 먼저 도입되었는지 말해준다.

어쨌든, 이제 우리는 이 기능을 사용할 수 있다. 그러니 잘 활용하면 된다. 우리가 이 메서드를 오버라이드 `override`하면, 실제로 단 하나의 생성-후 함수를 가지게 된다. 이 함수는 무조건 변함없이 호출된다. 그 예외를 생성할 때 사용되는 생성자와 무관하게 말이다. 즉, 여러분은 여러분의 클래스를 위해 사용자 지정 생성자를 정의하지 않아도 된다. 따라서 사용자들은 `Exception` 클래스의 많은 생성자들 중 무엇이든 호출할 수 있다. 그러면서도 사용자 지정 동작을 가질 수 있게 되는 것이다. 예를 들어, 여러분은 주어진 클래스(또는 하위클래스)의 모든 예외를 기록 `log`할 수 있다.

참고 호출되는 생성자와 상관없이 초기화 코드를 실행하는 또다른 방법은 `TObject`의 가상 메서드인 `AfterConstruction`을 오버라이드 하는 것이다.

아래는 사용자 정의 예외 클래스다. 이것은 `RaisingException` 메서드를 오버라이드 하고 있다 (이것도 `AdvancedExcept` 예제에 있는 정의에서 발췌함).


```

type
  ECustomException = class(Exception)
  protected
    procedure RaisingException(P: PExceptionRecord); override;
  end;

procedure ECustomException.RaisingException(P: PExceptionRecord);
begin
  // 예외 정보 기록하기
  FormExceptions.Show( '예외(Exception) 주소: ' + IntToHex(
    Integer(P.ExceptionAddress), 8));
  FormExceptions.show( '예외(Exception) 메시지: ' + Message);

  // 메시지 변경하기
  Message := Message + ' (필터링 된 것임)';

  // 표준 처리
inherited;
end;

```

위 메서드 구현은 예외에 대한 정보 몇 가지를 기록하고, 예외 메시지를 수정한 다음 기반 `base` 클래스의 표준 처리(중첩된 예외 메커니즘을 작동하기 위함)를 호출한다. `RaisingException` 메서드가 호출되는 시점은 예외 오브젝트가 생성된 `created` 후에, 그렇지만 예외가 발생되기 `raised` 보다는 전이다. 그 사실은 바로 알 수 있을 것이다. 위 `Show` 호출들을 통해 만들어지는 출력은 이 예외가 디버거에게 잡히기 전이다! 또한, 여러분이 `RaisingException` 코드 안에 중단점 `breakpoint` 를 넣으면, 디버거가 예외를 잡기도 전에 그 지점에서 중단된다.

다시 말하지만, 중첩된 예외 `nested exception` 그리고 예외 가로채기 `intercept` 메커니즘은 애플리케이션 코드에서 그리 많이 사용되지 않는다. 이 언어 기능은 라이브러리 및 컴포넌트 개발자에게 더 의미가 있다.

10: 프로퍼티와 이벤트 Properties and Events

이전 세 장에서는, 오브젝트 파스칼 언어로 OOP의 기초를 다뤘다. 그 개념을 설명하고 이 특징들이 대부분의 객체 지향 언어에서 어떻게 제공되고 있는지 보여주었다. 델파이 초창기부터, 오브젝트 파스칼 언어는 완전한 객체 지향 언어였다. 하지만 고유한 특징을 가지고 있다. 실제로, 오브젝트 파스칼은 컴포넌트 기반 시각적 개발 도구 언어로도 확장되었다.

이 둘(OOP와 컴포넌트-기반 시각적 개발 모델 [model](#))은 따로 떨어진 요소들이 아니다: 컴포넌트-기반 개발 모델에 대한 지원은 프로퍼티나 이벤트 같은 몇몇 핵심 언어 특징들을 기반으로 한다. 이것들은 다른 OOP 언어들보다 오브젝트 파스칼에서 가장 먼저 도입했다. 예를 들어, 프로퍼티는 다른 언어들 중 자바와 C#에도 있는데, 그 직접적인 조상은 오브젝트 파스칼이다. 하지만, 나는 오브젝트 파스칼의 원조 구현을 개인적으로 더 좋아한다. 이 점은 잠시 후에 설명하겠다.

오브젝트 파스칼의 빠른 애플리케이션 개발(Rapid Application Development, RAD)과 시각적 프로그래밍 지원 능력은 이 장에서 다룰 프로퍼티, published 접근 지정자, 이벤트, 컴포넌트 개념, 그리고 몇몇 다른 아이디어들이 나오게 된 이유다. 이 개발 모델은 종종 PME 모델(property-method-event model)이라는 용어로 설명되는데, RAD 접근 방식의 구체적인 구현 중 하나에 해당한다.

프로퍼티 정의하기 Defining Properties

프로퍼티 [property](#)란 무엇일까? 프로퍼티는 우리가 오브젝트의 상태에 접근하고 변경할 때 사용할 수 있도록 하는 식별자다 - 그 식별자 뒤에 코드를 두고, 그 코드를 작동한다. 오브젝트 파스칼에서, 프로퍼티는 필드나 메서드를 통한 데이터 접근을

추상화 *abstract* 하거나 숨긴다 *hide*. 따라서, 프로퍼티는 캡슐화를 구현하는 주요 방법이다. 프로퍼티를 "*캡슐화의 극대화(encapsulation to the max)*" 라고 설명하기도 한다.

기술적으로, 프로퍼티는 데이터 타입을 가지는 식별자다. `read` 및 `write` 지정자를 사용해 실제 데이터에 매핑된다. 자바나 C#과 달리, 오브젝트 파스칼의 `read` 와 `write` 지정자는 게터 *getter* 와 세터 *setter* 메서드를 연결해도 되고 필드 직접 가리켜도 된다.

예를 들어, 일반적인 접근 방식(필드에서 바로 읽고, 메서드를 통해 쓰기)을 사용하는 낱자 오브젝트에 있는 프로퍼티 정의를 보면 다음과 같다.

```
Private
  FMonth: Integer;
procedure SetMonth(Value: Integer);
public
property Month: Integer read FMonth write SetMonth;
```

위 코드는 Month 프로퍼티의 값에 접근하기 위해서 비공개 *private* 필드인 FMonth의 값을 읽는다. 그런데, 그 값을 변경하려면 SetMonth 메서드를 호출한다. 그 값을 변경하는 (그리고 음수 값을 가지지 않도록 방지하는) 코드는 다음과 같다:

```
procedure TDate.SetMonth(Value: Integer);
begin
  if (Value <= 0) or (Value > 12) then
    FMonth := 1
  else
    FMonth := Value;
end;
```

참고 입력이 잘못된 경우에는, 예를 들어 음수 값의 달을 입력하는 경우에는, 드러내지 않고 값을 교정하기보다는 (예외를 발생시켜서) 오류를 보여주는 것이 더 좋지만, 간단하게 예제를 보여주 고자 이 코드를 사용했다.

알아 둘 점이 있다. 필드와 프로퍼티는 그 데이터 타입이 반드시 정확히 일치해야 한다 (만약 다르면, 간단한 메서드를 사용해 타입을 변환해야 한다); 비슷한 이유로 세터 프로시저의 단일 파라미터와 게터 함수의 반환 값의 데이터 타입도 해당 프로퍼티의 타입과 일치해야 한다.

다르게 조합할 수도 있다 (예: 값을 `read` 할 때 메서드를 사용하기, `write` 지시어에서 값을 직접 바꾸기). 하지만, 메서드를 통해 프로퍼티의 값을 바꾸는 것이 일반적이다. 아래는 똑같은 프로퍼티를 조금 다르게 구현한 것이다.

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write FMonth;
```

참고 프로퍼티에 접근하는 코드를 작성할 때에는, 메서드가 호출될지도 모른다는 사실을 아는 것이 중요하다. 문제는 이 메서드 중 일부는 실행하는 데 시간이 걸리는 것들이 있을 수 있다는 점 이다; 또한, 몇 가지 부작용을 일으키는 것들도 있을 수 있다. 예를 들면 화면 위 컨트롤들을 변경하는 (느린) 작업 등이 발생할 수 있다. 이런 부작용들을 언급하는 문서는 거의 없다. 하지만, 우리는 그럴 수 있다는 점을 알아야 한다. 특히 코드를 최적화할 때에는 더욱 그렇다.

프로퍼티의 `write` 지시어를 생략해도 된다. 그러면 읽기 전용 프로퍼티가 된다.

property Month: Integer **read** GetMonth;

기술적으로, `read` 지시어를 생략해도 된다. 즉 쓰기 전용 `write-only` 프로퍼티를 만들 수도 있다. 하지만, 그건 보통 말이 되지 않고, 실제로도 그런 경우는 거의 없다.

다른 프로그래밍 언어들과 프로퍼티를 비교

자바나 C#과 비교해보자. 이 두 언어에서 프로퍼티는 메서드에 매핑 된다. 하지만, 자바는 암시적 매핑 `implicit mapping` (프로퍼티가 기본적인 호출 규약임)을 사용하는데 반해, C#은 명시적 매핑 `explicit mapping` 을 사용한다. 이 점은 오브젝트 파스칼과 같지만, 오직 메서드에만 매핑할 수 있다.

```
// 자바의 프로퍼티
private int mMonth;
public int getMonth() { return mMonth; }
public void setMonth(int value) {
    if (value <= 0)
        mMonth = 1;
    else
        mMonth = value;
}

int s = date.getMonth();
date.setMonth(s + 1);

// C#의 프로퍼티
private int mMonth;

public int Month {
    get { return mMonth; }
    set {
        if (value <= 0)
            mMonth = 1;
        else
            mMonth = value;
    }
}

date.Month++;
```

여기에서 프로퍼티가 주는 상대적 장점들을 다양한 프로그래밍 언어에서 심도 있게 비교하고 싶지는 않다. 하지만, 이 장의 처음에서 소개했듯이, 내 생각에 명시적으로 정의된 프로퍼티를 가지는 것은 유용한 아이디어이다. 또한, 프로퍼티를 필드에 매핑을 하는 것은, 메서드라는 추가 부담 없이도 더 높은 수준의 추상화를 얻을 수 있기 때문에 이 역시 있는 것이 매우 좋다.

이것이 바로 내가 왜 다른 언어에 비해 오브젝트 파스칼의 프로퍼티 구현을 더 좋아하는 이유다.

프로퍼티는 캡슐화(Encapsulation)를 구현한다

프로퍼티는 매우 건전한 OOP 메커니즘이자 캡슐화(encapsulation)의 아이디어를 제대로 고려한 적용 사례다. 근본적으로, 우리는 이름을 가지게 되는데, 그 이름은 클래스의 정보에 대한 접근이 어떻게 구현되어 있는지를 (해당 데이터를 직접 접근하는지 아니면 메서드를 호출하는지) 숨겨준다.

사실, 프로퍼티를 사용하면 우리는 변경될 가능성이 거의 없는 인터페이스를 얻게 된다. 동시에, 우리가 만든 클래스 내의 필드들 중 몇 가지만 사용자들이 접근하기 하는 것을 원할 경우, 그 필드들만 프로퍼티를 사용해 손쉽게 공개하면 되기 때문에, 그 필드들을 완전히 공개(public)로 지정하지 않아도 된다. 코드를 더 작성하지 않아도 된다. 그래도 여전히 클래스의 구현을 변경할 수도 있다. 이렇게 하고 나면, 데이터에 직접 접근하는 방식을 메서드 호출 방식으로 바꾼다 해도, 이 프로퍼티를 사용하는 소스 코드를 바꾸지 않아도 된다. 그저 다시 컴파일하기만 하면 된다.

프로퍼티는 캡슐화를 최대한의 능력까지 끌어 올리는 개념이라고 여기자!

참고 혹시, 프로퍼티가 private 변수에 직접 접근하도록 정의된 경우라면, 캡슐화의 장점 중 하나를 없애 버리는 것은 아닌지 궁금한가? 사용자는 그 private 변수의 데이터 타입이 변경되는 상황에서 보호받을 수 없다. 게터(getter)와 세터(setter)가 있으면 보호받을 수 있는데 말이다. 하지만, 사용자가 데이터에 접근하려면 프로퍼티를 통해야 한다는 점을 생각해보면, 이 클래스를 작성하는 사람은 언제나 기반이 되는 필드의 데이터 타입을 변경할 수 있고, 게터와 세터를 넣을 수도 있다. 그렇게 해도 이 클래스를 사용하는 쪽의 코드에는 영향을 주지 않는다. 이것이 바로 캡슐화를 극대화(encapsulation to the max)한다고 내가 말하는 이유다. 다른 한편으로, 이건 오브젝트 파스칼의 실용적인 측면을 보여주기도 한다. 프로그래머는 환경에 적합하다면 더 쉬운 방식(그리고 빠른 코드 실행)을 선택할 수 있다. 또한 필요한 경우, “적절한 OOP” 방식으로 부드럽게 전환할 수도 있다.

하지만 오브젝트 파스칼에서 프로퍼티를 사용할 때의 주의 사항이 있다. 우리는 종종 프로퍼티에 값을 대입(assign)하거나 그 값을 읽을 수 있다. 또한, 표현식 안에서 프로퍼티를 자유롭게 사용할 수 있다. 하지만, 프로퍼티를 프로시저나 메서드의 참조 파라미터(reference parameter)로 넘겨주지는 못한다. 프로퍼티는 메모리 위치가 아니라 추상화이기 때문이다. 따라서 우리는 프로퍼티를 참조(var) 파라미터로 쓸 수 없다. 예를 들어, C#과 달리 우리는 프로퍼티에 대해 Inc를 호출할 수 없다.

참고 이와 관련된 기능, 즉 프로퍼티를 참조로 넘겨주는 방법에 관해서는 이 장의 후반부에 다룬다. 하지만 그 기능은 거의 사용되지 않는다. 특별한 컴파일러 설정이 필요한 기능이며, 주류 기능은 절대 아니다.

프로퍼티를 위한 코드 완성 Code Completion for Properties

만일 클래스에 프로퍼티를 추가하는 작업이 지루하다면, IDE 에디터에서 다음과 같은 방식으로 손쉽게 프로퍼티를 *자동으로* 완성할 수 있다. 먼저 프로퍼티 선언의 시작 부분을 (클래스 안에) 다음과 같이 작성한다.

```
type
  TMyClass = class
    public
      property Month: Integer;
    end;
```

그리고 나서 프로퍼티 정의 부분에 커서를 두고 Ctrl+Shift+C 키 조합을 누르면 된다. 그러면, 클래스에 새 필드와 새 세터 메서드 setter method 가 추가된다. 그리고 그것들은 프로퍼티 정의 안에 들어가고 알맞게 매핑된다. 또한 세터 메서드의 구현도 완성된다. 그 기본 구현에는 필드의 값을 변경하는 코드가 들어 있다. 즉, 위 코드를 작성하고 키보드 단축키(또는 델파이 에디터 안의 해당 메뉴)를 누르면 아래 코드가 생긴다.

```
type
  TMyClass = class
    private
      FMonth: Integer;
      procedure SetMonth(const Value: Integer);
    public
      property Month: Integer read FMonth write SetMonth;
    end;

{ TMyClass }
procedure TMyClass.SetMonth(const Value: Integer);
begin
  FMonth := Value;
end;
```

게터 메서드 getter method 역시 원한다면, 프로퍼티 정의의 read 부분을 GetMonth 정의로 바꾸면 된다. 다음과 같다:

```
property Month: Integer read GetMonth write SetMonth;
```

다시 Ctrl+Shift+C를 누르면, 이 함수도 역시 추가된다. 다만 이번에는 값에 접근하는 코드가 미리 작성되어 들어가지는 않는다.

```
function TMyClass.GetMonth: Integer;
begin
end;
```

폼 form에 프로퍼티 추가하기

프로퍼티를 사용한 캡슐화의 구체적인 예제를 보도록 하자. 별도의 클래스를 만드는 대신, 이번에는 IDE 안에서 여러분이 만든 각 비주얼 폼들에 맞게, IDE 가 생성해주는 폼 클래스들을 수정한다. 클래스 완성 Class Completion은 이번에도 역시 활용한다.

애플리케이션 안에 여러 폼들이 있는 경우, 하나의 폼에 대한 정보를 다른 폼에서 접근 가능하게 하면 종종 편리하다. public 필드를 추가하고 싶은 유혹이 들 수 있겠지만, 그건 절대적으로 나쁜 생각이다. 하나의 폼의 정보를 다른 폼에게 보여주기를 원한다면, 프로퍼티를 쓰는 것이 더 좋다.

간단하게 폼 클래스 선언에 프로퍼티 이름과 타입을 적어준다:

```
property Clicks: Integer;
```

그리고 Ctrl+Shift+C 키 조합으로 코드 완성을 작동한다. 그러면 다음과 같이 된다.

```
type
  TFormProp = class(TForm)
  private
    FClicks: Integer;
    procedure SetClicks(const Value: Integer);
  public
    property Clicks: Integer read FClicks write SetClicks;
  end;
implementation

procedure TForm1.SetClicks(const Value: Integer);
begin
  FClicks := Value;
end;
```

이것은 타이핑하는 수고를 확실히 줄여준다. 이제 사용자가 폼을 클릭할 때 클릭 횟수를 하나 늘려보자. 아래 코드 한 줄을 적으면 된다. (FormProperties 예제의 OnMouseDown 이벤트 안에 있는 코드임)

```
Clicks := Clicks + 1;
```

FClicks 를 직접 증가시켜도 되는데 왜 그렇게 하지 않는 걸까? 글썄, 이 상황만 보면 그래도 된다. 하지만, 여러분은 SetClicks 메서드를 사용할 수도 있다. 그러면 사용자 인터페이스를 업데이트하고 현재의 값을 표시하기까지 한다. 만일 여러분이 프로퍼티를 건너뛰고 필드를 직접 접근한다면, 그 세터 메서드 안에 있는 추가 코드들 즉 사용자 인터페이스를 변경하는 코드는 실행되지 않는다. 그 결과, 화면이 동기화 되지 못한다.

캡슐화는 다른 장점도 있다. 다른 폼에서 여러분은 그 클릭 횟수를 적절히 추상화된 방법으로 참조할 수 있다. 사실, 폼 클래스 안에 있는 프로퍼티들은 개발자가 폼 안에 정의한 값들에 대한 접근을 위해 사용된다. 그런데 그 폼 안에 있는 컴포넌트들에 대한 접근을 캡슐화도 할 수 있다. 예를 들어, 레이블 [Label](#)에 정보를 표시하는 폼이 있는데, 다른 폼에서 그 레이블의 텍스트 변경하고 싶다면, 아마 이렇게 쓰고 싶을 것이다.

```
Form1.StatusLabel.Text := 'new text';
```

이건 흔한 방법이다. 하지만, 좋은 방법은 아니다. 폼의 구조 즉 컴포넌트에 대한 캡슐화를 제공하지 않기 때문이다. 이런 접근 코드들이 애플리케이션 여기저기에 있고,

나중에 그 폼의 사용자 인터페이스를 변경하고 싶다면 (예: 그 레이블이 업데이트될 때마다 추가하고 싶은 동작이 있는 경우), 그 많은 곳의 코드를 모두 고쳐야 할 것이다.

대안은, 메시지를 사용하거나 또는 더 좋게는 프로퍼티를 사용하는 것이다. 그래서 구체적인 컨트롤을 숨긴다. 다음과 같이 한다.

```
property StatusText: string read GetStatusText write SetStatusText;
```

위 코드를 작성하고 Ctrl+Shift+C 키 조합을 다시 누르면, 에디터는 그 프로퍼티를 읽고 쓰기 위한 메서드 두 개에 대한 정의를 자동으로 추가해준다.

```
function TFormProp.GetStatusText: string;
begin
    Result := LabelStatus.Text
end;

procedure TFormProp.SetStatusText(const Value: string);
begin
    LabelStatus.Text := Value;
end;
```

주의할 점이 있다. 위 경우, 프로퍼티는 클래스의 로컬 필드에 매핑된 것이 아니라 그 폼의 하위 오브젝트인 레이블에 매핑되어 있다(자동 코드 생성을 활용한다면, 여러분은 에디터가 개발자를 대신해 추가한 FStatusText 프로퍼티를 삭제하는 것을 잊지 말자).

프로그램의 다른 폼에서, 이제 이 폼의 StatusText 프로퍼티를 참조하기만 하면 된다. 그러면, 사용자 인터페이스가 바뀌는 경우, 이 프로퍼티의 Set 과 Get 메서드의 구현만 영향을 받는다. 이 폼의 다른 메서드에서 접근할 때에도 이 프로퍼티를 사용해야 한다. 그것이 내부 구현을 접근하는 것보다 더 좋다.

```
procedure TFormProp.SetClicks(const Value: Integer);
begin
    FClicks := Value;
    StatusText := FClicks.ToString + ' clicks';
end;
```

TDate 클래스에 프로퍼티를 추가하기

7 장에서, TDate 클래스를 만들었었다. 이제 프로퍼티를 사용하여 그것을 확장해보자. 이 새 예제는 DateProperties 며 7 장 ViewDate 예제를 확장한 것이다. 여기에는 (프로퍼티의 값을 넣거나 가져오는) 새 메서드 몇 개와 프로퍼티 4 개가 있다.

```
type
    TDate = class
    private
        FDate: TDateTime;
        function GetYear: Integer;
        procedure SetYear(const Value: Integer);
        function GetDay: Integer;
```



```

procedure SetDay(const Value: Integer);
function GetMonth: Integer;
procedure SetMonth(const Value: Integer);
public
  constructor Create; overload;
  constructor Create(Y, M, D: Integer); overload;
  procedure SetValue(Y, M, D: Integer); overload;
  procedure SetValue(NewDate: TDateTime); overload;
  function LeapYear: Boolean;
  procedure Increase(NumberOfDays: Integer = 1);
  procedure Decrease(NumberOfDays: Integer = 1);
  function GetText: string; virtual;
  property Day: Integer read GetDay write SetDay;
  property Month: Integer read GetMonth write SetMonth;
  property Year: Integer read GetYear write SetYear;
  property Text: string read GetText;
end;

```

위 Year, Day, Month 프로퍼티들은 값을 읽고 쓸 때 해당 메서드를 사용한다. Month 프로퍼티를 지원하기 위해 필요한 메서드들의 구현은 아래와 같다.

```

function TDate.GetMonth: Integer;
var
  Y, M, D: Word;
begin
  DecodeDate(FDate, Y, M, D);
  Result := M;
end;

procedure TDate.SetMonth(const Value: Integer);
begin
  if (Value < 1) or (Value > 12) then
    raise EDateOutOfRangeException.Create('Invalid month');
  SetValue(Value, Day, Year);
end;

```

SetValue 호출은 날짜를 실제로 인코딩한다. 범위를 벗어나는 날짜인 경우에는 항상 미리 작성해 놓은 사용자 지정 예외 [custom exception](#) 클래스를 사용하여 예외를 발생시킨다:

```

type
  EDateOutOfRangeException = class(Exception);

```

네 번째 프로퍼티인 Text 에는 읽기 메서드만 매핑 되어 있다. 이 함수는 virtual 로 선언되어 있다. 하위 클래스인 TNewDate 에서 교체할 수 있다. 프로퍼티의 Get 이나 Set 메서드는 (8 장에서 설명한 기능인) 나중에 바인딩하기 [late binding](#) 를 하지 않을 이유가 전혀 없다.

참고 이 예제에서 알아야 할 요점은 이 프로퍼티들이 값에 직접 매핑되지 않는다는 것이다. 이 프로퍼티들은 다른 타입으로 그리고 서로 다른 구조로 저장된 정보로부터 연산을 수행하는 것들이지 프로퍼티가 암시하는 것을 반영하는 것들이 아니다.

새 프로퍼티로 클래스를 업데이트했기 때문에, 이제 우리는 필요할 때 프로퍼티를 사용하도록 예제를 변경할 수 있다. 예를 들어, 우리는 Text 프로퍼티를 직접 사용할

수 있다. 그리고 에디트 박스 [Edit box](#) 를 사용하여 3 개의 주요 프로퍼티의 값을 사용자가 읽거나 쓰도록 할 수 있다. 아래는 BtnRead 버튼을 누르면 실제로 일어나는 일이다.

```
procedure TDateForm.BtnReadClick(Sender: TObject);
begin
    EditYear.Text := TheDay.Year.ToString;
    EditMonth.Text := TheDay.Month.ToString;
    EditDay.Text := TheDay.Day.ToString;
end;
```

BtnWrite 버튼은 정반대 동작을 한다. 이 코드는 두 가지 방식으로 쓸 수 있다.

```
// 프로퍼티들을 직접 사용
TheDay.Year := EditYear.Text.ToInteger;
TheDay.Month := EditMonth.Text.ToInteger;
TheDay.Day := EditDay.Text.ToInteger;

// 한 번에 모든 값을 변경
TheDay.SetValue(EditMonth.Text.ToInteger,
    EditDay.Text.ToInteger,
    EditYear.Text.ToInteger);
```

이 두 접근법의 차이는 입력 값이 유효한 날짜가 아닐 때와 관련이 있다. 각 값을 따로 넣는 경우에는, 프로그램은 년도를 바꾼 뒤 예외를 발생시키고 남은 코드를 건너뛸 수도 있을 텐데, 그러면 날짜는 일부만 변경된다. 모든 값을 한 번에 변경하는 경우에는, 모든 값이 올바르게 때문에 전부 바뀌거나, 또는 어느 값 하나라도 잘못되었기 때문에 그 날짜 오브젝트가 그냥 원래의 값을 그대로 유지하는 두 가지 경우 중 하나만 발생한다. 실제로 사실 바로 이 이유때문에 이 3 가지 프로퍼티는 읽기-전용 프로퍼티이어야 한다. 그래야 데이터 무결성 [data integrity](#) 를 지킬 수 있다.

배열 프로퍼티 사용하기 [Using Array Properties](#)

프로퍼티는 일반적으로 단일 값에 접근하게 한다. 복잡한 데이터 타입에서도 그렇다. 하지만, 오브젝트 파스칼은 배열 프로퍼티 [array property](#) (C#에서는 인덱서 [indexer](#) 라고 부름) 또한 지원한다. 배열 프로퍼티에는 추가 파라미터가 있다. 그 추가 파라미터는 실제 값을 가리킬 인덱스 [index](#) 또는 선택자 [selector](#) 이며, 무슨 타입이든 사용할 수 있다.

여기에 정수 값을 가리키는 정수를 인덱스로 사용하는 배열 프로퍼티의 정의를 보여주는 예제가 있다.

```
private
function GetValue(I: Integer): Integer;
procedure SetValue(I: Integer; const Value: Integer);
public
property Value[I: Integer]: Integer read GetValue write SetValue;
```

배열 프로퍼티는 반드시 읽기 및 쓰기 메서드들에 매핑 되어야 한다. 매핑되는 메서드에는 추가 파라미터가 있어야 하는데, 인덱스를 명시해야 하기 때문이다. 일반적인 프로퍼티인 경우, 코드 완성을 사용해 이 메서드들을 정의할 수 있다.

값과 인덱스 조합은 많은 곳에서 볼 수 있다. 또한 RTL 안에 있는 몇몇 클래스들은 배열 프로퍼티를 상당히 많이 사용하기도 한다. 예를 들어, TStrings 클래스는 배열 프로퍼티 5 개를 사용한다.

```
property Names[Index: Integer]: string
    read GetName;
property Objects[Index: Integer]: TObject
    read GetObject write PutObject;
property Values[const Name: string]: string
    read GetValue write SetValue;
property ValueFromIndex[Index: Integer]: string
    read GetValueFromIndex write SetValueFromIndex;
property Strings[Index: Integer]: string
    read Get write Put; default;
```

이런 배열 프로퍼티 대부분은 문자열의 인덱스를 리스트의 파라미터로 사용하지만, 어떤 배열 프로퍼티(위 예제의 경우 Values)들은 문자열 파라미터를 룩업 [look-up](#) 값이나 검색 [search](#) 값으로 사용하기도 한다.

배열 프로퍼티 정의는 맨 뒤에 중요한 요소를 하나 더 사용한다: 바로 default 키워드다. 이것은 매우 강력한 구문 헬퍼 [syntax helper](#) 이다: 배열 프로퍼티의 이름을 생략할 수 있게 해준다. 그래서, 대괄호 연산자를 해당 오브젝트에 직접 붙일 수 있다.

가령 TStrings 타입인 SList 오브젝트의 경우, 다음 코드는 둘 다 작동한다.

```
SList.Strings[1]
SList[1]
```

즉, default 를 가지는 배열 프로퍼티는 어떤 오브젝트에든 [] 연산자를 정의할 수 있는 방법을 제공한다.

프로퍼티를 참조로 설정하기 [Setting Properties by Reference](#)

이 부분은, 오브젝트 파스칼에 익숙하지 않다면, 건너뛰는 것이 좋을 수 있을 정도로 적게 쓰이는 고급 기능 주제를 다룬다. 그러나 만약 여러분이 오브젝트 파스칼에 익숙하다면 들어본 적 없을 기능에 관한 내용을 보는 기회가 될 것이다

오브젝트 파스칼 컴파일러가 윈도우 COM [Component Object Model](#) 프로그래밍을 직접 지원할 수 있게 확장되었을 때, "(COM 용어로) 참조로 넣는 [put by ref](#) 프로퍼티" 즉 값이 아니라 참조를 받을 수 있는 프로퍼티라는 기능이 추가되었다.

참고 "참조로 넣는 [Put by ref](#)" 이란 말은 Chris Bensen이라는 프로그래머가 (텔파이 연구개발 엔지니어로 재직할 때) 자신의 블로그 게시물에 소개한 이 기능에 붙인 이름이다:
<http://chrisbensen.blogspot.com/2008/04/delphi-put-by-ref-properties.html>

이 기능은 세터 메서드에 `var` 파라미터를 사용하여 쓸 수 있다. 이것이 상당히 이상한 상황을 만들 수도 있다는 점에서, (여전히 이 언어의 일부이기는 하지만) 이 기능은 언어 규칙보다는 예외로 여겨진다. 그래서 기본적으로 활성화되지 않다. 즉, 이 기능을 사용하려면, 다음 컴파일러 지시어를 명시하여 활성화해야 한다.

```
{ $VARPROPSETTER ON }
```

이 지시어가 없으면 아래 코드는 컴파일 되지 않으며 *"E2282 Property setters cannot take var parameters"* 오류를 낸다.

```
type
  TMyIntegerClass = class
  private
    FNumber: Integer;
    function GetNumber: Integer;
    procedure SetNumber(var Value: Integer);
  public
    property Number: Integer read GetNumber write SetNumber;
end;
```

이 클래스는 `VarProp` 예제에 들어있다. 여기서 기이한 것은 `Number` 프로퍼티의 세터 안에서 일종의 부작용이 발생하는 것이다.

```
procedure TMyIntegerClass.SetNumber(var Value: Integer);
begin
  Inc(Value); // 부작용 발생
  FNumber := Value;
end;
```

흔치 않은 다른 효과가 있다. 여러분은 이 프로퍼티에 상수 값을 대입 [assign](#) 할 수 없다. 오직 변수만 대입할 수 있다 (파라미터를 참조로 전달 [pass by reference](#) 하므로 당연하다).

```
var
  Mic: TMyIntegerClass;
  N: Integer;
begin
  ...
  Mic.Number := 10; // Error: E2036 Variable required
  Mic.Number := N;
```

늘 사용하는 기능이 아니지만, 여러분이 프로퍼티를 통해 할당되는 값을 변경 또는 초기화하는 다소 고급 기술을 생각해 볼 수 있었을 것이다. 이것은 다음과 같이 정말 이상한 코드로 이어질 수 있다.

```
  N := 10;
  Mic.Number := N;
  Mic.Number := N;
  Show(Mic.Number.ToString);
```

위에서는 두 번 연속 똑 같은 대입을 하고 있다. 상당히 이상하게 보이겠지만 실제 숫자를 12 로 바꾸는 특이한 효과를 만든다. 이 결과를 얻기 위해 위와 같이 하는 것은 아마도 가장 복잡하면서 직관적이지 않은 방법일 것이다!

published 접근 지정자 Setting Properties by Reference

public, protected, private (그리고 상대적으로 덜 쓰이는 strict private 와 strict protected) 접근 지시어 외에, 오브젝트 파스칼 언어는 특이하게 published 라는 지시어가 하나 더 있다. published 프로퍼티 (또는 필드, 메서드)는 public 처럼 런타임에 접근 가능할 수 있다. 그런데 거기에서 그치지 않고, 확장된 런타임 타입 정보 [Extended RTTI](#) 를 제공하기 때문에, 런타임에 쿼리(질의)를 처리할 수도 있다.

컴파일 언어에서, 컴파일 되는 심볼들은 컴파일러가 처리한다. 그리고 애플리케이션을 시험할 때는 디버거가 심볼들을 사용하지만, 런타임에서는 아무런 흔적을 남기지 않는다. 즉 (적어도 초기의 오브젝트 파스칼 언어부터) 만일 어떤 클래스에게 Name 이라는 프로퍼티가 있다면 개발자는 코드 안에서 그것을 사용하여 해당 클래스와 상호작용을 할 수 있지만, 런타임에는 그 상호작용이 숫자로 된 주소 값으로 변환되므로, 식별자라는 소스 코드 수준의 개념이 더 이상 유효하지 않다. 따라서 "Name"이라는 문자열과 일치하는 프로퍼티 식별자가 그 클래스 안에 있는지를 런타임에는 알 방법이 없다.

참고 자바와 C#은 둘 다, 컴파일 언어이긴 하지만, 복잡한 가상 실행 환경을 가지므로, 그 장점을 활용하여 방대한 런타임-정보를 가질 수 있다. 이 개념은 리플렉션([reflection](#))이라고 불린다. 오브젝트 파스칼 언어는 (published 키워드를 통해서) 초창기부터 기본적인 [basic](#) 리플렉션을 가지고 있었다. 이는 컴파일되는 다른 언어들에 비해 훨씬 앞선 시점이다. 그 후, 리플렉션에 더 많은 정보를 넣어서, 지금은 확장된 [extended](#) RTTI를 추가로 가지고 있다. 이 장에서는 published 키워드로 제공되는 기본적인 RTTI를 다룰 것이며 확장된 RTTI는 16장에서 다룰 것이다.

어째서 클래스에 관한 이런 추가적인 정보가 필요할까? 그것은 오브젝트 파스칼 라이브러리가 의존하는 컴포넌트 모델 [component model](#) 과 시각적 프로그래밍 모델 [visual programming model](#) 의 기반 중 하나이기 때문이다. 이런 정보 중 일부는 개발 환경에서 디자인을 할 때 오브젝트 인스펙터 [Object Inspector](#) 안에 해당 컴포넌트가 제공하는 프로퍼티를 목록으로 채우는 데에 사용된다. 이 목록은 하드-코딩 된 것이 아니다. 컴파일 된 코드와 함께 있는 추가적인 RTTI 데이터를 런타임 시점에 검사해서 만들어 내는 목록이다.

지금 완전히 살펴보기에는 조금 복잡할 수 있는 또다른 예제로 FMX 및 DFM, 그리고 그에 동반되는 비주얼 폼 [visual forms](#) 파일들의 생성 과정에 숨겨진 스트리밍 [streaming](#) 메커니즘이 있다. 스트리밍은 핵심 언어 요소 라기보단 런타임 라이브러리의 요소이기에 18 장에서만 다룰 것이다.

이 개념을 요약하자면, 여러분이나 다른 사람이 개발 환경에서 쓸 컴포넌트를 작성한다면, published 키워드를 적극적으로 사용하는 것은 중요하다. 나중에 다루겠지만 컴포넌트의 published 부분에는 오직 프로퍼티만 넣는 것이 일반적이다. 하지만, 폼 클래스에서는 published 필드와 published 메서드들도 활용한다.

디자인할 때의 프로퍼티 Design-Time Properties

이 장의 초반에 우리는 프로퍼티가 클래스 내 데이터를 캡슐화 하는데 중요한 역할을 한다는 것을 보았다. 프로퍼티는 또한 시각화 개발 모델을 사용하는 데 중요한 역할을 한다는 것도 보았다. 개발자는 실제로 컴포넌트 클래스를 작성하고, 그것을 개발 환경에서 사용할 수 있도록 하고, 폼이나 유사한 디자인 화면에 그것을 추가하면 오브젝트가 만들어 지도록 하고, 그 프로퍼티들을 오브젝트 인스펙터 (시각적 프로그래밍 환경에서 프로퍼티를 접근할 때 사용하는 도구)를 통해 다룰 수 있도록 할 수 있다. 모든 프로퍼티가 그런 상황에서 쓰이는 것은 아니며, 컴포넌트 클래스에서 `published` 로 지정된 것들만 그렇게 쓸 수 있다. 그렇기 때문에 오브젝트 파스칼 프로그래머들이 *디자인할 때의* 프로퍼티와 *런타임 전용* 프로퍼티를 구분한다.

디자인할 때의 프로퍼티들은 클래스 선언의 `published` 구역 안에 선언되며 통합 개발 환경 IDE의 디자인 화면 또는 코드 에디터에서 사용된다. `public` 구역 안에 선언된 다른 프로퍼티들은 디자인 화면에서는 사용할 수 없으며 코드 안에서만 사용될 수 있는데, 이를 *런타임 전용* runtime only이라고 부른다.

달리 말하면, 디자인 화면에서 우리는 `published` 프로퍼티의 값을 오브젝트 인스펙터를 통해서 볼 수 있고 바꿀 수도 있다. 런타임에서, 개발자는 다른 클래스에서 `public` 이나 `published` 로 선언된 모든 프로퍼티를 같은 방식으로 접근할 수 있으므로 코드 안에서 읽고 쓸 수 있다.

모든 클래스가 프로퍼티를 가지는 것은 아니다. 프로퍼티는 컴포넌트와 `TPersistent` 클래스의 하위클래스 subclass에 제공되는데, 주로 프로퍼티들은 스트리밍 되고 파일로 저장되기 때문이다. 사실, 폼 파일은 그저 그 폼 안에 있는 컴포넌트들이 가진 `published` 프로퍼티들의 모음일 뿐이다.

보다 정확하게 말하자면, 개발자는 `published` 구역이라는 개념을 지원하기 위해 반드시 `TPersistent`를 상속해야 하는 것은 아니다. 하지만, 클래스를 컴파일할 때 `$M` 컴파일러 지시어 compiler directive를 반영해야 한다. 이 지시어를 넣거나 지시어가 반영되어 컴파일 된 클래스 또는 그런 클래스를 상속받은 각 클래스는 모두 `published` 구역을 지원한다. `TPersistent`는 이 설정이 반영되어 컴파일 된 클래스이기 때문에, 그것을 상속한 다른 클래스도 `published` 구역을 지원한다.

참고 아래에 이어지는 두 소단원은 폼의 기본 가시성 visibility과 자동 RTTI를 설명한다. 거기에서는 `$M` 지시어와 `published` 키워드의 효과에 대한 더 많은 내용을 다룰 것이다.

published와 폼 Published and Forms

통합 개발 환경이 폼을 만들어낼 때, 컴포넌트와 메서드의 정의는 폼의 첫 부분에 위치한다, 즉 `public`과 `private` 키워드 보다 앞쪽에 있다. 이렇게 클래스 시작 부분에 놓인 필드와 메서드들은 `published` 이다.

주의할 점이 있다. 이들이 전역으로 `globally` 외부에 노출된다고 해서, 이것들을 전역으로 접근하는 것은 별로 좋은 생각이 아니다. (이 소단원 앞쪽에서, 폼에 프로퍼티를 추가하기를 설명하면서, 이미 언급했던 바와 같이) 전역으로 접근하는 것들에 대해서는 특히 주의를 기울여야 하기 때문이다. 또한, 컴포넌트 클래스의 요소 `element` 앞에 명시적인 `explicit` 접근 지시어가 명시될 때, 기본 접근 수준이 `published` 라는 점에도 주의해야 한다.

참고 보다 정확하게 말하자면, `published`는 어떤 클래스가 `$M+` 컴파일러 지시어와 함께 컴파일 되거나 그런 클래스의 자식인 경우에서만 기본 키워드다. `TPersistent` 클래스 안에서 이 접근 지시어가 사용되기 때문에, 델파이 라이브러리에 있는 대부분의 클래스와 모든 컴포넌트 클래스들은 `published`가 기본 키워드다. 하지만 (`TStream`이나 `TList` 같이) 컴포넌트 클래스가 아닌 것들은 `$M-` 컴파일러 지시어로 컴파일 되므로 기본으로 `public` 가시성을 가진다.

여기 예제가 있다:

```
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    BtnTest: TButton;
```

이벤트에 대입 `assign` 되는 메서드들은 모두 `published` 메서드여야 한다. 그리고, 컴포넌트에 대응하는 폼 안에서 있는 필드 역시 `published` 여야만 자동으로 폼 파일 안에 기술되는 해당 오브젝트에 자동으로 연결되고, 폼이 생성될 때 함께 생성된다.

폼 선언의 첫 `published` 부분에 있는 컴포넌트와 메서드들만 오브젝트 인스펙터 안에 나타난다(그 폼의 컴포넌트 목록, 이벤트를 위해 드롭다운 리스트 `drop-down list` 를 선택했을 때 표시되는 사용 가능한 메서드 목록 등).

왜 클래스 컴포넌트들을 `published` 필드로 선언해야 할까? 비공개 `private` 로 선언하면 OOP 의 캡슐화 원칙을 더 잘 따르게 할 수 있는데도 말이다. 그 이유는 이들 컴포넌트들은 스트리밍 된 표현 `streamed representation` 을 읽어 생성되지만, 일단 생성되고 나면, 자신에게 해당되는 폼 필드에 대입되어야 하기 때문이다.

이것은 RTTI 를 사용해 수행되는데, RTTI 는 `published` 필드에 대해서 생성된다.

참고 기술적으로, 컴포넌트에 `published` 필드를 사용하는 것이 필수는 아니다. 비공개 필드로 만들어서 OOP 상식에 더 잘 맞게 할 수 있다. 하지만, 그러려면 추가적인 런타임 코드가 필요하다. 이것은 이 장의 마지막 소단원인 "RAD와 OOP"에서 조금 더 다룬다.

자동 RTTI Automatic RTTI

오브젝트 파스칼 컴파일의 또다른 특별한 동작이 있다. 개발자가 `TPersistent` 를 상속받지 않은 클래스에 `published` 키워드를 추가할 경우, 컴파일러는 자동으로 `{$M+}` 동작을 추가해서 RTTI 생성을 활성화한다.

이런 클래스가 있다고 가정하자.

```
type
  TMyTestClass = class
  private
    FValue: Integer;
    procedure SetValue(const Value: Integer);
  published
    property Value: Integer read FValue write SetValue;
end;
```

컴파일러는 다음 경고 메시지를 보여준다.

```
[dcc32 Warning] AutoRTTIForm.pas(27): W1055 PUBLISHED caused RTTI
($M+) to be added to type 'TMyTestClass'
```

이처럼 컴파일러가 자동으로 `{$M+}` 지시어를 코드에 삽입한다는 것을 볼 수 있다. `AutoRTTI` 예제에 위 코드가 들어 있다. 이 프로그램에서는 다음과 같은 코드를 작성할 수 있다. 아래 코드는 (옛날 방식인 `System.TypInfo` 유닛을 사용하여) 프로퍼티에 동적으로 `dynamically` 접근한다.

```
uses
  TypeInfo;
procedure TFormAutoRtti.BtnTetClick(Sender: TObject);
var
  Test1: TMyTestClass;
begin
  Test1 := TMyTestClass.Create;
  try
    Test1.Value := 22;
    Memo1.Lines.Add(GetPropValue(Test1, 'Value'));
  finally
    Test1.Free;
  end;
end;
```

참고 위 예제에 `TypeInfo` 유닛이나 그 안에 정의된 `GetPropValue` 같은 함수를 쓰긴 했지만, RTTI 접근에 대해 진정으로 강력한 힘을 얻으려면, 보다 현대화된 RTTI 유닛 그리고 그 유닛을 통한 확장된 리플렉션 `reflection` 지원을 활용한다. 이것은 보다 복잡한 주제이기 때문에 나는 별도의 장을 할애하는 것이 중요하다고 생각했다. 또한, 현대의 오브젝트 파스칼에서 사용할 수 있는 두 종류의 RTTI를 구별할 필요가 있었다.

이벤트 기반 프로그래밍 Event-Driven Programming

컴포넌트 기반 라이브러리(또는 다른 많은 상황)에서 우리가 작성하는 코드는 그저 단순한 순서를 따르는 동작 **action** 의 연속 **sequence** 이 아니라, 반응 **reaction** 들의 모음이 대부분이다. 즉 개발자는 어떤 일이 일어났을 때 애플리케이션이 어떻게 "반응"해야 하는지를 정의한다는 의미이다. 그 "어떤 일"이란 마우스 버튼 클릭 같은 사용자의 작동 **operation** 일 수도 있고, 시스템 명령일 수도 있으며, 센서의 상태 변화 또는 원거리 통신 연결 **remote connection** 을 통해 받은 어떤 데이터 등 어떤 것이든 될 수 있다.

이들 외부 혹은 내부의 동작 발동 **triggers of actions** 은 흔히 이벤트 **event** 라고 불린다. 이벤트는 원래 윈도우 같은 메시지 기반 **message-oriented** 운영체제에 있는 일종의 매핑 **mapping** 이지만 원래의 개념으로부터 정말 많은 변화를 거쳤다. 현대 라이브러리들을 보면, 이벤트 대부분은 내부에서 발동 **trigger** 한다. 개발자가 프로퍼티를 설정할 때, 메서드를 호출할 때, 어떤 컴포넌트와의 (혹은 간접적으로 컴포넌트와 다른 컴포넌트 간의) 상호작용을 할 때 등이다.

어떻게 이벤트와 이벤트 기반 프로그래밍이 OOP 와 관련되는가? 두 접근법은 더 일반적인 접근법을 위해, 상속된 새 클래스를 만들려고 할 때 언제, 어디서 만들어야 할지 결정하는 측면에서 차이가 있다.

순수한 객체 지향 프로그래밍 형태에서는, 두 오브젝트가 서로 다른 동작 **behavior** (혹은 다른 메서드)을 가지는 경우, 서로 다른 클래스에 속해야 한다. 우리는 이미 이와 관련한 많은 예제를 보았다.

그러나 이런 상황을 가정하자. 폼에 버튼 4 개가 있고 각 버튼을 클릭하면 서로 다른 동작 **behavior** 을 해야 한다. 순수한 객체 지향 프로그래밍의 관점에서, 개발자는 "클릭" 메서드의 서로 다른 버전을 위해 서로 다른 서브클래스 4 개를 만들어야 한다. 공식적으로는 이 방식이 옳다. 하지만, 그렇게 하면 엄청나게 많은 코드를 쓰고 유지관리해야 하므로 복잡도 **complexity** 가 증가할 것이다.

이벤트 기반 프로그래밍은 이와 비슷한 상황을 가정하고 (클래스가 똑같은) 여러 버튼 오브젝트들에 개발자가 몇 가지 동작을 추가할 수 있도록 한다. 이런 동작은 해당 오브젝트의 상태 **status** 에 대한 장식 **decoration** 또는 확장 **extension** 이다. 그래서 새 클래스가 필요 없다. 이 모델은 델리게이션 **delegation/위임** 이라고도 부른다. 왜냐하면, 어느 오브젝트의 동작이 그 오브젝트 자체의 클래스가 아니라 다른 클래스의 메서드에게 위임되기 **delegated** 때문이다.

이벤트는 프로그래밍 언어에 따라 서로 다른 방식으로 구현된다. 예를 들어:

- 메서드에 대한 참조 **reference** (오브젝트 파스칼에서는 메서드 포인터라고 부른다) 또는 내부 메서드를 가진 이벤트 오브젝트에 대한 참조를 사용한다 (C#의 경우)
- 인터페이스를 구현하는 특별한 클래스에 이벤트 코드를 위임한다 (자바 **Java**의 경우)

- 클로저 [closure](#)를 사용한다. 자바스크립트가 이 방식을 사용한다. (이 방식 역시 오브젝트 파스칼에서 익명 메서드를 통해 지원한다. 15장에서 다룰 것이다). 다만 자바스크립트에선 모든 메서드가 클로저이므로 이 두 개념의 구분이 조금 모호하다.

이벤트와 이벤트 기반 프로그래밍 개념은 상당히 흔하게 사용된다. 그리고 다른 많은 프로그래밍 언어와 사용자 인터페이스 라이브러리들이 지원하고 있다. 그러나 델파이에서 이벤트 지원을 구현한 방식은 상당히 독특하다. 다음 절에서 이벤트를 뒷받침하는 기술에 대해 자세히 설명한다.

메서드 포인터 [Method pointers](#)

4장의 마지막 부분에서, 우리는 오브젝트 파스칼 언어에는 함수 포인터 [function pointer](#) 라는 개념이 있다는 것을 보았다. 함수 포인터는 변수인데 함수의 메모리 위치를 담는다. 따라서 우리는 그 변수를 사용하여 그 변수가 가리키는 함수를 바로 호출할 수 있다. 함수 포인터 선언에는 (파라미터 타입들의 집합 그리고 만약 있다면 해당 반환 타입까지) 명시된 서명 [signature](#) 이 붙는다.

이와 비슷하게, 오브젝트 파스칼에는 메서드 포인터 [method pointer](#) 라는 개념이 있다. 메서드 포인터는 클래스에 속한 메서드가 차지한 메모리 위치에 대한 참조 [reference](#) 이다. 함수 포인터 타입처럼 메서드 포인터 타입에도 명시된 서명이 붙는다. 하지만, 메서드 포인터에는 그보다 더 많은 정보가 전달된다. 즉, 그 메서드가 적용될 오브젝트가 전달된다 (즉, 그 메서드가 호출될 때 `Self` 파라미터로 전달되어 사용될 오브젝트가 자신이라고 알려준다).

다르게 표현하자면, 메서드 포인터는 메서드에 대한 참조인데 (그 메서드가 있는 위치는 해당 클래스의 모든 오브젝트들이 공유한다), 메모리 안에 있는 특정 오브젝트 하나(특정 메모리에 자신의 데이터를 담고 있는 특정 인스턴스)를 대상으로 작동한다. 만일 개발자가 메서드 포인터에 어떤 값을 대입 [assign](#) 할 경우, 그 개발자는 반드시 주어진 오브젝트의 메서드(즉, 특정 인스턴스의 메서드)를 담아야 한다!

참고 메서드 포인터가 어떻게 구현되는지 더 잘 이해하려면 `TMethod`의 데이터 구조의 정의 안에서 저-수준에서 이 구조를 표현하고 있는 부분을 보기 바란다. `TMethod` 레코드 [record](#)는 `Code`와 `Data` 라는 두 필드 [field](#)를 가지고 있는데, `Code` 필드는 메서드 주소를 나타내고, `Data` 필드는 (그 메서드 포인터를 적용할) 오브젝트를 나타낸다. 이처럼 코드에 대한 참조 [code reference](#)는, 다른 유사한 언어에서는, 델리게이트 클래스 [delegate class](#) (C#) 또는 인터페이스의 메서드 (Java)에 묶인다.

메서드 포인터 타입의 선언은 함수 타입 [procedural type](#) 선언과 비슷하지만, 선언의 끝에 `of object` 라는 키워드가 추가로 붙는다.

```
type
TIntProceduralType = procedure(Num: Integer);
TStringEventType = procedure(const S: string) of object;
```


위와 같이 메서드 포인터 선언을 하고 나면, 이제 이 타입에 해당하는 변수를 선언하고, 어떤 오브젝트의 메서드이든 호환되는 [compatible](#) 메서드를 그 변수에 대입하면 된다. 그럼 호환되는 메서드는 무엇일까? 그것은 메서드 포인터 타입이 요구하는 파라미터들을 그대로 가지고 있는 메서드다. 위 예제를 놓고 보면, 문자열 파라미터 하나를 가지는 모든 메서드가 해당된다.

이제 메서드 포인터 타입이 있으니 아래와 같이, 이 타입의 변수를 선언하고 그 변수에 호환되는 메서드를 대입할 수 있다:

```
type
  TEventTest = class
  public
    procedure ShowValue(const S: string);
    procedure UseMethod;
  end;

procedure TEventTest.ShowValue(const S: string);
begin
  Show(S);
end;

procedure TEventTest.UseMethod;
var
  StringEvent: TStringEventType;
begin
  StringEvent := ShowValue;
  StringEvent('Hello');
end;
```

위에 있는 간단한 코드만으로 이벤트의 유용함을 알기는 어렵다. 저-수준 메서드 포인터 타입 개념을 보다 잘 알 수 있도록 하는데 초점을 맞추고 있기 때문이다. 이벤트의 기술적 기반은 위와 같지만, 이것을 바탕으로, 어떤 오브젝트(예: 버튼) 안에 메서드 포인터를 넣어두고, 그것이 다른 오브젝트의 메서드(예: 그 버튼을 담고 있는 폼에 속한 OnClick 핸들러)를 참조하도록 하는 방식을 통해 우리는 더 많은 것을 할 수 있다. 많은 경우에, 이벤트는 프로퍼티를 이용하여 구현된다.

참고 혼하지는 않지만, 오브젝트 파스칼에서는 익명 메서드 [anonymous methods](#)를 사용해 이벤트 핸들러 [event handler](#)를 정의할 수도 있다. 이것이 희귀한 이유는 익명 메서드가 비교적 최근에 도입되었고, 그 시점에 기존의 라이브러리들이 이미 많이 있었기 때문이다. 거기에 더해 익명 메서드는 약간의 추가적인 복잡성을 제공한다. 이 접근법과 관련된 예제는 15장에서 보겠다. 다른 가능한 확장 [extension](#)으로는, C#에서 지원하는 것과 같이, 이벤트 하나에 여러 핸들러를 정의하는 것이다. 표준 기능은 아니지만 직접 구현할 수는 있다.

델리게이션이라는 개념 [The Concept of Delegation](#)

언뜻 보았을 때, 델리게이션의 목표는 명확하지 않겠지만, 이것은 오브젝트 파스칼의 컴포넌트 기술의 주춧돌이다. 그 비밀은 단어 '[위임](#)' [delegation](#)에 있다. 만일 누군가

오브젝트 안에 메서드 포인터를 넣어둔다면, 우리는 그 오브젝트의 동작 *behavior* 를 자유롭게 바꿀 수 있다. 그저 그 메서드 포인터에 새 메서드를 대입하기만 하면 된다. 뭔가 익숙한 것 같지 않은가? 당연히 익숙할 것이다.

우리가 버튼에 `OnClick` 이벤트 핸들러를 추가할 때, 개발 환경이 하는 일이 바로 그것이다. 버튼에는 메서드 포인터가 있고 그 이름이 `OnClick` 이다. 우리는 거기에 품이 가진 메서드를 (직접적이든 간접적이든) 대입할 수 있다. 사용자가 그 버튼을 클릭하면 우리가 대입해 놓은 메서드가 작동한다. 우리가 그 메서드를 다른 클래스 (보통 폼 클래스)에서 정의해도 작동한다.

다음 코드는 델파이 라이브러리 안에서 버튼 컴포넌트의 이벤트 핸들러를 실제로 정의하는데 사용되는 코드와 이와 연관된 폼의 코드의 개요다.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

  TMyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class(TForm)
    procedure Button1Click(Sender: TObject);
    Button1: TMyButton;
  end;

var
  Form1: TForm1;
```

그러면 이제, 프로시저 안에서 다음과 같이 작성하면 된다:

```
MyButton.OnClick := Form1.Button1Click;
```

위 코드 조각과 라이브러리 안에 있는 코드와 실제로 다른 점은 오직 하나다. 즉, `OnClick` 이 프로퍼티의 이름이고, 그 프로퍼티가 `FOnClick` 이라는 실제 데이터를 가리키고 있다는 점만 다르다. 사실, 오브젝트 인스펙터 안의 `Events` 페이지에 나열되는 이벤트들은 그 오브젝트의 프로퍼티 중 메서드 포인터인 것들일 뿐 별다른 게 아니다. 즉, 컴포넌트에 결합된 이벤트 핸들러를 개발자가 디자인 화면에서 동적으로 바꿀 수 있다. 심지어, 실행 중에 새 컴포넌트를 생성하고 그 컴포넌트에 이벤트 핸들러를 대입할 수도 있다.

다음 `DynamicEvents` 예제는 두 경우를 모두 보여준다. 그 폼에는 버튼이 하나 있고, 그 버튼에는 표준 `OnClick` 이벤트 핸들러가 결합되어 있다. 하지만 예제에서 나는 서명이 같은 (즉, 파라미터들이 같은) `public` 메서드를 하나 더 작성해 넣었다.

```
public
procedure BtnTest2Click(Sender: TObject);
```


그리고 아래 코드를 실행하면, 버튼을 눌렀을 때, 메시지를 보여주고 나서 바로 해당 이벤트 핸들러를 새로 만든 두번째 메서드로 바꾼다. 따라서, 앞으로 실행되는 그 클릭 명령은 처음과 다른 동작을 한다.

```
procedure TForm1.BtnTestClick(Sender: TObject);
begin
  ShowMessage('Test message');
  BtnTest.OnClick := BtnTest2Click;
end;

procedure TForm1.BtnTest2Click(Sender: TObject);
begin
  ShowMessage('Test message, again');
end;
```

이제, 버튼을 처음 클릭하면 첫 번째(기본) 이벤트 핸들러가 실행되지만, 그 이후의 모든 클릭은 두번째 이벤트 핸들러를 실행한다.

참고 이벤트에 메서드를 대입하는 코드를 타이핑하면, 사용 가능한 이벤트 이름들을 코드 완성 [Code Completion](#)이 제안해 줄 것이다. 그리고 이름을 선택되면 하고, 해당 함수 호출로 코드를 바꾼다 (끝에 중괄호를 붙인다). 그런데, 이것은 옳지 않다. 이벤트에는 메서드 그 자체를 대입하는 것이 옳다. 메서드를 호출하는 것은 옳지 않다. 만약 호출을 그대로 둔다면, 컴파일러는 (존재하지도 않는 프로시저임에도) 메서드를 호출하고 그 결과를 대신 대입하려고 할 것이므로, 오류를 일으킨다.

예제 프로젝트의 두 번째 부분에서는 완전히 동적인 이벤트 결합을 볼 수 있다. 폼의 표면을 클릭하면 새 버튼이 (이벤트 핸들러와 함께) 동적으로 생성된다. 그 이벤트 핸들러는 그 버튼의 텍스트를 보여준다(Sender 오브젝트는 동적으로 생성된 그 버튼이 된다. 그 버튼에 대한 이벤트 핸들러이기 때문이다):

```
procedure TForm1.BtnNewClick(Sender: TObject);
begin
  ShowMessage('You selected ' + (Sender as TButton).Text);
end;

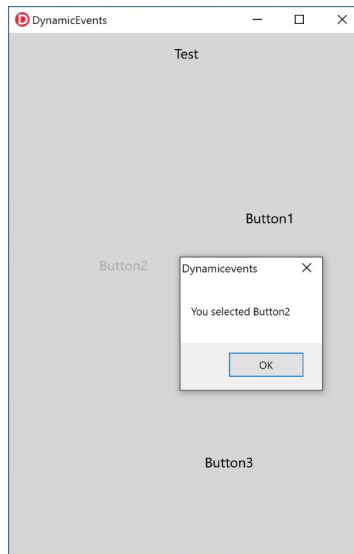
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
var
  AButton: TButton;
begin
  AButton := TButton.Create(Self);
  AButton.Parent := Self;
  AButton.SetBounds(X, Y, 100, 40);
  Inc(FCounter);
  AButton.Text := 'Button' + IntToStr(FCounter);
  AButton.OnClick := BtnNewClick;
end;
```

위 코드를 통해 동적으로 생성되는 버튼을 클릭하면, 생성된 그 버튼에 대한 메시지를 보여줄 것이다. 단 하나의 이벤트 핸들러를 (동적으로 생성되는) 모든

버튼들이 같이 사용하기는 하지만, 그 이벤트에는 `Sender` 파라미터가 있기 때문에 해당 오브젝트에 맞게 작동한다. 이 예제의 결과를 보여주는 예시는 아래 그림 10.1에 나와 있다.

그림 10.1:

동적으로 생성된 버튼에 의해
출력되는 메시지
(**DynamicEvents** 예제)



이벤트는 프로퍼티다 Events are Properties

여기서 중요한 개념은 오브젝트 파스칼 이벤트들은 거의 모두 (메서드 포인터 타입인) 프로퍼티로 구현된다는 것이다. 즉 컴포넌트의 이벤트를 다루려면, 여러분이 메서드를 해당 이벤트 프로퍼티에 대입하면 된다. 코드로 설명하자면, 이벤트 핸들러에게는 오브젝트의 메서드를 대입할 수 있다. 아래와 같이 하면 된다 (이미 앞에서 보았던 것이다).

```
| Button1.OnClick := ButtonClickHandler;
```

다시 말하지만, 이벤트라는 메서드 포인터 타입에 대입되려면, 메서드의 서명 [signature](#) 이 같아야 한다. 그렇지 않으면 컴파일러가 오류를 낸다.

델파이 시스템에서는 자주 사용하는 이벤트를 위한 표준 메서드 포인터 타입 여러 가지가 정의되어 있다. 그 중 간단한 것을 보면 다음과 같다.

```
| type  
    TNotifyEvent = procedure(Sender: TObject) of object;
```

이것은 `OnClick` 이벤트 핸들러에서 일반적으로 사용하는 타입이다. 따라서, (클래스 안에서) 해당 메서드 선언은 반드시 다음과 같아야 한다.

```
| procedure ButtonClickHandler(Sender: TObject);
```


이것이 조금 혼란스럽게 들렸다면, IDE 안에서 생기는 일들을 생각해보자: 예를 들어, IDE 에서 Button1 이란 버튼을 선택하고, 더블-클릭을 하면, 컨테이너 모듈(예: 그 버튼을 담고 있는 폼)에는 다음과 같은 새 빈 메서드가 추가된다:

```
procedure TForm1.Button1Click(Sender: TObject)
begin

  end;
```

여러분이 이 빈 메서드 안에 코드를 채기만 하면, 모두 잘 작동한다! 이벤트 핸들러를 이벤트에 대입하는 코드가 자동으로 생성되었기 때문이다. 여러분이 디자인 화면에서 프로퍼티들을 설정하면 해당 컴포넌트에 그것들이 반영되는 것과 같은 방식이다.

위 설명을 통해서, 이벤트와 거기 대입되는 메서드는 일대일 대응이 아니라는 것을 알 수 있었을 것이다. 오히려 그 반대에 가깝다. 이벤트 여러 개가 하나의 이벤트 핸들러를 공유할 수도 있다. 그래서 Sender 파라미터가 필요하고 또 자주 사용된다. Sender 는 그 이벤트를 발동시킨 오브젝트를 가리키기 때문이다. 예를 들어, 동일한 OnClick 이벤트 핸들러를 버튼 두 개에 붙여도, Sender 값에는 클릭을 한 그 버튼 오브젝트에 대한 참조가 들어간다.

참고 위 예시처럼, 코드를 사용하면 다른 이벤트들에게 같은 메서드를 대입할 수 있다. 하지만 디자인 화면에서도 그렇게 할 수 있다. 오브젝트 인스펙터에서 이벤트를 선택할 때, 이벤트 이름 옆의 화살표 버튼을 눌러 드롭다운 목록을 펼치면 “호환되는” 메서드들 — 즉, 그 메서드 포인터 타입에 일치하는 메서드들 — 이 나타난다. 따라서 컴포넌트가 달라도 이벤트가 같다면 거기에 같은 메서드를 쉽게 지정할 수 있다. 몇몇 경우에는, 같은 컴포넌트 안에 있는 이벤트 핸들러들 중에서 서로 호환되는 것들이 있다면, 그것들에게 동일한 이벤트 핸들러를 대입할 수도 있다.

TDate 클래스에 이벤트 추가하기 Adding an Event to the TDate Class

앞에서 TDate 클래스에 프로퍼티들을 추가했기 때문에 이제 이벤트를 추가할 수 있다. 매우 간단한 이벤트를 넣어보자. 이벤트의 이름은 OnChange 다. 용도는 날짜 값이 바뀌었음을 컴포넌트 사용자에게 알려주는 것이다. 이벤트 정의 방법은 간단하다. 그 이벤트 타입으로 프로퍼티를 하나 정의하고, 그 이벤트가 참조하는 실제 메서드 포인터를 값을 데이터를 추가하면 된다. 이 클래스에 새 정의를 추가하면 다음과 같다 (DateEvent 예제에서 발췌함):

```
type
  TDate = class
    private
      FOnChange: TNotifyEvent;
    protected
      procedure DoChange; dynamic;
    public
      property OnChange: TNotifyEvent read FOnChange write FOnChange;
    end;
```


이 프로퍼티 정의는 매우 간단하다. 이 클래스를 사용하는 개발자는 이 프로퍼티에 새 값을 대입할 수 있다. 따라서 비공개 `private` 인 `FOnChange` 필드에 새 값을 대입할 수 있다. 프로그램이 시작될 때는, 대체로 이 필드의 값이 대입되어 있지 않다. 이벤트 핸들러는 컴포넌트 작성자가 아니라 컴포넌트 사용자에게 필요한 것이기 때문이다. 컴포넌트 작성자 입장에서 추가하고 싶은 동작 `behavior` 이 있으면, 그 컴포넌트의 메서드에 추가하면 된다.

다르게 말하자면, `TDate` 클래스는 그저 전달받은 이벤트 핸들러를 받아서 `FOnChange` 필드에 담아둔다. 그리고, 날짜 값이 바뀔 때마다 `FOnChange` 필드에 담긴 메서드를 호출한다. 당연히, 그 이벤트 프로퍼티에 대입된 메서드가 있는 경우에만, 그 메서드를 호출한다.

`DoChange` 메서드는 이 필드에 핸들러 메서드가 대입되었는지 확인하고 나서 그 메서드를 실행한다 (참고로, 이벤트를 발동하는 메서드는 이것처럼 전통적으로 `dynamic` 으로 선언된다).

```
procedure TDate.DoChange;
begin
    if Assigned(FOnChange) then
        FOnChange(Self);
end;
```

참고 8장에서 소개했듯이, 동적 메서드 `dynamic method` 는 가상 메서드 `virtual method` 와 비슷하다. 하지만, 구현이 다르다. 구현 상, 호출이 메모리 사용량 `footprint` 이 적은 대신, 살짝 느리다는 단점이 있다.

`DoChange` 메서드는 값 중 하나가 변경될 때마다 호출된다. 코드는 다음과 같다:

```
procedure TDate.SetValue(Y, M, D: Integer);
begin
    FDate := EncodeDate(Y, M, D);
    // 이벤트를 발동한다
    DoChange;
end;
```

이제 이 클래스를 사용하는 프로그램을 들여다보자. 우리는 이 코드를 크게 단순화할 수 있다.

```
type
    TDateForm = class(TForm)
    ...
    procedure DateChange(Sender: TObject);
```

이 메서드의 코드는 `TDate` 오브젝트의 `Text` 프로퍼티를 현재 값으로 업데이트할 뿐이다.

```
procedure TDateForm.DateChange;
begin
    LabelDate.Text := TheDay.Text;
end
```


그리고, 이 이벤트 핸들러를 FormCreate 메서드 안에서 대입한다.

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
    TheDay := TDate.Init(7, 4, 1995);
    LabelDate.Text := TheDay.Text;
    // 이 이벤트 핸들러를 대입해 놓아서, 이후에 발생하는 변경을 반영하도록 한다.
    TheDay.OnChange := DateChange;
end;
```

글쎄, 상당히 작업이 많은 것 같다. 이벤트 핸들러가 코딩에 드는 수고를 줄여준다는 말은 거짓일까? 아니다. 이제 코드를 조금 추가하면, 이 오브젝트의 데이터 중 몇 가지가 변경되어도, 우리는 해당 레이블을 업데이트하는 것을 완전히 잊어도 된다. 예를 들어, 이 폼에 있는 버튼 하나의 OnClick 이벤트의 핸들러를 보자.

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
    TheDay.Increase;
end;
```

다른 많은 이벤트 핸들러 안에도 이처럼 간단한 코드가 있다. 일단 이벤트 핸들러를 장착 [install](#) 하고 나면, 레이블을 업데이트 업데이트하는 것을 계속 기억하지 않아도 된다. 그 결과, 프로그램에서 발생할 수 있는 잠재적 오류를 상당히 많이 줄여준다. 또한, 이 코드의 시작 부분에 코드 몇 줄을 작성해야 했던 이유는 TDate 가 단순히 클래스이기 때문이라는 점을 알아야 한다. 만약 TDate 가 컴포넌트이고 그 컴포넌트를 개발 환경에 설치했다면, 그저 오브젝트 인스펙터 안에서 해당 이벤트 핸들러를 선택하고, 그 안에 레이블을 업데이트하는 코드 한 줄만 넣으면 된다.

여기서 의문이 한 가지 든다. 델파이에서 새 컴포넌트를 작성하는 것은 얼마나 어려울까? 다음 소단원에서 그것이 얼마나 간단한지 보도록 하자.

참고 이 책은 사용자 지정 컴포넌트 [custom component](#) 작성에 대한 자세한 부분을 다루지 않는다. 하지만, 프로퍼티와 이벤트의 역할 그리고 컴포넌트 작성에 대해 짧게 소개한다. 왜냐하면, 이런 특징들 [features](#)에 대한 기본 이해는 모든 델파이 개발자들에게 중요하기 때문이다.

TDate 컴포넌트를 만들기 [Creating a TDate Component](#)

이제 우리는 프로퍼티와 이벤트를 이해했다. 그 다음 단계로 컴포넌트가 무엇인지 보자. 이 주제를 간략히 알아보기 위해, 우리의 TDate 클래스를 컴포넌트로 바꿔보자.

먼저, TDate 클래스가 TComponent 클래스를 상속받아야 한다. 기본값인 TObject 에서 TComponent 로 변경하자. 코드는 다음과 같다:

```
type
    TDate = class(TComponent)
```



```

...
public
  constructor Create(AOwner: TComponent); overload; override;
  constructor Create(Y, M, D: Integer); reintroduce; overload;

```

두 번째 단계는 아래와 같이, 기본 `default` 생성자를 오버라이드 `override` 하는 새 생성자를 추가하여 알맞은 초기 값을 제공할 수 있도록 한다. 또한, 오버로딩 된 `overloaded` 버전도 있기 때문에 `reintroduce` 지시어를 사용할 필요가 있다. 그래야 컴파일러가 경고 메시지를 출력하지 않는다. 새 생성자의 코드에서는 기반 클래스 `base class`의 생성자를 호출한 뒤에 날짜 값에 오늘 날짜를 지정한다.

```

constructor TDate.Create(AOwner: TComponent);
var
  Y, D, M: Word;
begin
  inherited Create(AOwner);
  FDate := Date; // 오늘 날짜

```

여기까지 했으면, 이제 우리의 클래스를 정의하고 있는 유닛(DateComp 예제에 있는 Dates 유닛)에 Register 프로시저를 추가해야 한다. (이 식별자는 반드시 대문자 R로 시작해야 한다. 그러지 않으면 인식되지 않으니 주의하자). 이 프로시저는 이 컴포넌트를 IDE에 추가하기 위해 필요하다.

Register 프로시저를 유닛의 interface 구역 안에 선언한다. 파라미터는 필요 없다. 그리고 나서, 아래 코드를 implementation 구역 안에 작성한다.

```

procedure Register;
begin
  RegisterComponents('Sample', [TDate]);
end;

```

이 코드는 개발 도구의 툴 팔레트 `Tools Palette` 안에 있는 *Sample* 페이지에 이 새 컴포넌트를 추가한다. 만일 *Sample* 페이지가 없을 경우, 이 페이지가 새로 생긴다.

마지막 단계는 컴포넌트를 설치 `install` 하는 작업이다. 그러려면, 패키지 `package`를 만들어야 한다. 패키지는 컴포넌트를 담는 특별한 애플리케이션 유형이다. 다음 순서대로 진행하면 된다:

- IDE의 메뉴에서 File > New > Other 순으로 선택한다. 그러면 New Items 대화 상자가 열린다.
- "Package"를 선택한다.
- 이름을 지정하고 그 패키지를 저장한다 (가능하면, 실제 컴포넌트 코드를 담고 있는 유닛과 같은 폴더 안에)
- 패키지 프로젝트가 새로 만들어지면, 프로젝트 매니저 `Project Manager` 창 안의 Contains 노드에서 마우스 오른쪽 클릭을 하고, Add New Unit을 선택한 후, TDate 컴포넌트 클래스를 담고 있는 유닛을 선택한다.

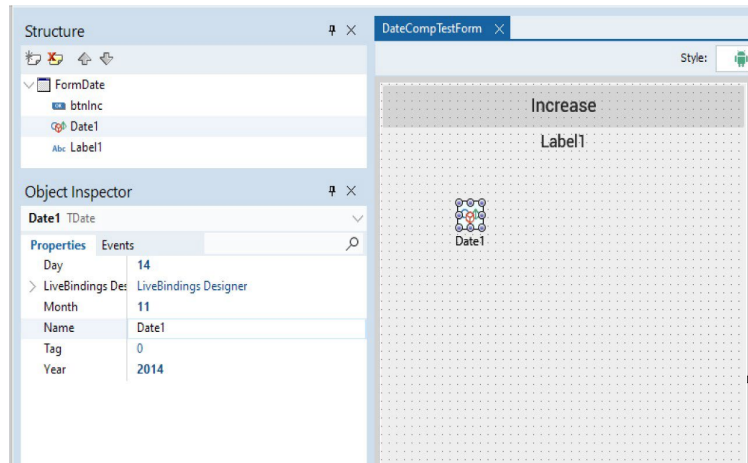
- 프로젝트 매니저 창 안의 이 패키지 노드에서 마우스 오른쪽 클릭을 하고 Build 명령을 선택한다. 빌드가 성공하면 다시 마우스 오른쪽을 클릭한 후, 이번에는 Install 메뉴 아이템을 선택하여, 개발 환경 안에 그 컴포넌트를 추가한다.

만일 이 책에 있는 코드를 가지고 한다면, 마지막 단계에 나와 있는 순서를 그대로 따르면 된다: DateComponent 폴더에서 DatePackage 프로젝트를 열고 컴파일 [Compile](#) 한 후 설치 [Install](#) 한다.

이제, 새 프로젝트를 만들고 툴 팔레트를 보면, *Sample* 아래에 새 컴포넌트가 들어 있을 것이다. 툴 팔레트의 검색 창에 새 컴포넌트의 이름을 타이핑하기 시작하면, 그 컴포넌트가 기본 아이콘과 함께 나타난다. 이제부터는, 그 컴포넌트를 폼 위에 올려 놓을 수 있고, 오브젝트 인스펙터에서 그 컴포넌트의 프로퍼티들을 조작할 수 있다. 그림 10.2 와 같다. 또한, OnChange 이벤트를 이전 예제에 비해 훨씬 쉬운 방법으로 다룰 수 있다.

그림 10.2

새 TDate 컴포넌트의
프로퍼티들을
오브젝트 인스펙터 안에서
볼 수 있다



이 컴포넌트를 사용하는 예제 애플리케이션을 직접 구축해보기를 강력히 권장한다. 그렇기는 하지만, DateComponent 예제를 열어 보아도 된다. 이 예제는 이 장의 뒷부분에서 차근차근 구축했던 그 컴포넌트를 업데이트한 버전이다. 기본적으로 DateEvent 예제를 보다 간단하게 만든 버전이다. 왜냐하면 이제는 이벤트 핸들러를 오브젝트 인스펙터 안에서 바로 사용할 수 있기 때문이다.

참고

만일 DateCompTest 예제를 열기 전에, 해당 컴포넌트(즉 DatePackage 예제) 컴파일하고 설치하는 단계를 진행하지 않았다면, IDE는 그 컴포넌트를 인식하지 못한다. 따라서 폼을 열다가 오류 메시지를 내보낼 것이다. 이 새 컴포넌트를 설치하기 전까지는, 이 예제 프로그램을 컴파일하지 못할 것이다. 즉 작업을 제대로 할 수 없을 것이다. IDE가 이런 상황에서 표시하는 오류 메시지 안에는 두 가지 선택이 제시된다. *Cancel*을 누르면 정의되지 않은 [undefined](#) 컴포넌트를 건너뛸 수 있고, *Ignore*를 누르면 코드에서 해당 컴포넌트를 제거할 수 있다.

클래스 안에 열거형 Enumeration 지원 구현하기

3장에서 전통적인 for 루프 대신 for-in 루프를 사용하는 방법을 보았다. 여기서는 어떻게 for-in 루프를 배열, 문자열, 세트, 기타 시스템 데이터 타입에 사용하는지 설명한다. 이런 루프를 어떠한 클래스든 적용하는 것이 가능하다. 그 클래스가 열거형 지원을 정의하면 된다. 가장 눈에 띄는 예는 요소들의 목록을 담고 있는 클래스다. 그런데, 기술 관점에서, 이 기능 *feature*은 거의 모든 것에 적용할 수 있다.

요소를 열거 *enumerating* 할 수 있게 클래스를 지원하려면, 여러분은 반드시 GetEnumerator 라는 메서드를 추가하고 그것이 클래스(열거를 실제로 수행하는 클래스)를 반환하도록 해야 한다. 또한 여러분은 그 (열거를 수행하는) 클래스를 정의하고 그 안에 MoveNext 메서드(요소 사이를 이동)와 Current 프로퍼티(실제 요소 하나를 반환)를 넣어야 한다. (방법은 잠시 후 실제 예제로 보자) 여기까지 하면, 컴파일러는 for-in 루프를 풀어낼 수 있다. 그 루프 안에서 대상은 우리의 클래스다. 그리고, 개별 요소의 타입은 그 열거자 *enumerator* 클래스의 Current 프로퍼티의 타입과 같아야 한다.

필수 요구 사항은 아니지만, 열거자를 지원하기 위한 클래스는 중첩된 타입 *nested type* (7장에서 다뤘다)으로 구현하는 것이 좋다. 왜냐하면 자기 자신을 열거하기 위한 용도인데 별개의 타입을 사용한다는 것은 말이 되지 않기 때문이다.

아래 클래스는 어느 범위의 숫자들을 (추상 컬렉션 형태로) 담고 있다. 또한, 그것들을 열거할 수 있다. 왜냐하면, 중첩된 타입으로 열거자가 정의되어 있고, GetEnumerator 함수가 그 열거자를 반환하기 때문이다 (NumbersEnumerator 예제에서 발췌함).

```
type
  TNumbersRange = class
  public
    type
      TNumbersRangeEnum = class
      private
        NPos: Integer;
        FRange: TNumbersRange;
      public
        constructor Create(ARange: TNumbersRange);
        function MoveNext: Boolean;
        function GetCurrent: Integer;
        property Current: Integer read GetCurrent;
      end;
    private
      FNStart: Integer;
      FNEnd: Integer;
    public
      function GetEnumerator: TNumbersRangeEnum;
      procedure Set_NEnd(const Value: Integer);
      procedure Set_NStart(const Value: Integer);

      property NStart: Integer read FNStart write Set_NStart;
      property NEnd: Integer read FNEnd write Set_NEnd;
    end;
```


위 GetEnumerator 메서드는 오브젝트를 하나 생성한다. 생성되는 오브젝트의 타입은 그 클래스 안에 중첩된 타입이다. 역할은 해당 데이터를 돌면서 하나씩 접근하기 위해 필요한 상태 정보를 저장하는 것이다.

이 열거자의 생성자 `constructor` 를 잘 보자. 열거할 실제 오브젝트(파라미터를 통해 전달되는 오브젝트, 여기에서는 `Self` 가 사용됨)에 대한 참조를 간직하는 방법과 그 시작 위치에 실제 오브젝트의 맨 앞을 지정하는 방법을 알 수 있다.

```
function TNumbersRange.GetEnumerator: TNumbersRangeEnum;
begin
    Result := TNumbersRangeEnum.Create(Self);
end;

constructor TNumbersRange.TNumbersRangeEnum.
    Create(ARange: TNumbersRange);
begin
    inherited Create;
    FRange := ARange;
    NPos := FRange.NStart - 1;
end;
```

참고 왜 생성자의 초기값에 첫번째 값을 그대로 쓰지 않고 거기서 1을 뺀까? `for-in` 루프를 위해 컴파일러가 생성하는 코드에서는 해당 열거자를 생성하고, *while MoveNext do*에서 *Current* 를 사용한다. 위 코드는 첫 번째 값을 얻기 전에 먼저 조건 검사가 수행된다. 목록 안에 값이 전혀 없을 수도 있기 때문이다. 즉 첫 번째 요소가 사용되기도 전에 먼저 *MoveNext*가 호출된다. 이것을 구현하기 위해 더 복잡한 논리를 사용할 수도 있겠지만, 여기에서는 간단히 시작 값을 '첫 값의 이전 값'으로 지정했다. 그러면 *MoveNext*가 처음 실행될 때 열거자의 위치는 첫번째 값에 놓이게 된다.

마지막으로, 이 열거자의 메서드들을 통해 우리는 데이터에 접근할 수 있고 그 목록의 다음 값(또는 그 범위 `range` 안에 있는 다음 요소)으로 이동할 수 있다:

```
function TNumbersRange.TNumbersRangeEnum.GetCurrent: Integer;
begin
    Result := NPos;
end;

function TNumbersRange.TNumbersRangeEnum.MoveNext: Boolean;
begin
    Inc(NPos);
    Result := NPos <= FRange.NEnd;
end;
```

위 코드에서 볼 수 있듯이 *MoveNext* 메서드는 두 가지 일을 한다. 즉, 목록 안에서 다음 요소로 이동하는 일과 열거자가 마지막 요소에 도달했는지를 점검하여 도달한 경우 `False`를 반환하는 일을 한다.

여기까지 모두 했다면, 이제 `for-in` 루프를 사용하여 이 범위 오브젝트의 값들을 돌면서 탐색할 수 있다.


```

var
  ARange: TNumbersRange;
  I: Integer;
begin
  ARange := TNumbersRange.Create;
  ARange.NStart := 10;
  ARange.NEnd := 23;

  for I in ARange do
    Show(IntToStr(I));
  end;
end;

```

아래 출력 결과는 10 부터 23 까지 열거된 값들을 담은 목록을 보여준다.

```

10
11
12
13
14
15
16
17
18
19
20
21
22
23

```

참고 RTL과 VCL 라이브러리들 안을 보면, 열거자를 정의하는 경우가 많다. 예를 들어, 각 TComponent는 자신이 소유^{own}하고 있는 컴포넌트들을 열거할 수 있다. 하지만, 자식 컨트롤들까지 열거하지는 못한다. 그것을 만드는 방법은 12장의 "클래스 헬퍼를 사용해 열거형 추가하기" 소단원에서 다룬다. 여기에서 그 예제를 보지 않는 이유는 클래스 헬퍼를 먼저 설명해야 하기 때문이다.

RAD와 OOP 섞어 쓰기와 관련된 15가지 팁

이 장에서는 프로퍼티, 이벤트, published 키워드를 다뤘다. 이것들은 빠른 애플리케이션 개발(RAD) 즉 시각적 개발 또는 이벤트 주도 프로그래밍 [Event-Driven Programming](#) 과 관련된 핵심 언어 특징들이다 (참고로, 이 세 용어가 가리키는 개념 모델은 동일하다). 매우 강력한 모델이지만, 그 뒷받침은 견고한 OOP 구조다. 가끔, 개발자들은 RAD 방식을 고려하느라 좋은 OOP 적용을 깜빡할 수 있다. 이와 동시에, 순수한 코드 작성을 고집하느라 RAD 방식을 무시하는 것은 비생산적이다. 이 마지막 소단원에서는 두 접근법 사이를 연결하는 것과 관련된 몇 가지 팁과 제안을 나열한다. 즉, 이 소단원의 주제를 “RAD 너머의 OOP” 이라고 이해해도 좋다.

참고 이 장의 마지막 소단원은 원래 "The Delphi Magazine" 17호(1999년 7월)에 "텔파이어에서, OOP를 위한 20가지 규칙^{20 Rules for OOP in Delphi}"이라는 제목으로 배포되었던 글이다. 여기에서는 규칙 중 몇 가지를 다듬고 일부 표현을 바꾸었다. 하지만, 핵심은 그대로이다.

팁 1: 폼은 클래스다 A form is a class

프로그래머들은 종종 폼을 오브젝트라고 생각하고 다루지만, 사실 폼은 클래스다. 둘의 차이는 클래스 하나를 바탕으로 폼 오브젝트가 여러 개 있을 수 있다는 점이다.

그런 혼란은 IDE 때문에 생긴다. IDE는 기본 글로벌 변수 `default global(전역)variable`를 생성한다. 그리고, 개발자가 프로젝트 안에서 정의한 모든 폼 클래스에 해당하는 폼 오브젝트를 생성한다 (이 작업은 기동할 때 프로젝트 설정에 맞추어 진행된다). 초보자인 경우에는 이 방식이 편리하다. 하지만, 사소하지 않은 애플리케이션인 경우에는 일반적으로 상당히 나쁜 버릇이다.

당연히, 폼(과 그 폼의 클래스)마다 또한 유닛마다 의미 있는 이름을 지정하는 것은 매우 중요하다. 폼과 유닛 간 이름이 (불행하게도) 다를 수 있지만, 그 둘이 짝을 서로 짝지어 주는 규약 convention 을 일관성 있게 사용하면 좋다 (예: `AboutForm` 폼과 `About.pas` 유닛).

다음 팁들을 본다면 "폼은 클래스다"라는 개념이 얼마나 유용한 효과가 있는지 보게 될 것이다.

팁 2: 컴포넌트에 이름을 붙여라 Name Components

설명하기 쉬운 이름을 컴포넌트에 붙이는 것 역시 중요하다. 폼 디자이너가 만들어 주는 기본 default 이름을 사용하지 마라. 가장 흔한 common 이름 짓기 방식은 첫 글자 몇 개는 클래스 타입을 나타내고, 그 뒤에는 그 컴포넌트의 역할을 적는 방식이다. 예를 들면, `BtnAddCustomer` 나 `EditName` 처럼 말이다. 이 스타일을 따르는 비슷비슷한 이름 형식들이 실제로 많이 있다. 그리고 그 중에서 무엇이 최고라고 말할 수는 없다. 개인의 취향이라고 보는 것이 맞다.

팁 3: 이벤트에 이름을 붙여라 Name Events

이벤트를 다루는 메서드에 적절한 이름을 붙이는 것도 중요하다. 컴포넌트에 적절한 이름을 붙였다면, `OnClick` 핸들러의 기본 이름은, 예를 들어, `BtnAddCustomerClick` 일 것이다. 물론 이 버튼 이름만으로 우리는 메서드가 하는 일을 추측할 수 있다. 하지만, 메서드가 발동되는 시점을 가리키는 이름보다는 메서드의 효과를 설명하는 이름을 사용하는 것이 더 좋다. 예를 들어, `BtnAddCustomer` 버튼의 `OnClick` 이벤트에는 `AddCustomerToList` 처럼 그 메서드가 하는 일이 무엇인지를 설명하는 이름을 붙인다.

이 방식은 코드를 더 명확히 읽을 수 있도록 해준다. 특히 그 이벤트 핸들러를 그 클래스에 있는 다른 메서드에서 호출할 때 그렇다. 또한 이 방식을 사용하면, 같은 메서드를 여러 이벤트 또는 다른 컴포넌트에 붙일 때도 도움이 된다; 물론, 사소하지 않은 애플리케이션이라면, 단일 이벤트를 여러 사용자 인터페이스 요소에 붙일 때는 액션 Actions 을 것이 더 좋다고 말하고 싶다.

참고 Action과 ActionList 컴포넌트는 VCL과 파이어몽키^{FMX} UI 라이브러리 모두에 있으며, 매우 멋진 구조적 기능이다. 이것들은 사용자의 동작^{action}(또한 그 상태^{status})을 개념적으로 따로 빼내어, 거기에 연결되는 사용자 인터페이스 컨트롤들과 분리할 수 있도록 한다. 연결된 컨트롤을 활성화하면 해당 액션이 실행된다. 실제로, 해당 액션을 논리적으로 비활성화하면, 그 액션이 연결된 UI 요소 또한 비활성화된다. 이 주제는 이 책의 범위를 벗어나지만, VCL이나 FMX 라이브러리를 사용하는 개발자라면 살펴볼 가치가 있다.

팁 4: 폼 메서드를 사용하라^{Use Form Methods}

폼이 클래스라면 그 코드는 메서드들 안에 모여 있어야 한다. 이벤트 핸들러를 제외하고(이것은 특별한 역할을 한다. 하지만 다른 메서드가 호출할 수도 있다), 개발자가 직접 만든 메서드 ^{custom method}를 폼 클래스에 추가하는 것이 종종 유용하다. 개발자는 폼에 연관된 일반적인 액션과 동작을 수행하는 메서드, 폼에 있는 컴포넌트들의 상태를 접근하는 메서드 등을 추가할 수 있다. 폼에 있는 컴포넌트들에 대한 접근을 위해서는 공개 ^{public} 메서드 또는 공개 ^{public} 프로퍼티를 추가하는 것이 시스템의 다른 부분에서 직접 접근할 수 있도록 하는 것보다 훨씬 더 좋다.

팁 5: 폼 생성자를 추가하라^{Add Form Constructors}

실행 중에 생성 ^{create} 되는 부수적인 폼은 (TComponent 클래스에서 상속받은) 기본 생성자 외에 구체적인 다른 생성자를 쓸 수 있다. 특정한 초기화가 필요한 경우에는, 기본 Create 메서드를 오버로딩 ^{overloading} 해서 필요한 초기화 파라미터를 추가하는 것을 권한다. 아래 코드 조각처럼 하면 된다.

```
public
    constructor Create(const AText: string); reintroduce; overload;

constructor TFormDialog.Create(const AText: string);
begin
    inherited Create(Application);
    Edit1.Text := AText;
end;
```

팁 6: 글로벌^{global/전역} 변수는 가급적 사용을 피하라^{Avoid Global Variables}

글로벌 ^{global/전역} 변수 ^{variable} (즉, 유닛의 interface 구역에 선언되는 변수)는 가급적 사용을 피해야 한다. 이것들로 인해 코드를 유지보수하기 어려워질 뿐만 아니라, 피할 수 있는 버그들이 생기는 경우가 종종 있기 때문이다.

폼에 추가 데이터를 저장할 곳이 필요하다면, 그 데이터를 위한 비공개 ^{private} 필드를 추가하라. 이 경우에는 그 폼의 각 인스턴스 ^{instance} 마다 그 데이터의 복사본을 가지게 된다.

유닛 변수(즉, 유닛의 `implementation` 구역에 선언되는 변수)를 사용할 수도 있다. 이 경우에는 그 품의 여러 인스턴스들이 그 데이터를 공유한다. 하지만, 이럴 때는 클래스 데이터를 사용하는 것이 훨씬 더 좋다(12 장에서 설명한다).

팁 7: 인스턴스 변수 `Instance Variable`를 그 구현 `Implementation` 안에서 사용하지 말라

그 오브젝트의 클래스는 메서드 안에서는 특정 오브젝트를 절대로 참조하면 안 된다. 예를 들어, `TMyForm` 클래스의 메서드 안에서 (자동 생성된) `MyForm` 변수를 참조하지 마라. 현재의 오브젝트를 참조하고 싶다면, `Self` 식별자를 사용하라. 그 클래스의 현재 오브젝트를 참조할 필요가 있는 경우가 거의 없다는 점도 명심하자. 그 클래스 코드에서는 현재 오브젝트의 메서드와 데이터를 직접 참조할 수 있기 때문이다. 이 규칙을 따르지 않으면 그 클래스의 인스턴스를 여러 개 만들 때 진짜 곤경에 처할 수 있다(아마 품에서 그럴 수 있다).

팁 8: 품의 변수는 가급적 쓰지 말라 `Seldom Use a Form's Variable`

심지어 다른 클래스의 코드 안이라 할지라도, 글로벌 `global/전역` 오브젝트를 직접 참조하지 않도록 노력하라(예: 다른 품 클래스일지라도 코드 안에서 `MyForm` 등 글로벌 오브젝트를 직접 참조하지 않는 것이 좋다). 훨씬 더 좋은 방식은 로컬 `local/지역` 변수 또는 비공개 `private` 필드를 선언하여 (`MyForm` 등) 다른 품들을 참조하는 것이다. 예를 들어, 프로그램의 메인 품 `main form` 안에 비공개 `private` 필드를 하나 두고, 그 필드가 대화 상자 품을 참조하도록 한다. 이 규칙은 이 부가적인 품의 인스턴스를 여러 개 만들 생각이라면, 이 규칙은 분명히 필수이다. 개발자는 메인 품 안에 동적 배열 `dynamic array`을 하나 만들어서 그 안에 목록을 보관할 수 있다. 또는 간편하게 글로벌 오브젝트인 `Screen`에 `Forms` 배열을 사용하면 그 애플리케이션 안에 현재 존재하고 있는 어떤 품이든 참조할 수 있다.

팁 9: 글로벌 `global/전역` 변수인 `Form1`을 제거하라 `Remove the Global Form1 Variable`

개발자가 IDE에서 새 품을 추가하면 프로젝트 안에 자동으로 글로벌 `global/전역` 품 오브젝트(예: `Form1`)가 생긴다. 이번 제안은 이런 품 오브젝트들을 제거하라는 것이다. 그렇게 하려면, 그 품을 자동으로 만드는 기능을 비활성화 해야 한다. 어쨌든 이런 것을 없애면 좋다. 아니면 프로젝트 소스에서 그 품을 생성하는데 쓰이는 코드 줄을 지워도 된다.

당연하게도, 품이 자동 생성되지 않는다면, 애플리케이션 안에서 품을 생성하는 코드와 그 품을 참조하는 변수를 개발자가 직접 넣어주어야 한다. 이 글로벌 `global/전역` 품 오브젝트를 제거하는 것은 오브젝트 파스칼 초보자에게 매우 유용하다. 그러면 클래스와 글로벌 오브젝트를 더 이상 혼동하지 않게 될 것이다. 실제로, 그 글로벌 오브젝트가 삭제된 후에는, 그것을 참조하는 코드는 모두 오류가 발생한다.

팁 10: 폼 프로퍼티를 추가하라 Add Form Properties

이미 이 장의 "폼에 프로퍼티 추가하기" 소단원에서 언급한 내용이다. 폼을 위한 데이터가 필요하다면, 비공개 `private` 필드를 추가하라. 만일 그 데이터를 다른 클래스에서 접근해야 한다면, 프로퍼티를 그 폼에 추가하라. 이 방식을 사용하면, 그 폼과 그 데이터의 코드(또는 폼의 사용자 인터페이스까지)를 변경해도, 다른 폼이나 다른 클래스의 코드를 바꾸지 않아도 된다. 부가적인 폼 또는 대화 상자를 초기화하거나 그것의 최종 상태를 읽을 때에도 프로퍼티 또는 메서드를 사용하는 것이 좋다. 초기화는 생성자 `constructor` 를 사용해서 수행할 수도 있다. 이점은 이미 설명했었다.

팁 11: 컴포넌트 프로퍼티를 노출하라 Expose Component Properties

다른 폼의 상태에 접근해야 할 때, 개발자는 그 폼의 컴포넌트들을 직접 참조해서는 안 된다. 그러면, 다른 폼들 또는 다른 클래스들의 코드가 해당 사용자 인터페이스에 같이 묶이게 된다. 그런데, 사용자 인터페이스는 애플리케이션에서 가장 많이 변경되는 부분 중 하나이기도 하다. 뿐만 아니라, 이런 컴포넌트들에 대한 관리를 둘러싸고 있는 로직 `logic/논리`의 캡슐화를 비켜가는 결과를 낳는다. 그러지 말고, 폼 프로퍼티를 하나 선언하고 이것을 해당 컴포넌트 프로퍼티에 매핑하라: 프로퍼티에서 `Get` 메서드를 통해 그 컴포넌트의 상태를 읽고, `Set` 메서드를 통해 상태를 쓰면 된다. 지금 사용자 인터페이스를 변경한다고 가정해보자. 개발자는 오직 그 프로퍼티에 연결된 `Get` 메서드와 `Set` 메서드만 고치면 된다. 그 컴포넌트를 참조지도 모를 모든 폼과 클래스의 소스 코드를 확인하고 고칠 필요가 없다.

팁 12: 필요하다면 배열 프로퍼티를 쓰라 Use Array Properties when Needed

폼 안에서 일련의 값들을 다루어야 한다면, 배열 프로퍼티 `array property` 하나를 선언하라. 그 폼에 중요한 정보일 경우에는, 그 배열 프로퍼티가 그 폼에서 기본 `default` 배열 프로퍼티가 되도록 해라. 그러면, 그 값을 직접 접근할 때, 코드에서 `self[3]`이라고 적으면 된다. 만약 다른 폼에서 접근할 때는, `FDataForm[3]`와 같이 적으면 된다. 이미 "배열 프로퍼티 사용하기" 소단원에서 다룬 것들인데, 거기에서는 폼에 국한하지 않고 보다 일반적인 경우를 설명했었다.

팁 13: 프로퍼티의 시작 동작 Starting Operations in Properties

글로벌 `global/전역` 데이터에 접근하는 대신 프로퍼티를 사용함으로써 얻는 이득 중 하나는 메서드를 호출하도록 할 수 있기 때문에 프로퍼티의 값을 쓸 때 (또는 읽을 때) 어떤 동작이든 수행할 수 있다고 설명했던 것을 기억하는가? 예를 들어, 폼의 표면에 직접 그리기, 여러 프로퍼티들에 값을 지정하기, 특별한 메서드를 호출하기, 여러 가지 컴포넌트들의 상태를 변경하기를 할 수 있다. 필요하다면, 이벤트를 발동할 수도 있다.

관련된 예제를 하나 더 보자. 프로퍼티 게터 [getter](#) 를 사용하면 지연된 생성 [delayed creation](#) 을 구현할 수 있다. 클래스 생성자 안에서 하위 오브젝트 [sub-object](#) 를 생성하는 대신, 필요한 첫 시점에 그 하위 오브젝트를 생성할 수 있다. 코드는 다음과 비슷하다.

```
private
  FBitmap: TBitmap;
public
  property Bitmap: TBitmap read GetBitmap;

function TBitmap.GetBitmap: TBitmap;
begin
  if not Assigned(FBitmap) then
    FBitmap := ... // 생성한다. 그리고 초기화 한다
  Result := FBitmap;
end;
```

팁 14: 컴포넌트를 숨겨라 [Hide Components](#)

순수 OOP 를 추종하는 사람으로부터 내가 너무 자주 듣는 불평이 있다. 폼의 [published](#) 구역 안에서 컴포넌트들의 목록이 들어있는데, 그 방식은 캡슐화라는 원칙을 위반한다는 불평이다. 사실 그들은 중요한 문제를 지적하고 있는 것이다. 하지만, 그들 중 대부분은 그 해결책이 간단하고, 라이브러리를 다시 작성하거나 언어를 변경할 필요 없다는 것을 알지 못한다. 폼에 추가된 컴포넌트 참조들을 폼 선언의 [private](#) 구역으로 옮기면 된다. 그러면 다른 폼에서는 그 컴포넌트 참조에 접근하지 못한다. 이 방식은 컴포넌트들에 매핑된 프로퍼티를 사용해야만 해당 컴포넌트의 상태에 접근할 수 있도록 강제할 수 있다 (앞의 소단원을 참고할 것).

만일 IDE 가 모든 컴포넌트를 [published](#) 구역에 넣었다면, 그 이유는 그 필드들을 스트리밍 파일(DFM 또는 FMX)로부터 생성되는 컴포넌트들에 묶는 [binding](#) 방식 때문이다. 개발자가 컴포넌트의 이름을 설정하면, VCL 은 그 컴포넌트 오브젝트를 폼 안에 있는 해당 참조에게 부착한다. 이 동작은 그 참조가 [published](#) 상태일 때만 가능하다. 왜냐하면, 이 스트리밍 [streaming](#) 시스템은 전통적인 RTTI 그리고 TObject 의 몇 가지 메서드를 사용해서 이 동작을 수행하기 때문이다.

그러므로, 컴포넌트 참조를 [published](#) 구역에서 [private](#) 구역으로 옮기면 우리는 이와 같은 자동화된 동작을 잃게 된다. 이 문제를 고치기 위해서는, 수동으로 동작하도록 만들면 된다. 그러려면 폼의 OnCreate 이벤트 핸들러 안에서 각 컴포넌트마다 아래 코드를 수작업으로 추가하면 된다.

```
Edit1 := FindComponent( 'Edit1' ) as TEdit;
```

그 다음 여러분이 할 작업은 시스템에 컴포넌트 클래스를 등록 [register](#) 하는 것이다. 그러면 이 컴포넌트들에 대한 RTTI 정보는 컴파일 된 프로그램 안에 들어가고 시스템에서 사용할 수 있게 된다. 이 수작업은 각 컴포넌트 클래스마다 단 한 번만 하면 된다. 그것도 이 타입 [type](#) 의 모든 컴포넌트 참조를 [private](#) 구역으로 옮겼을 때만

하면 된다. 이 등록 작업이 내 애플리케이션에 필요한지 잘 몰라도 그냥 추가할 수 있다. 이미 등록된 클래스에 대해서는 `RegisterClasses` 메서드를 추가 호출해도 아무 효과가 없기 때문이다. `RegisterClasses` 호출은 폼을 담고 있는 유닛의 `initialization` 구역 안에 추가하는 것이 일반적이다:

```
| RegisterClasses([TEdit]);
```

팁 15: OOP 폼 마법사를 사용하라 Use an OOP Form Wizard

모든 폼의 모든 컴포넌트에 대해 위 두 작업을 반복하는 것은 지루하고 시간을 잡아먹는 일이다. 이 과도한 부담을 피할 수 있도록 간단한 마법사를 만들어 놓았다. 이것은 프로그램에 추가해야 하는 몇 줄의 코드를 만들어서 작은 창 안에 넣어 준다. 여러분은 그저 각 폼마다 이 코드를 복사해서 붙여넣기만 하면 된다. 이 마법사는 유닛의 `initialization` 구역에 소스 코드를 자동으로 붙여주는 것까지는 해주지 않기 때문이다.

해당 마법사는 어디에서 받을 수 있을까? 아래 링크에 있는 "Cantools Wizard" 안에 들어있다:

```
| https://github.com/marcocantu/cantools
```

팁 마무리

이것은 그저 더 균형 있는 RAD 및 OOP 개발 모델을 위한 팁과 제안들을 담은 작은 모음집에 불과하다. 물론, 이 주제에 대해서는 이야기할 것이 더 많지만 그건 이 책의 논점을 벗어난다. 이 책은 언어 자체에 관한 것이지 애플리케이션 아키텍처에 대한 최고의 해법들 [best practices](#)에 관한 것이 아니다.

참고 델파이로 애플리케이션 아키텍처를 다룬 책들이 몇 권 있다. 하지만, Nick Hodges가 쓴 "Coding in Delphi", "More Coding in Delphi", "Dependency Injection in Delphi"만큼 좋은 책 시리즈는 없다. 이 책들에 관한 정보를 <http://www.codingindelphi.com/> 에 있다.

11: 인터페이스 Interfaces

C++ 등 몇몇 다른 언어와 달리, 오브젝트 파스칼의 상속 모델은 다중 상속 [multiple inheritance](#) 을 지원하지 않는다. 즉 각 클래스는 오직 하나의 기반 클래스만 가진다.

다중 상속의 유용성은 OOP 전문가들 사이에 논쟁이 되는 주제다. 오브젝트 파스칼에 이 구조가 없다는 사실은 단점으로 여기지기도 한다. 여러분이 C++의 파워를 가질 수 없기 때문이다. 하지만, 장점이기도 하다. 언어가 더 단순하다. 또한, 다중 상속으로 인한 여러 문제들을 피할 수 있다. 다중 상속이 없다는 점을 오브젝트 파스칼에서 보완하는 방법이 있다. 인터페이스를 사용하면 된다. 인터페이스를 사용하면 클래스 하나가 여러 추상화 [abstractions](#) 들을 구현하도록 정의할 수 있다.

참고 많은 현재의 객체 지향 프로그래밍 언어들은 다중 상속을 지원하지 않는다. 대신 인터페이스를 사용한다. C#과 Java 등이 그렇다. 인터페이스는 유연성을 제공한다. 그리고 클래스 하나에서 여러 인터페이스들을 구현하는 선언이 가능하다. 진정한 다중 상속 지원은 C++ 언어에 한해 남아있다고 보는 것이 일반적이다. 몇몇 동적 [dynamic](#) 객체 지향 언어들은 믹스인 [mix-in](#) 을 지원한다. 이는 보다 단순하고 다중 상속과 비슷한 효과를 다른 방식을 통해 실현한다.

다중 상속에 관해 논쟁하기보다는, 그저 하나의 오브젝트를 여러 “관점 [perspective](#) 들”로 다룰 수 있으면 유용하다 점만 생각하자. 그런데, 예제를 통해 파악하기에 앞서, 오브젝트 파스칼 언어에서 인터페이스가 하는 역할과 그 작동 방식을 먼저 소개한다.

좀 더 일반적인 관점에서 보면, 인터페이스가 지원하는 객체 지향 프로그래밍 모델은 클래스가 지원하는 모델과 조금 다르다. 인터페이스를 구현 [implementing](#) 하는 오브젝트라면 자신이 지원하는 각 인터페이스에 따라 메서드 다형성 [polymorphism](#) 이 적용된다. 그 오브젝트가 구현한 인터페이스들 메서드 [interface method](#) 라면 무엇이든 호출할 수 있기 때문이다. 클래스와 비교하자면, 인터페이스는 캡슐화를 수용한다. 하지만, (구현까지 상속되는 것이 아니므로) 클래스 간의 연결은 클래스 상속보다 느슨하다.

참고 이 장에서 다루는 기술과 인터페이스에 대한 지원 전반이 오브젝트 파스칼에 최초로 도입된 것은 윈도우 Windows COM (Component Object Model) 구조 architecture를 지원하고 구현하기 위해서였다. 그 이후에, 그 기능이 확장되어 COM이 아닌 상황에서도 쓸 수 있게 되었다. 그러나, 오브젝트 파스칼 인터페이스 구현에는 지금도 여전히 COM 요소가 남아 있다. 예를 들면, ID를 통한 인터페이스 식별 interface identity, 참조 카운팅 reference counting 지원 등이다. 이런 요소들은 오브젝트 파스칼이 다른 언어들과 조금 다른 점이다.

인터페이스 사용하기 Using Interfaces

추상 클래스 abstract class (추상 메서드 abstract method 를 가진 클래스)를 선언할 수 있는 것은 물론이고, ‘순수한 pure 추상 클래스’ 즉 가상인 virtual 추상 메서드들만 가지는 클래스도 오브젝트 파스칼로 작성할 수 있다. interface 키워드를 사용하여 데이터 타입의 묶음 sets 를 정의하고 그것을 인터페이스로 참조하면 된다.

기술적으로, 인터페이스는 클래스가 아니다. 비록 비슷해 보이기는 해도 말이다. 클래스는 인스턴스 instance 를 가질 수 있지만 인터페이스는 그렇지 않다. 인터페이스를 실제로 구현하는 것은 (하나 또는 그 이상의) 클래스들이다. 그 클래스들의 인스턴스가 결국 그 인터페이스를 ‘지원’ 내지는 ‘구현’하는 것이다.

오브젝트 파스칼에서, 인터페이스는 몇 가지 눈에 띄는 특징들이 있다:

- 인터페이스 타입 변수는 참조 카운팅이 된다 (클래스 타입 변수와 다른 점이다). 그래서 자동 메모리 관리가 작동한다.
- 클래스는 오직 하나의 기반 base 클래스로부터 상속을 받을 수 있다. 하지만 그와 동시에 여러 개의 인터페이스들을 구현할 수 있다.
- 모든 클래스는 TObject로부터 상속을 받는다. 모든 인터페이스는 IInterface로부터 파생 descend 된다. 그리고 독립적 separate이며 직교 orthogonal 하는 구조를 이룬다.
- 인터페이스 이름은 통상 대문자 I로 시작한다. (다른 데이터 타입 대부분이 대문자 T로 시작한다는 점과 다른 점이다)

참고 원래 오브젝트 파스칼에서는 기반 인터페이스 타입의 이름이 IUnknown이었다. COM의 요구사항이기 때문이었다. 이 IUnknown 인터페이스는 나중에 Iinterface로 이름이 바뀌었다. COM이 다루는 영역이 아니거나 COM이 존재하지 않는 운영체제에서도 쓸 수 있다는 것을 강조하기 위해서였다. 어쨌든 Iinterface의 실제 동작은 이전의 IUnknown과 똑같다.

인터페이스 선언하기 Declaring an Interface

핵심 개념을 봤으니 이제 오브젝트 파스칼에서 실제 코드를 보자. 인터페이스가 어떻게 작동하는지 이해할 수 있을 것이다. 실 사용 면에서, 인터페이스 정의는 클래스의 정의와 닮았다. 인터페이스 정의 안에는 메서드의 목록이 있다. 하지만 이 메서드들은 구현이 전혀 되어 있지 않다. 이것이 클래스 안에 있는 추상 메서드와 다른 점이다.

다음 코드는 인터페이스의 정의다:

```
type
    ICanFly = interface
        function Fly: string;
    end;
```

각 인터페이스가 직접 혹은 간접적으로 기반 인터페이스 타입을 지정해 상속받으려면, 다음과 같이 쓸 수 있다:

```
type
    ICanFly = interface(IInterface)
        function Fly: string;
    end;
```

잠시 후에, IInterface에서 상속을 받는다는 의미와 그 상속으로 인한 결과를 보여줄 것이다. 지금은 IInterface 안에는 기본 메서드 [base method](#)들이 있다는 점만 알고 넘어가자 (다시 말하지만, TObject와 비슷하다).

인터페이스의 선언에 대해 마지막으로 알아 둘 것이 있다. 인터페이스의 경우, 타입 검사 부분이 동적으로 수행된다. 따라서, 각 인터페이스는 저마다 고유한 [unique](#) 식별자 즉 GUID가 필요하다. GUID는 델파이에서 Ctrl+Shift+G를 누르면 자동으로 생성된다. (참고로, 이 단축키는 코드 안에서 GUID를 정의해야 하는 어느 곳에서도 쓸 수 있다)

다음은 완전한 인터페이스 코드의 예시다:

```
type
    ICanFly = interface
        ['{D7233EF2-B2DA-444A-9B49-09657417ADB7}']
        function Fly: string;
    end;
```

이 인터페이스에 대한 구현은 아래에서 보여준다. (모두 Intf101 예제에서 발췌함)

참고 여러분은 GUID를 명시하지 않아도 인터페이스를 컴파일하고 사용할 수 있다. 그러나 GUID를 만드는 것을 대체로 선호하게 될 것이다. 인터페이스 찾기 [querying](#), 인터페이스 타입에 대한 동적 as 타입캐스트 [typecast](#) 등은 GUID가 있어야 할 수 있기 때문이다. 인터페이스의 핵심은 런타임에 크게 확장되는 타입 유연성을 활용하는 것이다. 이는 인터페이스에 GUID가 있는가에 달려있다.

인터페이스 구현하기 [Implementing an Interface](#)

모든 클래스는 인터페이스를 하나 혹은 그 이상 구현할 수 있다. 자신의 기반 클래스 뒤에 그것들을 나열하면 된다. 그리고 나열된 각 인터페이스의 각 메서드들에 대한 구현을 제공하면 된다:

```
type
    TAirplane = class(..., ICanFly)
        function Fly: string;
    end;
```



```
function TAirplane.Fly: string;
begin
    // 실제 코드
end;
```

클래스가 인터페이스를 구현할 때에는 반드시 모든 인터페이스 메서드들에 대한 구현을 제공해야 한다. 이때 메서드 서명까지 일치해야 한다. 위 경우, TAirplane 클래스의 Fly 메서드는 반드시 문자열 `string`을 반환하도록 구현되어야 한다. ICanFly 인터페이스는 기반 인터페이스(IInterface)로부터 상속받는다. 따라서, 그 인터페이스를 구현하는 클래스(TAirplane)는 반드시 그 인터페이스(ICanFly)의 모든 메서드들과 그 기반 인터페이스(IInterface)의 모든 메서드들을 하나도 빠짐없이 구현해 제공해야 한다.

이런 이유로, 인터페이스를 구현하려는 클래스들은 대체로 이미 IInterface 기반 인터페이스의 메서드들을 구현해 놓은 클래스를 상속받는다. 오브젝트 파스칼의 런타임 라이브러리 안에는 그 기본적인 동작들을 구현해 놓은 기반 클래스들이 몇 개 있다. 가장 단순한 것은 TInterfacedObject 클래스다. 그래서, 위 코드는 아래와 같이 된다:

```
type
    TAirplane = class(TInterfacedObject, ICanFly)
        function Fly: string;
    end;
```

참고 인터페이스를 구현할 때 우리는 정적 `static` 메서드나 가상 `virtual` 메서드를 사용할 수 있다. 만일 상속을 받은 클래스에서 오버라이딩 `overriding`할 것이라고 계획한다면, 그 메서드에는 가상 메서드를 사용하는 것이 이치에 맞다. 다른 대안이 있긴 하다. 자신의 기반 클래스 역시 똑같은 인터페이스를 상속하도록 하고 나서, 그 기반 클래스에서 구현한 그 인터페이스 메서드를 오버라이드 `override`하도록 명시하는 방법이다. 하지만, (필요할 때) 인터페이스 메서드를 가상 메서드로 구현하도록 선언하는 방식을 더 좋아한다.

인터페이스와 그것을 구현하는 클래스까지 모두 정의했으니, 이제 이 클래스를 사용하는 오브젝트를 만들자. 우리는 이 클래스를 평범한 클래스처럼 다룰 수 있다:

```
var
    Airplane1: TAirplane;
begin
    Airplane1 := TAirplane.Create;
    try
        Airplane1.Fly;
    finally
        Airplane1.Free;
    end;
end;
```

위 코드는, 이 클래스가 인터페이스를 구현하고 있다는 점을 무시하고 있다. 이제는 이와 달리 아래와 같이 쓸 수 있다. 즉, 인터페이스 타입으로 변수를 선언할 수 있다. 인터페이스 타입 변수를 사용하면 참조 메모리 모델 `reference memory model`이 자동으로 활성화된다. 따라서 우리는 try-finally 구문을 생략해도 된다.


```

var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;

```

위 코드 맨 위의 한 줄은 단순해 보이지만, 몇 가지 고려할 점이 있다(Intf101 예제의 일부임). 첫째, 오브젝트가 인터페이스 변수에 할당되면, 런타임은 그 오브젝트가 그 인터페이스를 구현하고 있는지 자동으로 점검한다. as 연산자의 특별 버전이 사용된다. 이 동작은 사실 여러분이 직접 명시할 수도 있다. 코드를 아래와 같이 작성하면 된다:

```

Flyer1 := TAirplane.Create as ICanFly;

```

둘째, 직접 대입 코드이든 as 문이 명시된 코드이든 관계없이, 런타임은 그 오브젝트의 _AddRef 메서드를 호출한다. 이것은 참조 카운트를 하나 증가시킨다. 이 메서드를 호출할 수 있는 이유는 우리 오브젝트가 TInterfacedObject를 기반 클래스로 상속받았기 때문이다.

동시에, Flyer1이 실행 범위를 벗어나면 (즉 end 문장이 수행될 때), 델파이 런타임은 _Release 메서드를 호출한다. 이는 참조 카운트를 하나 감소시킨다. 만약 참조 카운트가 0이 된다면 그 오브젝트를 소멸시킨다. 그러므로, 위 코드에는 오브젝트를 수작업으로 해제할 필요가 없다. 물론 try-finally 블록도 필요 없다.

참고 위 소스코드에 try-finally 블록이 없지만, 컴파일러가 try-finally 블록과 그 안에서 _Release 구문을 암묵적으로 그리고 자동으로 추가한다. 오브젝트 파스칼에서 이런 동작이 발생하는 경우는 많다: 기본적으로 메서드가 하나 또는 그 이상의 매니지드 타입([managed type](#) (문자열, 인터페이스, 동적 배열 등)을 가지고 있으면, 컴파일러는 try-finally 블록을 자동으로 암묵적으로 자동 추가한다.

인터페이스와 참조 카운팅 Interfaces and Reference Counting

위 코드에서 봤듯이, 인터페이스 변수에 의해 참조되는 오브젝트 파스칼의 오브젝트는 참조 카운팅이 된다(단, 그 인터페이스 타입 변수에 weak 또는 unsafe 표시가 있는 경우는 제외: 뒤에서 설명한다). 그리고 그 오브젝트를 참조하는 인터페이스 변수가 전혀 없게 될 때, 메모리에서 자동으로 제거([deallocated](#))된다.

중요한 점이 있다. 비록 이와 관련된 몇 가지 컴파일러 마법(숨겨진 _AddRef와 _Release 호출)이 있긴 해도, 사실 참조 카운팅 메커니즘은 개발자 또는 런타임 라이브러리가 제공하는 특정 구현을 따른다. 직전 예시에서, 참조 카운팅 작동은 사실 TInterfacedObject 클래스의 메서드에 있는 코드 때문이다 (여기 간략한 버전이 있다).

```

function TInterfacedObject._AddRef: Integer;
begin
  Result := AtomicIncrement(FRefCount);
end;

```



```

function TInterfacedObject._Release: Integer;
begin
    Result := AtomicDecrement(FRefCount);
    if Result = 0 then
        begin
            Destroy;
        end;
    end;

```

이제 역시 RTL에 있고 IInterface를 구현하는 기반 클래스지만 위와 다른 것을 보자. TNoRefCountObject다. 이 클래스는 참조 카운팅 메커니즘을 기본으로 비활성화한다:

```

function TNoRefCountObject._AddRef: Integer;
begin
    Result := -1;
end;

function TNoRefCountObject._Release: Integer;
begin
    Result := -1;
end;

```

참고 새 TNoRefCountObject 클래스는 참조 카운팅 메커니즘을 무시하는 오브젝트들을 정의한다. 델파이 11에서 도입되었으며, 이름이 이상한 TSingletonImplementation 클래스를 대체한다 (Generics.Defaults 유닛 안에 정의되어 있었다). 옛 TSingletonImplementation은 여전히 새 TNoRefCountObject의 별칭 [alias](#)으로 남아 있다. 두 클래스의 코드는 근본적으로 동일하다. 이렇게 이름을 바꾼 이유는 원래의 클래스 이름이 싱글톤 패턴 [singleton pattern](#)과 전혀 관련이 없음에도 잘못 지어졌기 때문이다. 우리는 싱글톤 패턴 예제를 다음 장에서 볼 것이다.

TNoRefCountObject가 자주 쓰이지는 않으니 자체적인 메모리 관리 모델이 없어 인터페이스를 구현하면서 참조 카운팅 메커니즘은 비활성화하는 다른 클래스를 소개해줬고 그것이 바로 TComponent 클래스다.

만일 인터페이스를 구현하는 사용자 지정 컴포넌트 [custom component](#)를 만들고 싶다면 참조 카운팅과 메모리 관리에 대해 신경 쓸 필요 없다. 우리는 인터페이스를 구현하는 사용자 지정 컴포넌트 예제를 "인터페이스로 패턴 구현하기" 소단원에서 볼 것이다.

참조 혼용으로 인한 오류 [Errors in Mixing References](#)

오브젝트를 사용할 때, 그 오브젝트에 대한 접근은 오직 오브젝트 변수들만 사용하거나 아니면 오직 인터페이스 변수들만 사용해야 한다. 두 방식을 혼용하면, 오브젝트 파스칼의 참조 카운팅 구조가 망가진다. 그래서 메모리 오류가 발생한다. 이런 오류는 추적하기가 매우 어렵다. 실전에서, 인터페이스를 사용하기로 결정했다면 오로지 [exclusive](#) 인터페이스 기반 [interface-based](#) 변수들만 사용해야 한다.

많은 가능한 상황들 중 하나를 보자. 인터페이스가 있고, 그 인터페이스를 구현하는 클래스가 있고, 전역 프로시저 하나가 그 인터페이스를 파라미터로 받는다고 보자:


```

type
  IMyInterface = interface
    [ '{F7BEADFD-ED10-4048-BB0C-5B232CF3F272}' ]
    procedure Show;
  end;

  TMyIntfObject = class(TInterfacedObject, IMyInterface)
  public
    procedure Show;
  end;

procedure ShowThat(AnIntf: IMyInterface);
begin
  AnIntf.Show;
end;

```

코드는 상당히 뻘하다. 그리고 100% 옳다. 잘못될 수 있는 부분은 여러분이 이 프로시저를 호출하는 방식이다 (IntfError 예제에서 발췌함):

```

procedure TForm1.BtnMixClick(Sender: TObject);
var
  AnObj: TMyIntfObject;
begin
  AnObj := TMyIntfObject.Create;
  try
    ShowThat(AnObj);
  finally
    AnObj.Free;
  end;
end;

```

위 코드를 보자. 인터페이스를 기대하는 함수에게 평범한 오브젝트를 전달하고 있다. 그 인터페이스를 지원하는 오브젝트를 전달했기 때문에, 컴파일러가 이 호출을 만들 때는 문제가 없다. 문제는 메모리 관리 방식에 있다.

맨 처음에, 이 오브젝트의 참조 카운트는 0이다. 어떤 인터페이스도 그것을 참조하지 않기 때문이다. ShowThat 프로시저에 진입하면 참조 카운트는 1이 된다. 그건 괜찮다. 그리고 그 호출된 프로시저가 진행된다. 이제 그 프로시저가 끝나면 그 오브젝트의 참조 카운트는 1 감소하여 0이 된다. 그래서 그 오브젝트는 소멸된다. 즉 AnObj를 ShowThat 프로시저에게 전달했더니 AnObj가 소멸되어 버리는 상당히 이상한 상황이 된다. 위 코드를 실제로 실행하면 메모리 오류와 함께 중단^{fail}된다.

몇 가지 해결책이 있다. 참조 카운트를 여러분이 강제로 올리거나 다른 저-수준^{low-level} 기교를 쓸 수 있다. 하지만 올바른 해결책은 인터페이스 참조와 오브젝트의 참조를 서로 섞지 않는 것이다. 예를 들어, 오직 인터페이스들로만 위 오브젝트를 참조한다 (아래 코드 역시 IntfError 예제에서 발췌함):

```

procedure TForm1.BtnIntfOnlyClick(Sender: TObject);
var
  AnIntf: IMyInterface;
begin
  AnIntf := TMyIntfObject.Create;
  ShowThat(AnIntf);
end;

```


위 예시의 경우는 쉽게 해결했다. 하지만, 다른 많은 환경에서는 올바른 코드로 정정하기가 매우 어렵다. 다시 강조하지만, 핵심 원칙은 서로 타입이 다른 참조들을 섞어 쓰지 않는 것이다. 하지만, 다음 소단원에서는 좀 다른 최근 대안들을 소개한다.

약한^{Weak} 인터페이스 참조와 안전하지 않은^{Unsafe} 인터페이스 참조

델파이 10.1 베를린부터 오브젝트 파스칼은 인터페이스 참조의 관리에 대한 몇 가지 개선점을 제공한다. 오브젝트 파스칼은 이제 여러 다른 종류의 참조들을 제공한다.

- 일반적인^{regular} 참조: 할당되거나 할당 해제될 때 오브젝트 참조 카운트를 늘리거나 줄인다. 오브젝트 참조 카운트가 0에 도달하면 그 오브젝트를 메모리에서 해제한다.
- ([weak] 제어자^{modifier}로 표시되는) 약한 참조^{weak reference}: 이 참조가 가리키는 오브젝트의 참조 카운트를 늘리지 않는다. 이 참조는 자동으로 관리된다. 따라서 참조되는 오브젝트가 파괴된 경우에 이 참조는 자동으로 nil로 설정된다.
- ([unsafe] 제어자로 표시되는) 안전하지 않은 참조^{unsafe reference}: 오브젝트를 가리키는 참조 카운트를 늘리지도 않고 자동으로 관리되지도 않는다 - 일반적인 포인터와 별로 다르지 않다.

참고 weak 참조와 unsafe 참조가 맨 처음 도입된 것은 모바일 플랫폼에서 사용되던 ARC 메모리 관리를 지원하기 위해서였다. ARC가 단계적으로 폐지되고 있는 현재 델파이에서는 인터페이스 참조에서만 이 기능이 남아서 지원되고 있다.

참조 카운팅이 활성화되는 일반적인 경우로는 아래 코드를 꼽을 수 있다. 여기서는 참조 카운팅에 의존해서 임시 오브젝트^{temporary object}를 소멸^{dispose}시킨다:

```
procedure TForm3.Button2Click(Sender: TObject);
var
  OneIntf: ISimpleInterface;
begin
  OneIntf := TObjectOne.Create;
  OneIntf.DoSomething;
end;
```

만약 이 오브젝트가 표준 참조 카운트 구현을 가지고 있는데, 여러분은 그 총 참조 카운트에 포함되지 않는 인터페이스 참조를 생성하고 싶다면 어떻게 할까? 그러려면, 인터페이스 변수 선언에 [unsafe] 애트리뷰트^{attribute}를 붙이면 된다. 위 코드를 이렇게 변경한다:

```
procedure TForm3.Button2Click(Sender: TObject);
var
  [unsafe] OneIntf: ISimpleInterface;
begin
  OneIntf := TObjectOne.Create;
  OneIntf.DoSomething;
end;
```


사실 그리 좋은 생각이 아니다. 위 코드는 메모리 누수 [memory leak](#) 가능성이 있기 때문이다. 참조 카운팅을 끄면 이 변수가 해당 메모리 범위를 벗어나도 [out of scope](#) 아무 일이 생기지 않는다. 이것이 유익한 경우도 있다. 여러분이 여전히 인터페이스를 사용하지만, 참조를 추가로 더 만들지 않는 경우다. 다시 말해서, unsafe 참조는 포인터처럼 다루어진다. 컴파일러가 추가로 더 지원하는 게 전혀 없다.

unsafe 애트리뷰트를 사용해, 참조 카운트를 늘리지 않는 참조를 만들겠다는 생각을 하기 전에, 대부분의 상황이라면, 더 좋은 방법이 하나 더 있다는 점을 먼저 생각하자: 약한 참조 [weak reference](#)를 사용하는 것이다. 약한 참조 역시 참조 카운트를 늘리지 않는다. 하지만, 자동으로 관리되는 참조다. 시스템은 약한 참조를 추적하여 실제 오브젝트가 삭제될 경우 약한 참조를 nil로 설정한다. 안전하지 않은 참조를 사용할 경우 연결된 오브젝트의 상태를 알 수 없다(흔히 허상 참조 [dangling reference](#)라고 부르는 상황이다).

어떤 상황에 약한 참조가 유용할까? 고전적인 경우는 교차 참조 [cross-reference](#) 상태인 두 오브젝트다. 이런 경우, 한 오브젝트는 다른 오브젝트의 참조 카운트를 부풀린다 [inflate](#). 그래서 둘 다 근본적으로 메모리 위에 영원히 존재한다 (참조 카운트가 1로 설정됨). 심지어 더 이상 도달할 수 없게 되어도 그렇다.

예를 보자. 아래에는 자신과 타입이 같은 다른 인터페이스에 대한 참조를 받아들이는 인터페이스가 있다. 그리고 그 인터페이스에 대한 참조를 내부에 담는 클래스가 있다.

```
type
  ISimpleInterface = interface
    procedure DoSomething;
    procedure AddObjectRef(Simple: ISimpleInterface);
  end;

  TObjectOne = class(TInterfacedObject, ISimpleInterface)
  private
    AnotherObj: ISimpleInterface;
  public
    procedure DoSomething;
    procedure AddObjectRef(Simple: ISimpleInterface);
  end;
```

만일 두 개의 오브젝트를 만들고 서로 교차 참조를 하면 메모리 누수가 발생한다:

```
var
  One, Two: ISimpleInterface;
begin
  One := TObjectOne.Create;
  Two := TObjectOne.Create;
  One.AddObjectRef(Two);
  Two.AddObjectRef(One);
```

델파이에서 이 문제를 해결하려면, 비공개 [private](#) 필드인 AnotherObj를 약한 참조로 표시하면 된다:

```
private
  [weak] AnotherObj: ISimpleInterface;
```


위처럼 바꾸면, 여러분이 AddObjectRef를 호출하면서 넘긴 오브젝트의 참조 카운트는 변하지 않고 여전히 1이다. 그리고 그 변수가 자신의 범위를 벗어나면 그 오브젝트의 참조 카운트는 0이 된다. 따라서 그 오브젝트는 메모리에서 해제된다.

그 밖에도 약한 참조가 유용한 경우가 많다. 그리고 그 저변의 구현에는 정말 복잡한 것들이 있다. 정말 대단한 기능이다. 하지만 완전히 익히는 데에 적잖은 노력이 든다. 또한, 런타임 비용도 있다. 약한 참조는 관리되기 때문이다 (unsafe 참조는 아님).

인터페이스를 위한 약한 참조와 그 작동 방식에 대한 추가 정보는 13장 *"오브젝트와 메모리"*의 *"메모리 관리와 인터페이스"* 소단원에서 다룰 것이다.

고급 인터페이스 기술들 Advanced Interface Techniques

인터페이스의 능력에 더 깊이 들어가려면, 실제 상황을 다루기 전에, 그 기술의 고급 기능들을 더 다루는 것이 중요하다. 예를 들어, 여러 인터페이스들을 구현하는 법이나 이름이 다른 메서드로 인터페이스 메서드를 구현하는 법(이름 충돌의 경우) 등이다.

인터페이스의 또 다른 중요한 기능은 프로퍼티다. IntfDemo 예제는 인터페이스와 관련된 고급 기능들을 보여준다.

인터페이스 프로퍼티 Interface Properties

이 소단원의 코드는 서로 다른 2 가지 인터페이스를 기반으로 작성되었다. IWalker와 IJumper인데, 둘 다 메서드 몇 개와 프로퍼티 하나를 정의한다.

인터페이스 프로퍼티 [interface property](#)는 그저 read와 write 메서드에 매핑되는 이름일 뿐이다. 클래스의 경우와 다른 점이 있다. 인터페이스의 프로퍼티는 필드에 매핑할 수 없다. 인터페이스에 연결되는 코드가 없기 때문이다.

아래는 실제 인터페이스 정의들이다:

```
IWalker = interface
  ['{0876F200-AAD3-11D2-8551-CCA30C584521}']
  function Walk: string;
  function Run: string;
  procedure SetPos(Value: Integer);
  function GetPos: Integer;
  property Position: Integer read GetPos write SetPos;
end;

IJumper = interface
  ['{0876F201-AAD3-11D2-8551-CCA30C584521}']
  function Jump: string;
  function Walk: string;
  procedure SetPos(Value: Integer);
  function GetPos: Integer;
  property Position: Integer read GetPos write SetPos;
end;
```


프로퍼티 있는 인터페이스를 구현하려면, 접근하는 메서드를 구현하기만 하면 된다:

```
TRunner = class(TInterfacedObject, IWalker)
private
  FPos: Integer;
public
  function Walk: string;
  function Run: string;
  procedure SetPos(Value: Integer);
  function GetPos: Integer;
end;
```

메서드 코드 구현은 복잡하지 않다(IntfDemo 예제에서 발췌함). 새 위치를 계산하고 무엇이 실행되었는지 표시한다.

```
function TRunner.Run: string;
begin
  Inc(FPos, 2);
  Result := FPos.ToString + ': Run';
end;
```

IWalker 인터페이스와 그것을 구현한 TRunner를 사용하는 예시 코드다:

```
var
  Intf: IWalker;
begin
  Intf := TRunner.Create;
  Intf.Position := 0;
  Show(Intf.Walk);
  Show(Intf.Run);
  Show(Intf.Run);
end;
```

그 출력은 그리 놀랍지 않다:

```
1: Walk
3: Run
5: Run
```

인터페이스 델리게이션 Interface Delegation

비슷한 방법으로, IJumper 인터페이스를 구현하는 간단한 클래스를 정의할 수 있다:

```
TJumperImpl = class(TAggregatedObject, IJumper)
private
  FPos: Integer;
public
  function Jump: string;
  function Walk: string;
  procedure SetPos(Value: Integer);
  function GetPos: Integer;
end;
```

이 구현은 이전 코드와 다른 점이 있다. 기반 클래스로 TAggregatedObject를 사용한다. 인터페이스를 지원하기 위해 내부적으로 사용되는 오브젝트들을 정의하기 위해 만들어진 클래스다. 잠시 후, 그 구문 [syntax](#)을 보여주겠다.

참고 TAggregatedObject는 IInterface의 또다른 구현이며 System 유닛 안에 정의되어 있다. TInterfacedObject와 비교하자면, 참조 카운팅 구현(컨테이너^{container} 또는 컨트롤러^{controller}에게 모든 참조 카운팅을 기본적으로 위임함) 그리고 인터페이스 찾기 구현(컨테이너가 여러 인터페이스를 지원할 경우에 필요함)이 다르다.

이것을 조금 다른 방법으로 사용해보자. 아래의 TMyJumper 클래스에서는 IJumper 인터페이스(와 그 메서드들)에 대한 구현을 반복하지 않는다. 그 대신, 이미 그것을 구현한 클래스에게 그 인터페이스 구현을 위임^{delegate}한다. 이건 상속으로는 불가능하다(우리는 두 개의 기반 클래스를 가질 수 없기 때문이다). 대신 오브젝트 파스칼의 특정 기능인 ‘인터페이스 델리게이션^{interface delegation}’을 사용한다. 아래 클래스는 프로퍼티를 통해 구현 오브젝트를 참조함으로써 인터페이스를 구현한다. 따라서 그 인터페이스의 실제 메서드들 구현을 직접 하지 않아도 된다:

```
TMyJumper = class(TInterfacedObject, IJumper)
private
  FJumpImpl: TJumperImpl;
public
  constructor Create;
  destructor Destroy; override;
  property Jumper: TJumperImpl read FJumpImpl implements IJumper;
end;
```

위 선언을 보자. TMyJumper 클래스에서 IJumper 인터페이스를 구현하는 것은 FJumpImpl 필드다. 물론 이 필드는 그 인터페이스의 모든 메서드들을 실제로 구현하고 있어야 한다. 그러기 위해, TMyJumper 오브젝트가 생성될 때, FJumpImpl 필드에 들어갈 적절한 오브젝트를 생성해야 한다 (생성자에서 파라미터가 사용되고 있다. 이는 기반 클래스인 TAggregatedObject가 요구하는 것이다).

```
constructor TMyJumper.Create;
begin
  FJumpImpl := TJumperImpl.Create(Self);
end;
```

이 클래스에는 소멸자도 있다. 일반 필드로 참조(이 경우 참조 카운팅이 작동하지 않음)하고 있는 그 내부 오브젝트를 해제^{free}하기 위해서다.

이 예제는 간단하지만, 대체로 이 내부 오브젝트인 FJumpImpl에서 동작하는 메서드들을 여러분이 변경하거나 새로 추가하면 점점 복잡해진다. 요점은 여러분이 인터페이스 구현 하나를 여러 클래스에서 재사용할 수 있다는 것이다. 이처럼 인터페이스를 간접적으로 구현한 것이나 표준 방식으로 작성된 것이나 사용하는 코드는 동일하다:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Intf: IJumper;
begin
  Intf := TMyJumper.Create;
  Intf.Position := 0;
  Show(Intf.Walk);
  Show(Intf.Jump);
```



```
Show(Intf.Walk);
end;
```

다중 인터페이스와 메서드 별칭 Multiple Interfaces and Methods Aliases

매우 중요한 인터페이스의 특징이 더 있다. 클래스 하나가 여러 인터페이스를 구현할 수 있다. 아래 TAthlete 클래스는 IWalker와 IJumper 인터페이스를 둘 다 구현한다:

```
TAthlete = class(TInterfacedObject, IWalker, IJumper)
private
  FJumpImpl: TJumperImpl;
public
  constructor Create;
  destructor Destroy; override;
  function Run: string; virtual;
  function Walk1: string; virtual;
  function IWalker.Walk = Walk1;
  procedure SetPos(Value: Integer);
  function GetPos: Integer;

  property Jumper: TJumperImpl read FJumpImpl implements IJumper;
end;
```

한 인터페이스는 직접 구현이 되었다. 하지만, 다른 인터페이스는 내부의 FJumpImpl 오브젝트에게 위임되었다.

여기서 문제가 하나 있다. 우리가 구현하려는 이 두 인터페이스에는 모두 Walk라는 메서드가 있는데 그 서명도 똑같다. 그렇다면, 이 클래스에서 이 두 메서드를 어떻게 구현할까? 오브젝트 파스칼은 메서드 이름 충돌을 (여러 인터페이스 상황에서) 어떻게 처리할까? 그 해답은 그 메서드에게 다른 이름을 붙이는 것이다. 그리고 그것을 해당 인터페이스의 메서드에 매핑한다. 그 인터페이스 이름을 접두사로 붙이면 된다.

```
function IWalker.Walk = Walk1;
```

위 선언의 의미를 보자. 이 클래스는 IWalker 인터페이스의 Walk 메서드를 구현한다. 그런데 구현된 메서드의 이름은 Walk1이다(이름이 같은 메서드를 쓰지 않는다). 끝으로, 이 클래스의 모든 메서드들을 구현하다 보면, 내부 FJumpImpl 오브젝트의 Position 프로퍼티를 참조해야 한다.

Position 프로퍼티를 위한 새 구현을 우리가 선언했기 때문에, 우리는 한 운동 선수의 위치를 두 개나 가지는 좀 이상한 상황에 처한다. 여기 몇 가지 예제가 있다:

```
function TAthlete.GetPos: Integer;
begin
  Result := FJumpImpl.Position;
end;

function TAthlete.Run: string;
begin
  FJumpImpl.Position := FJumpImpl.Position + 2;
  Result := IntToStr(FJumpImpl.Position) + ': Run';
end;
```


TAthlete 오브젝트를 가리키는 인터페이스 하나를 생성해서 IWalker와 IJumper에 있는 동작들을 모두 참조할 수 있는 방법은 뭘까? 글썄, 사실 우리는 정확히 그렇게 할 수는 없다. 그것을 가능하게 할 하나의 기반 인터페이스 [base interface](#)가 없기 때문이다. 하지만, 인터페이스는 타입 검사 [type checking](#)와 타입 캐스팅 [type casting](#)을 훨씬 동적으로 할 수 있다. 따라서 우리는 한 인터페이스에서 다른 인터페이스로 변환할 수 있다. 우리가 참조하고 있는 오브젝트가 그 두 인터페이스를 모두 지원한다면 말이다. 이는 런타임이 되어야 컴파일러가 알아낼 수 있다. 이런 상황을 위한 코드는 다음과 같다:

```
procedure TForm1.Button3Click(Sender: TObject);
var
    Intf: IWalker;
begin
    Intf := TAthlete.Create;
    Intf.Position := 0;
    Show(Intf.Walk);
    Show(Intf.Run);
    Show((Intf as IJumper).Jump);
end;
```

물론, 우리는 두 인터페이스 중 하나를 선택하면 된다. 그리고 다른 것으로 변환하면 된다. `as` 캐스트 사용은 런타임 변환 [runtime conversion](#)을 하는 방법이다. 하지만, 여러분이 인터페이스를 다룰 때 선택할 수 있는 방법은 더 많다. 다음 소단원에서 보여 주겠다.

인터페이스의 다형성 [Interface Polymorphism](#)

앞에서 우리는, 여러분이 어떻게 여러 인터페이스들을 정의하고 어떻게 클래스 하나로 그 둘을 구현하는지를 보았다. 물론, 그 수는 얼마든지 더 확장할 수 있다. 여러분은 인터페이스들의 계층 [hierarchy](#)를 만들 수도 있다. 즉 한 인터페이스가 다른 인터페이스를 상속하도록 할 수 있다:

```
ITripleJumper = interface(IJumper)
    ['{0876F202-AAD3-11D2-8551-CCA30C584521}']
    function TripleJump: string;
end;
```

이 새 인터페이스 타입은 기반 인터페이스 타입의 모든 메서드(와 프로퍼티)들을 가진다. 그리고 새 메서드를 추가한다. 물론, 인터페이스의 호환성 [compatibility](#) 규칙들도 존재한다. 클래스를 위한 규칙과 동일하다.

이 소단원에서는 조금 다른 주제인 인터페이스-기반 다형성 [polymorphism](#)에 집중하기 바란다. 일반적인 기반 클래스 오브젝트가 있다면, 우리는 가상 메서드를 부를 수 있고 알맞은 버전이 호출된다고 보장할 수 있다. 인터페이스에서도 마찬가지다. 하지만 인터페이스에서는 한 걸음 더 나아가, 인터페이스를 찾는 동적인 코드를 종종 만든다. 오브젝트-인터페이스 관계는 충분히 복잡할 수 있다 (오브젝트 하나가 여러 인터페이스들을 구현한다. 또한 그들의 기반 인터페이스들도 간접적으로 구현한다). 따라서 이 상황에서 우리가 무엇을 할 수 있는지 더 좋은 그림을 파악하고 있는 것은 중요하다.

먼저, 여러분이 일반적인 `IInterface` 참조를 하나 가지고 있다고 가정하자. 그 참조가 특정 인터페이스를 지원하고 있는지 여부를 어떻게 알 수 있을까? 실제로 여러 가지 방법들이 있다. 무슨 클래스를 대하고 있는지에 따라 약간씩 다르다:

- `is` 문장을 사용해 테스트한다. (이어서 변환까지 하려면 `as` 캐스트 [cast](#)를 사용한다). 이것을 사용하면, 오브젝트가 특정 인터페이스를 지원하는지 알 수 있다. 하지만 인터페이스에 의해 참조되고 있는 오브젝트가 다른 인터페이스도 지원하는지 알 수는 없다(즉, 여러분은 `is`를 인터페이스들에게는 적용하지 못한다). 알아 둘 점이 있다. 어떤 경우이든 `as` 변환을 써야 한다. 인터페이스를 직접 캐스트 하면 거의 항상 오류를 내기 때문이다.
- 전역 함수인 `Support`를 호출한다. 오버로드된 버전이 많으므로 그 중 하나를 쓰면 된다. 이 함수에게 여러분이 테스트할 오브젝트(또는 인터페이스)와 타겟 인터페이스(`GUID` 또는 타입 이름)를 전달하면 된다. 덤으로 제공되는 결과까지도 원한다면 인터페이스 변수도 전달한다. 그러면 이 함수는 (호출이 성공하는 경우) 그 변수 안에 실제 인터페이스를 넣어서 돌려줄 것이다.
- 기반 인터페이스인 `IInterface`에 있는 `QueryInterface` 메서드를 직접 호출한다. 이것은 약간 저수준 [low-level](#) 방법이다. 이 경우, 덤으로 제공되는 결과를 받기 위한 인터페이스 타입 변수 하나를 무조건 전달해야 한다. 이 함수의 결과 값은 `HRESULT` 숫자 값이다. 불리언 [boolean](#) 값이 아니다.

마지막 두 기법을 일반적인 `IInterface` 변수에 적용하는 방법을 보여주는 코드 조각은 다음과 같다(역시 같은 `IntfDemo` 예제에서 발췌함):

```
procedure TForm1.Button4Click(Sender: TObject);
var
    Intf: IInterface;
    WalkIntf: IWalker;
begin
    Intf := TAthlete.Create;
    if Supports(Intf, IWalker, WalkIntf) then
        Show(WalkIntf.Walk);

    if Intf.QueryInterface(IWalker, WalkIntf) = S_OK then
        Show(WalkIntf.Walk);
end;
```

`QueryInterface`를 호출하는 것에 비해 `Supports` 함수의 오버로드된 버전들 중 하나를 사용하는 것을 권장한다. `Supports`는 궁극적으로 `QueryInterface`를 호출한다. 하지만, 사용하기에 더 간단하고 더 수준이 높다.

인터페이스의 다형성을 여러분이 사용하고 싶어질 또다른 경우로는, 더 상위에 있는 인터페이스 타입들이 담겨있는 인터페이스의 배열 [array](#)을 여러분이 다뤄야 하는 경우다 (오브젝트의 배열에서 사용할 수도 있다. 특정 인터페이스를 지원하는 오브젝트들이 담겨 있을 수 있으니까 말이다).

인터페이스 참조로부터 오브젝트 추출하기 Extracting Objects from Interface References

오브젝트 파스칼의 많은 예전 버전에서는, 여러분이 오브젝트를 인터페이스 변수에 대입하고 나면, 그 원래 오브젝트에 접근할 수 있는 방법이 없었다. 그래서 개발자들은 이따금 인터페이스 안에 `GetObject` 메서드를 추가했다. 그래서 그 동작을 구현하도록 유도했다. 하지만, 그것은 상당히 이상한 설계^{odd design}다.

지금의 오브젝트 파스칼에서는, 인터페이스 참조를 캐스트^{cast} 하여 그 안에 대입되었던 원래 오브젝트로 되돌릴 수 있다. 세 가지 단계별 방법이 있다:

- `is` 테스트를 작성한다. 그래서 주어진 타입의 오브젝트가 그 인터페이스 참조로부터 정말로 추출될 수 있는지 검증한다.

| `IntfVar is TmyObject`

- `as` 캐스트를 작성해 타입 캐스트를 수행한다. 오류인 경우에는, 예외가 발생한다.

| `IntfVar as TMyObject`

- 하드^{hard} 타입 캐스트를 작성해 변환한다. 오류인 경우에는, `nil` 포인터가 반환된다.

| `TMyObject(IntfVar)`

참고 모든 경우에 타입 캐스트 동작은 오브젝트 파스칼의 오브젝트로부터 가져온 인터페이스에서만 작동하며 COM 제공자^{server}에서는 작동하지 않는다. 또 원래의 오브젝트의 클래스로 정확하게 캐스트 할 뿐만 아니라 (타입 호환성 규칙에 따라) 그 기반 클래스로도 캐스트 할 수 있다.

예를 들어, 인터페이스와 그것을 구현하는 클래스를 보자. (`ObjFromIntf` 예제에서 발췌):

```
type
  ITestIntf = interface(IInterface)
    [ '{2A77A244-DC85-46BE-B98E-A9392EF2A7A7}' ]
    procedure DoSomething;
  end;

  TTestImpl = class(TInterfacedObject, ITestIntf)
  public
    procedure DoSomething;
    procedure DoSomethingElse; // 인터페이스 안에 없음
    destructor Destroy; override;
  end;
```

위 정의가 있으니, 이제 인터페이스 변수를 정의하고, 거기에 오브젝트를 대입하고, 그것을 사용해 그 인터페이스에 없는 메서드까지 부를 수 있다. 새 캐스트를 활용함:

```
var
  Intf: ITestIntf;
begin
  Intf := TTestImpl.Create;
  Intf.DoSomething;
  (Intf as TTestImpl).DoSomethingElse;
```


여러분은 코드를 아래와 같이 작성할 수도 있다. `is` 테스트와 직접 캐스트를 사용한다. 언젠든 오브젝트의 실제 클래스에 대한 기반 클래스로 캐스트가 가능하다:

```
var
  Intf: ITestIntf;
  Original: TObject;
begin
  Intf := TTestImpl.Create;
  Intf.DoSomething;
  if Intf is TObject then
    Original := TObject(Intf);
  (Original as TTestImpl).DoSomethingElse;
```

직접 캐스트는 성공하지 않으면 `nil`을 반환한다. 따라서 아래와 같이 작성할 수도 있다 (`is` 테스트를 뺐음):

```
Original := TObject(Intf);
if Assigned(Original) then
  (Original as TTestImpl).DoSomethingElse;
```

주의할 점이 있다. 인터페이스에서 추출한 오브젝트를 변수에 대입하면 참조 카운팅 문제에 노출될 수 있다: 그 인터페이스가 `nil`이 되거나 범위 밖으로 벗어나면 해당 오브젝트는 실제로 제거된다. 그리고 그 오브젝트를 참조하고 있는 변수는 유효하지 않게 된다. 그 문제를 잘 보여주는 코드는 이 예제의 `BtnRefCountIssueClick` 이벤트 핸들러에서 찾을 수 있다.

인터페이스로 어댑터 패턴 [Adapter Pattern](#)을 구현하기

인터페이스 활용의 현실 세계의 예시로, 어댑터 패턴 [adapter pattern](#)을 다뤄보겠다. 어댑터 패턴은 한 클래스의 인터페이스를 다른 인터페이스로 변환해서 그 클래스의 사용자의 기대에 맞추려고 할 때 사용된다. 이 패턴을 통해 여러분은 이미 존재하는 클래스를 (정의된 인터페이스를 맞추도록 요구하는) 프레임워크 안에 사용할 수 있다.

이 패턴을 구현하려면 클래스 계층 [hierarchy](#) 하나를 새로 만든다. 새 클래스 계층은 기존 클래스들을 매핑하거나 확장하여, 그 클래스들이 새 인터페이스를 노출하도록 한다. 이는 다중 상속 (언어가 지원하는 경우) 또는 인터페이스들을 사용하여 실현할 수 있다. 후자의 경우, 지금 설명하려는 방식이다, 상속을 받아 새 클래스를 만들고, 주어지는 인터페이스를 자신에게 구현한다. 그리고 그 메서드를 기존 동작 [behavior](#)에 매핑한다.

구체적인 상황을 보자. 이 어댑터 [adapter](#)는 여러 가지 컴포넌트들의 값을 질의하는 공통 인터페이스를 제공한다. 그런데, (UI 라이브러리에서 종종 그렇듯이) 인터페이스들은 컴포넌트에 따라 일관적이지 않을 수 있다. 이 인터페이스의 이름을 `ITextAndValue`라고 붙인 이유는 컴포넌트의 상태에 접근하여 텍스트로 된 설명이나 숫자 값을 가져오기 위한 것이기 때문이다.


```

type
  ITextAndValue = interface
    '[51018CF1-0D3C-488E-81B0-0470B09013EB]'
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;

    property Text: string read GetText write SetText;
    property Value: Integer read GetValue write SetValue;
end;

```

다음 단계는 이 인터페이스를 사용하고 싶은 컴포넌트들마다 각각 하위클래스 [subclass](#)를 새로 만든다. 예를 들어 다음과 같이 작성한다:

```

type
  TAdapterLabel = class(TLabel, ITextAndValue)
  protected
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;
end;

```

이 메서드 4개에 대한 구현은 꽤 간단하다. 왜냐하면, 그저 해당 Text 프로퍼티에 매핑하면 되기 때문이다. Text 프로퍼티는 그 값(또는 그 텍스트)이 숫자인 경우 타입 변환을 수행한다. 그렇기는 해도, 여러분은 이 새 컴포넌트를 설치해야만 한다(이전 장에서 설명한 바와 같다) 또한 여러분의 폼에 있던 기존 컴포넌트를 이 새 컴포넌트로 대체해야 한다. *어댑터를 적용*하려는 모든 컴포넌트마다 이 과정을 반복하려면 상당한 시간이 든다.

훨씬 더 간단한 대안은 *인터포저 클래스* [interposer class](#) 라는 것(클래스를 정의할 때 자신의 기반 클래스와 똑 같은 이름을 사용한다. 그런데, 이 클래스를 기반 클래스와는 다른 유닛 안에 넣는다)을 쓰는 방식이다. 컴파일러와 런타임 스트리밍 시스템 [runtime streaming system](#)은 이것을 올바르게 인식한다. 그러므로, 실행될 때 [run-time](#)는 결국 그 새 특정 클래스의 오브젝트를 얻게 된다. 유일한 차이점은 디자인할 때 [design-time](#)는 기반 컴포넌트 클래스의 인스턴스를 보게 되고 그것과 상호작용한다는 점뿐이다.

참고 인터포저 클래스가 처음으로 언급되고 그 이름으로 불리게 된 것은 오래 전 The Delphi Magazine에서였다. 약간 꼼수 [hack](#)에 가까우나, 상당히 편리한 개념이다. 인터포저 클래스 즉, 기반 클래스와 이름이 같은 클래스이면서, 기반 클래스가 담긴 유닛이 아닌 다른 유닛 안에 정의된 클래스에 대해 나는 오브젝트 파스칼의 일반적인 문장 [idiom](#)을 넘어서었다고 간주한다. 이 메커니즘이 효력을 가지려면, *uses* 문을 나열할 때 인터포저 클래스가 정의된 유닛은 그것이 대체하려는 일반 클래스가 정의된 유닛보다 뒤에 적어야 하는 점을 기억하자. 다시 말해, *uses*에서 맨 나중에 서술된 유닛 안에 있는 심볼 [symbol](#)은 그보다 앞에 서술된 유닛 안에 정의된 (이름이 같은) 심볼을 대체한다. 물론, 이 심볼들을 언제나 구분할 수 있도록 하려면 심볼들 앞에 접두사 [prefix](#)를 붙이면 된다. 하지만, 그렇게 하면 이 꼼수는 아예 차단된다. 이 꼼수는 전역 이름 해석 규칙들 [global name resolution rules](#)을 활용하기 때문이다.

인터포저 클래스를 정의하려면, 대체로 새 유닛을 작성하고, 이미 존재하는 기반 클래스와 똑 같은 이름을 가진 클래스를 그 새 유닛 안에 정의한다. 해당 기반 클래스를 참조할 때는, 유닛 이름을 클래스 앞에 접두사로 붙인다(그렇지 않으면 컴파일러는 그 문장을 재귀적 정의라고 간주한다).

```
type
  TLabel = class(StdCtrls.TLabel, ITextAndValue)
protected
  procedure SetText(const Value: string);
  procedure SetValue(const Value: Integer);
  function GetText: string;
  function GetValue: Integer;
end;
```

이렇게 하면, 여러분이 컴포넌트를 설치하거나 기존 프로그램을 건드릴 필요가 없다. 그저, `uses` 문에 나열된 목록의 맨 뒤에 추가 `uses` 대상을 하나 더 붙이면 된다. 두 경우 모두(데모 애플리케이션에서는 인터포저 클래스를 사용하고 있다), 폼 안에 있는 컴포넌트들에게 이 어댑터 인터페이스 통해 질의할 수 있다. 그리고 예를 들어, 그 컴포넌트들 모두의 `값`을 50으로 지정할 수 있다. 이는 컴포넌트가 다르고 프로퍼티가 달라도 반영된다:

```
var
  Intf: ITextAndValue;
  I: integer;
begin
  for I := 0 to ComponentCount - 1 do
    if Supports(Components[I], ITextAndValue, Intf) then
      Intf.Value := 50;
end;
```

그 데모 안에서, 위 코드는 진행상태-바 `progress bar` 또는 숫자 상자 `number box`의 값에 영향을 준다. 또한 라벨 `label` 또는 에디트-상자 `edit-box`의 텍스트에 영향을 준다. 게다가 위 코드는 내가 어댑터 인터페이스를 정의해 놓지 않은 다른 컴포넌트들은 완전히 무시한다. 이것이 매우 특수한 경우이긴 하나, 만일 다른 디자인 패턴 `design pattern`들을 살펴본다면, 그 패턴들 중 몇 가지는 오브젝트 파스칼에서 (자바, C# 등 인터페이스를 광범위하게 사용하는 유명한 다른 언어들처럼) 클래스보다는 인터페이스가 가진 추가적인 유연성의 이점을 활용해서 더 좋게 구현될 수 있다는 것을 쉽게 알 수 있을 것이다.

12: 클래스 조작하기 Manipulating Classes

이전의 몇몇 장에서 여러분은 오브젝트 파스칼의 오브젝트 측면의 기초들을 보았다. 클래스, 오브젝트, 메서드, 생성자, 상속, 나중에 바인딩하기, 인터페이스, 등등이다. 이제 한 걸음 더 나가자. 조금 더 수준이 높고, 클래스를 다루는 것과 관련된 고유한 언어 기능들을 살펴보겠다. 클래스 참조부터 클래스 헬퍼까지, 이 장에서 다루는 많은 특징들은 다른 OOP 언어에서 찾을 수 없거나, 적어도 상당히 다르게 구현되어 있다.

이 장의 초점은 클래스 그리고 실행 중(runtime)에 클래스 조작하기다. 런타임 조작은 뒤에 가서 리플렉션(reflection)과 애트리뷰트(attribute)까지 뻗어 나간다. 이것은 16장에서 다룬다.

클래스 메서드와 클래스 데이터 Class Methods and Class Data

오브젝트 파스칼에서 클래스를 정의할 때, 다른 OOP 언어 대부분도 그렇듯, 여러분은 그 클래스의 오브젝트(즉 인스턴스)들의 데이터 구조와 각 오브젝트가 수행하게 될 동작들을 정의한다. 그런데 더 있다. 그 클래스의 모든 오브젝트들이 공유하는 데이터와 그 클래스에서 (실제 오브젝트와 상관없이) 호출할 수 있는 메서드들도 정의할 수 있다.

오브젝트 파스칼에서 이런 클래스 메서드(class method)를 정의하려면 class 키워드를 메서드 앞에 붙이면 된다. 정확한 위치는 procedure 또는 function 키워드 앞이다.

```
type
  TMyClass = class
    class function MeanValue: Integer;
```

아래 MyObject 오브젝트의 클래스는 TMyClass다. 따라서, 여러분은 MeanValue 클래스 메서드 호출을 오브젝트에 적용할 수도 있고 클래스 자체에서 적용할 수도 있다.


```
var
  MyObject: TMyClass;
begin
  ...
  I := TMyClass.MeanValue;
  J := MyObject.MeanValue;
```

즉, 여러분은 인스턴스가 없어도 클래스 메서드를 호출할 수 있다. 경우에 따라, 클래스 메서드들만으로 된 클래스도 있을 수 있다. 여러분이 오브젝트를 생성하지 않을 거라는 가정이 깔려 있다면 말이다(강제하려면, Create 생성자를 비공개 [private](#)로 선언하면 된다).

참고 클래스 메서드들을 일반적으로 사용하고, 특히 클래스 메서드들만으로 된 클래스들을 사용하는 경우는 전역 함수 사용을 허용하지 않는 OOP 언어들에서 더욱 흔하게 볼 수 있다. 오브젝트 파스칼은 여전히 여러분이 "구식" 전역 함수 선언할 수 있도록 허용하고 있다. 그러나, 최근 몇 년 간 시스템 라이브러리들 그리고 개발자들이 작성하는 코드들은 클래스 메서드를 일관되게 사용하는 쪽으로 가고 있다. 클래스 메서드를 사용해서 얻는 이득은 클래스에 논리적으로 묶인다는 점이다. 이 경우 클래스는 관련 함수들을 묶는 네임 스페이스 [name space/이름공간](#) 역할을 한다.

클래스 데이터 [Class Data](#)

클래스 데이터는 그 클래스의 모든 오브젝트들이 공유하는 데이터다. 즉, 전역 저장소인데, 그 클래스로만 접근한다 (접근 제한 기능도 활용 가능). 클래스 데이터를 선언하는 방법은 그저 클래스 안에 구역을 `class var` 키워드 조합으로 정의하면 된다:

```
type
  TMyData = class
    private
      class var
        FCommonCount: Integer;
    public
      class function GetCommon: Integer;
```

`class var` 구역은 하나 또는 그 이상의 클래스 데이터 필드들이 선언되는 블록이다. 같은 구역(`private` 아래) 안에서, 여러분은 `var` 구역을 써서 일반 인스턴스 필드들을 선언할 수 있다.

```
type
  TMyData = class
    private
      class var
        FCommonCount: Integer;
      var
        MoreObjectData: string;
    public
      class function GetCommon: Integer;
```

여러분이 선언할 수 있는 것은 클래스 데이터만이 아니다. 클래스 프로퍼티도 정의할 수 있다. 잠시 후에 보게 될 것이다.

가상 클래스 메서드와 숨겨진 Self 파라미터 Virtual Class Methods and the Hidden Self Parameter

클래스 메서드는 여러 프로그래밍 언어에서 볼 수 있지만, 오브젝트 파스칼의 클래스 메서드 구현은 몇 가지 특징이 있다. 첫째, 클래스 메서드는 암시적인 (즉 숨겨진) `Self` 파라미터를 가진다. 이 숨겨진 `Self` 파라미터는 클래스 자체에 대한 참조다. 참고로, 인스턴스 메서드에 숨겨진 `Self` 파라미터는 그 클래스의 인스턴스에 대한 참조다.

클래스 메서드에 숨겨진 파라미터가 있고, 그것이 클래스 자신을 참조하는 것이 쓸모 없어 보일 수도 있다. 컴파일러는 그 메서드를 가진 클래스를 알 수 있기 때문이다. 하지만, 이 언어만의 특이한 기능이 있다: 다른 언어들과 달리, 오브젝트 파스칼에서 클래스 메서드는 가상-virtual일 수 있다. 여러분은 파생된 클래스에서 기반 타입의 클래스 메서드를 오버라이드-override/재정의할 수 있다. 평범한 메서드에서 하는 방법과 똑같다.

참고 가상 클래스 메서드 지원은 가상 생성자(특수한 목적의 클래스 메서드) 지원과 연결된다. 그런데, 컴파일 되고-compiled 타입이 엄격한-strong-typed 다른 언어들 안에는 이 두 언어 요소가 없다.

클래스의 정적 메서드 Class Static Methods

정적인 클래스 메서드가 오브젝트 파스칼에 도입된 것은 플랫폼 호환성을 위해서였다. 평범한 클래스 메서드와 정적인-static 클래스 메서드는 다른 점이 있다. 정적인 클래스 메서드에는 그 클래스 자체에 대한 참조가 없다 (즉, 그 클래스 자체를 가리키는 `Self` 파라미터가 없음) 또한, 정적 클래스 메서드는 가상 메서드일 수 없다.

아래 예문에는 틀린 문장 몇 개를 주석 처리해 놓았다. (ClassStatic 예제에서 발췌):

```
type
  TBase = class
    private
      FTmp: Integer;
    public
      class procedure One;
      class procedure Two; static;
    end;

  class procedure TBase.One;
  begin
    // 예러: 인스턴스 멤버인 'FTmp'는 여기(클래스 메서드 안)에서 접근하지 못함
    // Show(FTmp);
    Show('One');
    Show(Self.ClassName);
  end;

  class procedure TBase.Two;
  begin
    Show('Two');
    // 예러: Undeclared identifier: 'Self' (정적 클래스 메서드는 Self가 없음)
    // Show(Self.ClassName);
    Show(ClassName);
  end;
```


위 클래스 메서드 두 개는 클래스에서 직접 호출할 수 있다. 뿐만 아니라, 오브젝트를 통해서도 호출할 수 있다. 아래와 같다:

```
TBase.One;
TBase.Two;

Base := TBase.Create;
Base.One;
Base.Two;
```

정적 클래스 메서드가 오브젝트 파스칼에서 유용하게 쓰이는 이유는 두 가지 흥미로운 특징 덕분이다. 첫째, 클래스 프로퍼티를 정의하는데 쓸 수 있다. 잠시 후에 설명한다. 둘째, 정적 클래스 메서드는 C언어와 완전히 호환된다. 지금 바로 설명한다.

정적 클래스 메서드와 윈도우 API 콜백 Static Class Methods and Windows API Callbacks

정적 클래스 메서드에는 (숨겨진) `self` 파라미터가 없다. 따라서 정적 클래스 메서드는 (윈도우 등) 운영체제에게 콜백 함수로 넘길 수 있다. 실전에서, 여러분은 정적 클래스 메서드를 `stdcall` 호출 규약을 붙여서 선언하고, 그것을 윈도우 API 콜백으로 사용할 수 있다. 아래 `TimerCallback` 메서드가 그 예시다 (`StaticCallback` 예시에서 발췌):

```
type
  TFormCallback = class(TForm)
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
  private
    class var
      NTimerCount: Integer;
  public
    procedure AddToList (const AMessage: string);
    class procedure TimerCallback(hwnd: THandle;
      uMsg, idEvent, dwTime: Cardinal); static; stdcall;
  end;
```

위 클래스 데이터는 콜백이 사용해서 그 횟수를 담는다. `OnCreate` 핸들러는 `SetTimer` 윈도우 API를 호출한다. 이때 이 정적 클래스 프로시저의 주소를 넘겨준다.

```
procedure TFormCallback.FormCreate(Sender: TObject);
var
  Callback: TFNTimerProc;
begin
  NTimerCount := 0;
  Callback := TFNTimerProc(@TFormCallback.TimerCallback);
  SetTimer(Handle, TIMERID, 1000, Callback);
end;
```

참고 `TFNTimerProc`는 파라미터로 메서드 포인터를 전달받는다. 따라서 정적 클래스 메서드의 이름 앞에 `@` 기호를 붙이거나 또는 `Addr` 함수를 사용해서 전달해야 한다. 우리는 그 메서드를 실행하는 것이 아니라 그 메서드의 주소를 얻어서 전달해야 하기 때문이다.

이 콜백 함수가 실제로 하는 일을 보자. 횟수를 증가하고 폼을 업데이트한다. 그러기 위해 폼을 참조하는 글로벌 변수를 그대로 쓰고 있다(피해야 할 방식이다. 데모라서 간단히 하느라 그랬다). 정적 클래스 메서드는 `Self`를 불러 그 폼을 참조하지 못한다:

```
class procedure TFormCallBack.TimerCallBack(
  hwnd: THandle; uMsg, idEvent, dwTime: Cardinal);
begin
  try
    Inc(NTimerCount);
    FormCallBack.AddToList(
      IntToStr(NTimerCount) + ' at ' + TimeToStr(Now));
  except on E: Exception do
    Application.HandleException(nil);
  end;
end;
```

위에서 `try-except` 블록의 역할은 예외가 윈도우로 전달되는 것을 방지하는 것이다. 여러분은 콜백이나 DLL 함수에서 이 규칙을 일관적으로 따라야 한다.

클래스 프로퍼티 Class Properties

정적 클래스 메서드를 이유 중 하나는 클래스 프로퍼티를 구현하기 위해서다. 클래스 프로퍼티는 뭘까? 표준 프로퍼티와 마찬가지로, 읽기 쓰기 메커니즘에 붙이는 심볼 *symbol*이지만, 표준 프로퍼티와 다른 점이 있다. 이것은 클래스 자체에 연관된다. 또한 그 구현 안에서는 클래스 데이터 또는 정적 클래스 메서드를 써야 한다. (다시 `ClassStatic` 예제로 돌아가서) `Tbase` 클래스에는 클래스 프로퍼티 두 개가 있다. 그 구현은 각각 다르다. 하지만, 우리는 두 방식 모두를 사용할 수 있다.

```
type
  TBase = class
  private
    class var
      FMyName: string;
  public
    class function GetMyName: string; static;
    class procedure SetMyName(Value: string); static;
    class property MyName: string read GetMyName write SetMyName;
    class property DirectName: string read FMyName write FMyName;
  end;
```

인스턴스 카운터를 가지는 클래스 A Class with an Instance Counter

클래스 데이터와 클래스 메서드는 클래스 자체에 관한 정보를 담는 데 쓸 수 있다. 그런 종류의 정보에는, 그 클래스에서 생성된 인스턴스의 총 개수 그리고 그 중 현재 존재하고 있는 인스턴스의 개수도 해당될 것이다. `CounterObj` 예제를 통해 보자. 정말 쓸모없는 프로그램이다. 단지, 이 특수한 문제만 보여주기 위한 것이다. 하는 일은 그저 숫자 값 하나를 가지는 간단한 클래스의 오브젝트를 생성하는 것뿐이다.


```

type
  TCountedObj = class(TObject)
  private
    FValue: Integer;
  private
  class var
    FTotal: Integer;
    FCurrent: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    property Value: Integer read FValue write FValue;
  public
    class function GetTotal: Integer;
    class function GetCurrent: Integer;
end;

```

오브젝트가 생성될 때마다 프로그램은 FTotal과 FCurrent 두 카운터를 모두 늘린다. 오브젝트가 소멸할 경우, FCurrent 카운터를 하나 줄인다.

```

constructor TCountedObj.Create(AOwner: TComponent);
begin
  inherited Create;
  Inc(FTotal);
  Inc(FCurrent);
end;

destructor TCountedObj.Destroy;
begin
  Dec(FCurrent);
  inherited Destroy;
end;

```

클래스 정보는 특정 오브젝트를 참조하지 않아도 접근할 수 있다. 심지어 메모리에 오브젝트가 하나도 없어도 여전히 접근할 수 있다.

```

class function TCountedObj.GetTotal: Integer;
begin
  Result := FTotal;
end;

```

현재 상태를 보여주는 코드는 다음과 같다:

```

Label1.Text := TCountedObj.GetCurrent.ToString + '/' +
  TCountedObj.GetTotal.ToString;

```

이 예제에서, 라벨을 변경하는 코드는 타이머(timer)에 의해 실행된다. 따라서, 이 코드는 특정 오브젝트 인스턴스를 참조할 필요가 없다. 또한 수작업으로 직접 발동할 필요도 없다. 이 예제의 버튼은 그저 오브젝트 몇 개를 생성하고 소멸할 뿐이다. 오브젝트를 메모리에 그대로 남겨 두기도 한다 (사실 잠재적 메모리 누수가 있는 프로그램이다).

클래스 생성자(와 소멸자) Class Constructors (and Destructors)

클래스 생성자(class constructor)는 클래스와 관련된 데이터를 초기화(initialize) 하는 방법 중 하나다. 또한 클래스 초기화 담당자 initializer 역할을 맡는다. 실제로 뭔가를 생성하는 경우는 없기 때문이다. 클래스 생성자는 표준인 인스턴스 생성자와 아무 상관이 없다: 그저 클래스 자체를 (그 클래스가 사용되기 전에) 초기화하는 코드일 뿐이다. 예를 들어, 클래스 생성자에서 클래스 데이터의 초기값(initial value)을 지정하고, 구성(configuration) 또는 지원 파일들을 적재하는 등을 할 수 있다.

오브젝트 파스칼에서는 클래스 생성자가 유닛의 초기화(unit initialization) 코드가 대안이 될 수 있다. 그 두 개가 (한 유닛 안에) 함께 있을 경우, 클래스 생성자가 먼저 실행되고 그 다음에 유닛 초기화 코드가 작동한다. 정반대로, 여러분은 클래스 소멸자(class destructor)를 정의할 수 있는데, 그것은 종료화 코드(finalization code) 이후에 실행된다.

그런데, 중요한 차이가 있다. 유닛의 초기화 코드는 유닛이 컴파일 되면 반드시 실행된다. 반면 클래스 생성자와 클래스 소멸자는 클래스가 실제로 사용되어야 연결(link)된다. 즉 클래스 생성자는 유닛 초기화 코드의 사용보다 훨씬 더 링크를 배려한다(linker friendly).

참고 다르게 설명하자면, 클래스 생성자와 클래스 소멸자의 경우, 해당 클래스 타입이 연결(link)되지 않으면, 그 초기화 코드는 프로그램 안에 포함되지 않는다. 그러니 실행되지도 않는다. 전통적인 경우라면 그 반대가 참이다. 초기화 코드는 링커로 하여금 클래스 코드의 일부를 가져다 넣도록 한다. 실제로 다른 곳에서도 전혀 사용되지 않아도 그렇다. 실제 활용 면에서, 이것은 제스처링(gesturing) 프레임워크와 함께 도입되었다(코드 양이 상당히 많은 프레임워크다). 그 프레임워크를 사용하지 않는 실행 파일 안에는 그 코드들이 컴파일 되어 들어가지 않는다.

코드로 적으면, 다음과 같다 (ClassCtor 예제 참조).

```
type
  TTestClass = class
  public
    class var
      StartTime: TDateTime;
      EndTime: TDateTime;
    public
      class constructor Create;
      class destructor Destroy;
  end;
```

위 클래스에는 클래스 데이터 필드 두 개가 있다. 클래스 생성자 안에서 초기화 되고, 클래스 소멸자 안에서 변경된다. 한편, 사용되는 위치는 유닛의 initialization과 finalization 구역 안이다:

```
class constructor TTestClass.Create;
begin
  StartTime := Now;
end;
```



```

class destructor TTestClass.Destroy;
begin
    EndTime := Now;
end;

initialization
    ShowMessage(TimeToStr(TTestClass.StartTime));

finalization
    ShowMessage(TimeToStr(TTestClass.EndTime));

```

어떻게 될까? 시작할 때는 개발자가 의도한 순서대로 작동한다. 그 클래스 데이터는 이미 사용할 수 있고, 여러분을 그 정보를 보여줄 수 있다. 그런데, 종료할 때는, ShowMessage 호출이 해당 클래스 소멸자에서 해당 값을 대입하기보다 먼저 실행된다.

알아 둘 점이 있다. 여러분은 클래스 생성자와 클래스 소멸자에 어떤 이름이든 붙일 수 있다 (물론 Create와 Destroy는 매우 좋은 기본값이다). 하지만, 여러분은 클래스 생성자와 클래스 소멸자를 여러 개 정의하지는 못한다. 그럴 경우, 컴파일러는 오류를 일으킨다. 아래와 비슷한 메시지를 보게 될 것이다:

```

[DCC Error] ClassCtorMainForm.pas(34): E2359 Multiple class
constructors in class TTestClass: Create and Foo

```

RTL 안에 있는 클래스 생성자들 Class Constructors in the RTL

몇몇 RTL 클래스들은 이미 이 언어 특징을 잘 활용하고 있다. 예를 들어, Exception 클래스는 클래스 생성자와 클래스 소멸자 둘 다 정의하고 있다:

```

class constructor Exception.Create;
begin
    InitExceptions;
end;

```

위의 InitExceptions 프로시저 호출이 있던 예전 위치는 System.SysUtils 유닛의 initialization 구역 안이었다.

일반적으로, 클래스 생성자와 소멸자를 사용하는 것이 유닛의 초기화와 종료화 코드를 쓰는 것보다 더 좋다. 대부분의 경우, 이것은 그저 문법적 조미료 *syntactic sugar*에 불과하다. 따라서 아마 여러분은 되돌아가서 기존 코드를 수정하고 싶지 않을 수도 있을 것이다. 하지만, 만약 전혀 사용되지 않는 (즉, 그 타입의 어떤 클래스도 결코 생성되지 않는) 데이터 구조들이 초기화되어 들어가는 위험에 당면한다면, 그 코드를 클래스 생성자로 옮기는 것이 확실히 이득이다. 이런 상황은 애플리케이션 코드보다 일반 라이브러리 안에서 훨씬 더 자주 발생한다. 일반 라이브러리의 경우, 그것이 제공하는 기능들이 모두 다 사용되는 것은 아니기 때문이다.

참고 클래스 생성자를 쓰는 매우 특수한 경우로 제네릭 클래스 *generic class*가 있다. 그 내용은 제네릭 *generic*을 다루는 장에서 보게 될 것이다.

싱글톤 패턴 구현하기 Implementing the Singleton Pattern

인스턴스를 오직 하나만 생성해야 하는 클래스들이 있다. 싱글톤 패턴(흔한 디자인 패턴 중 하나)은 오브젝트가 오직 하나만 있기를 요구하며, 그 오브젝트에 대해 "전역적인 접근 지점"을 하나만 가질 것을 촉구한다.

싱글톤 패턴을 구현하는 방법은 여러 가지다. 하지만, 클래식한 방법은 Instance라는 이름을 붙인 함수를 호출해 유일한 인스턴스에 접근하는 것이다. 많은 경우, 이 구현은 나중에 초기화하기 lazy initialization 규칙을 따른다. 그래서, 이 전역 인스턴스가 생성 시점이 프로그램이 시작될 때가 아니라, 필요한 시점이 되도록 한다(그것도 단 한 번만).

아래 구현은 클래스 데이터, 클래스 메서드뿐만 아니라 클래스 소멸자까지 활용한다. 소멸자는 마지막 정리 작업 clean-up을 한다. 코드는 다음과 같다:

```
type
  TSingleton = class(TObject)
  public
    class function Instance: TSingleton;
  private
    class var TheInstance: TSingleton;
    class destructor Destroy;
  end;
class function TSingleton.Instance: TSingleton;
begin
  if TheInstance = nil then
    TheInstance := TSingleton.Create;
  Result := TheInstance;
end;
class destructor TSingleton.Destroy;
begin
  FreeAndNil(TheInstance);
end;
```

여러분은 이 클래스의 유일한 인스턴스를 잡을 수 있다(이미 생성되어 있었든 아니든 상관없다) 아래와 같이 작성하면 된다:

```
var
  ASingle: TSingleton;
begin
  ASingle := TSingleton.Instance;
```

더 나아가, 여러분이 이 클래스의 일반 생성자를 숨긴다면 hide, 즉 비공개 private로 선언하면, 이 패턴을 따르지 않고 오브젝트 생성하기가 매우 어려울 것이다.

클래스 참조 Class Reference

메서드에 관한 여러 주제를 살펴봤으니, 이제 "클래스 참조 class reference"에 관한 주제로 넘어가 동적으로 컴포넌트를 만드는 우리 예제를 확장해보자. 먼저 명심할 점이 있다.

클래스 참조는 클래스가 아니다. 오브젝트도 아니며, 오브젝트의 참조도 아니다. 그저 클래스 타입에 대한 참조다. 즉 클래스 참조 데이터 타입에 들어가는 값은 클래스다.

클래스 참조 타입은 클래스 참조 변수의 타입을 결정한다. 헷갈리는가? 코드 몇 줄을 보면 명확해질 것이다. 가령 `TMyClass`라는 클래스를 정의했다고 가정하자. 여러분은 이제 그 클래스에 관련되는 새 클래스 참조 타입을 정의할 수 있다:

```
type
  TMyClassRef = class of TMyClass;
```

이제 이 두 타입을 위한 변수를 정의한다. 첫 변수는 오브젝트를 가리킨다. 두 번째 변수는 클래스를 가리킨다:

```
var
  AClassRef: TMyClassRef;
  AnObject: TMyClass;
begin
  AClassRef := TMyClass;
  AnObject := TMyClass.Create;
```

클래스 참조가 어디에 쓰이는지 궁금할 것이다. 일반적으로, 클래스 참조는 런타임에 클래스 데이터를 조작하는데 사용된다. 여러분은 어떤 표현식 [expression](#) 안에서도 클래스 참조를 쓸 수 있다. 데이터 타입에 맞게 사용한다면 말이다. 사실 그런 표현식은 많지 않다. 하지만, 몇 가지 상당히 흥미로운 경우가 있다. 가장 간단한 경우는 오브젝트를 생성하는 표현식이다. 위의 두 줄을 아래와 같이 다시 작성해도 된다:

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

이번에 `Create` 생성자를 적용한 곳은 실제 클래스가 아니라 클래스 참조다; 즉 나는 클래스 참조를 사용해서 그 클래스의 오브젝트를 생성한 것이다.

참고 클래스 참조는 다른 OOP 언어에 있는 메타클래스 [metaclass](#)의 개념과 관련 있다. 그런데 오브젝트 파스칼에서 클래스 참조는 그 자체가 클래스는 아니다. 그저 클래스 데이터를 가리키는 참조를 정의하는 특수한 타입일 뿐이다. 따라서 메타클래스(다른 클래스를 묘사하는 클래스)라고 지칭한다면 오해의 소지가 있다. 실제로, `TMetaClass`라는 데이터 타입이 C++ Builder 안에 있다.

클래스 참조가 있다면 클래스의 어떤 메서드에도 접근할 수 있다. 그래서 만일 `TMyClass`가 `Foo`라는 클래스 메서드를 가지면, 다음 두 방식을 모두 쓸 수 있다:

```
TMyClass.Foo
AClassRef.Foo
```

클래스 참조의 타입 호환성 [type-compatibility](#) 규칙은 자신이 참조하는 클래스 타입과 똑같다. 쓸모가 많은 이유는 그 때문이다. 여러분이 클래스 참조 변수를 선언하면, 예를 들어 `MyClassRef` 라고 위에서 선언한다면, 그 변수에는 그 특정 클래스뿐만 아니라 해당 서브클래스들도 대입할 수 있다. 그래서, 만약 `MyNewClass`가 내 클래스의 서브클래스

라면 다음과 같이 쓸 수 있다:

```
AClassRef := MyNewClass;
```

이것이 왜 정말 흥미로운지를 이해하려면, 여러분이 클래스 참조를 통해 호출하는 그 클래스의 클래스 메서드들이 가상(virtual)일 수 있다는 사실, 그래서 서브클래스에서 그 메서드를 오버라이드 할 수 있음을 기억해야 한다. 여러분은 클래스 참조들과 가상 클래스 메서드들을 사용해 다형성(polymorphism)의 형태를 클래스 메서드 수준에서 구현할 수 있다. 이를 지원하는 다른 정적 OOP 언어들 거의 없다 (혹시 있을까?). 또한 모든 클래스는 TObject에서 상속을 받는다는 사실을 고려하면, 여러분은 어떤 클래스 참조에서도 TObject의 메서드 즉, InstanceSize, ClassName, ParentClass, InheritsFrom 등을 적용할 수 있다. TObject의 이 클래스 메서드들과 기타 메서드들은 17 장에서 본다.

RTL 안에 있는 클래스 참조들 Class References in the RTL

System 유닛 그리고 기타 핵심 RTL 유닛들에는 매우 많은 클래스 참조들이 선언되어 있다. 예를 들면 아래와 같다:

```
TClass = class of TObject;
ExceptClass = class of Exception;
TComponentClass = class of TComponent;
TControlClass = class of TControl;
TFormClass = class of TForm;
```

자세히 보면, TClass 클래스 참조 타입은 모든 클래스가 궁극적으로 TObject에서 파생 되기에 오브젝트 파스칼에서 쓰는 어떤 클래스의 참조라도 넣을 수 있다. TFormClass 참조는 파이어몽키(FireMonkey)나 VCL 등에 기반한 기본 오브젝트 파스칼 프로젝트의 소스코드에서 쓰인다. 두 라이브러리의 Application 오브젝트의 CreateForm 메서드는 폼 생성을 위해 폼 클래스를 파라미터로 필요로 한다:

```
Application.CreateForm(TMyForm1, MyForm);
```

첫 파라미터는 클래스 참조이고, 두번째는 생성된 오브젝트 인스턴스에 대한 참조를 받는 변수다.

클래스 참조를 사용하여 컴포넌트 만들기 Creating Components Using Class References

오브젝트 파스칼에서 클래스 참조에 실용적 사용법은 무엇일까? 런타임에 데이터 타입을 조작하는 능력은 개발 환경의 필수 요소이다. 컴포넌트 팔레트(Component Palette)에서 선택하여 폼의 새 컴포넌트를 만들 때, 여러분은 데이터 타입을 선택하고 그 데이터 타입의 오브젝트를 만든다. (사실, 이건 개발 환경이 여러분을 위해 보이지 않는 곳에서 하는 일이다.)

클래스 참조가 작동하는 원리를 설명하기 위해 ClassRef 예제를 만들었다. 클래스 참조가 작동하는 원리를 더 좋은 방법으로 설명하기 위해 ClassRef 예제를 만들었다.

이 예제에서 보이는 폼^{form}은 상당히 단순하다. 3개의 라디오 버튼^{radio button}이 폼 상단의 패널에 붙어 있다. 세 개의 라디오 버튼을 선택하고 폼을 클릭하면, 버튼 라벨에 표시된 3종의 새 컴포넌트: 라디오 버튼, 평범한 버튼, 편집 상자가 생성된다. 이 프로그램이 제대로 작동하기 위해서 컴포넌트의 이름이 유일해야 하기 때문에 세 컴포넌트의 이름을 바꿀 필요가 있다. 또 이 폼은 클래스 참조 필드가 있어야 한다:

```
private
  FControlType: TControlClass;
  FControlNo: Integer;
```

첫 필드는 사용자가 3개의 라디오 버튼 중 하나를 누를 때마다 새로운 데이터 타입을 저장하고 그 상태를 바꾼다. 여기 3개의 메서드 중 하나가 있다:

```
procedure TForm1.RadioButtonRadioChange(Sender: TObject);
begin
  FControlType := TRadioButton;
end;
```

다른 두 라디오 버튼은 이와 유사하게 TEdit나 TButton을 FControlType 필드에 대입하는 OnChange 이벤트 핸들러를 가진다. 유사한 대입이 폼의 OnCreate 이벤트 핸들러에서도 초기화 메서드 형태로 존재한다.

이 코드의 흥미로운 파트는 폼의 대부분의 화면 영역을 차지하는 TLayout 컨트롤을 클릭했을 때 실행된다. 마우스 클릭 위치를 저장하기 위한 수단으로 폼의 OnMouseDown 이벤트를 선택했다:

```
procedure TForm1.Layout1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Single);
var
  NewCtrl: TControl;
  NewName: string;
begin
  // 컨트롤 생성
  NewCtrl := FControlType.Create(Self);

  // 깜빡임을 피하기 위해, 임시로 숨김
  NewCtrl.Visible := False;

  // 부모 컴포넌트와 위치 결정
  NewCtrl.Parent := Layout1;
  NewCtrl.Position.X := X;
  NewCtrl.Position.Y := Y;

  // 고유한 이름(과 내용)을 계산
  Inc(FControlNo);
  NewName := FControlType.ClassName + FControlNo.ToString;
  Delete(NewName, 1, 1);
  NewCtrl.Name := NewName;

  // 이제 보여줌
  NewCtrl.Visible := True;
end;
```


이 메서드의 코드 첫 줄에 핵심이 있다. 여기서 `FControlType` 필드에 저장된 클래스 데이터 타입의 오브젝트가 생성된다. 그저 클래스 참조에 `Create` 생성자를 적용하기만 해도 컨트롤 오브젝트를 만들 수 있다.

이제 `Parent` 프로퍼티의 값을 정하고, 새 컴포넌트의 위치를 설정한 다음 (Text로도 자동으로 쓰이는) 이름을 부여한 뒤 보이게 `visible` 하면 된다.

특히 이름을 만들기 위한 코드에 주목하라: 오브젝트 파스칼의 기본 명명 규칙의 전통 `convention`을 모방하기 위해 나는 클래스의 이름을 `TObject` 클래스의 클래스 메서드를 사용한 `FControlType.ClassName`으로 정했다. 그리고 나는 이름의 끝에 숫자를 추가하고 문자열의 첫 글자를 제거했다.

첫 라디오 버튼에서, `FControlNo`가 1이라 할 때 `FControlType`은 `TRadioButton`이다; 그러므로 `FControlType.ClassName`은 문자열 `"TRadioButton"`을 반환하고

`FControlNo.AsString`은 우리가 클래스 이름의 첫 글자를 제거하기 전에 붙였던 1을 반환하며, 즉 새 인스턴스를 위해 붙였던 이름은 앞 글자 `"T"`를 제거한 최종적인 문자열 `"RadioButton1"`이 되는 것이다. 한결 익숙하지 않은가?

그림 12.1:

윈도우에서 구동한
ClassRef
애플리케이션의 출력 모습

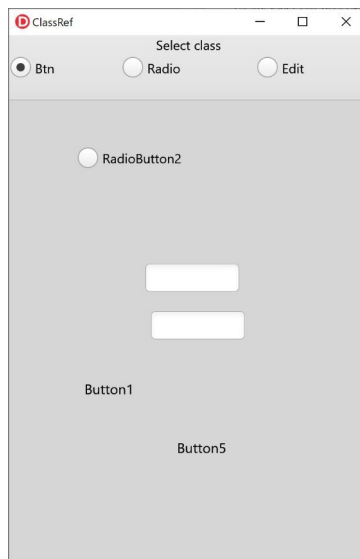


그림 12.1는 이 프로그램의 출력 모습이다. 각 컨트롤 타입에 별도의 카운터 `counter`를 사용하기에 IDE에서 보았던 것과 정확하게 같지 않은 이름을 가지는 것에 주의하자.

이 프로그램은 모든 컴포넌트에 대해 하나의 카운터만을 대신 사용한다. 그래서 `ClassRef` 애플리케이션에서 한 개의 버튼을 두고 그 옆에 라디오 버튼, 그 다음 2개의 편집 상자, 마지막으로 또 다른 버튼 하나를 만들면 그들의 이름은 그림처럼 `"Button1"`, `"RadioButton2"`, `"Edit3"`, `"Edit4"`, `"Button5"`가 된다.

폼 디자이너와 다르게, 생성한 편집 상자는 그들의 이름을 보여주지 않는다.

한편, 일반적인 컴포넌트를 생성하였을 때, 여러분은 16장에서 자세히 다룰 주제인 리플렉션^{reflection}으로 매우 동적인^{dynamic} 방법으로 그 프로퍼티에 접근할 수 있다. 같은 장에서 우리는 클래스 참조가 아니더라도 타입과 클래스 정보를 얻는 다른 방법을 볼 것이다.

클래스 헬퍼와 레코드 헬퍼 Class and Record Helpers

8장에서 클래스들의 상속 개념은 클래스에 새 요소를 제공하는 확장의 도구이면서 원래의 구현에는 영향을 미치지 않는다. 이것은 *개방-폐쇄 원칙*^{open-closed principle}이라 불리는 개념의 구현이다: 데이터 타입은 완전히 정의되었으나(폐쇄) 여전히 수정할 수 있다(개방).

타입 상속은 매우 강력한 메커니즘이지만, 그것이 이상적이지 않은 상황들이 있다. 이미 존재하는 복잡한 라이브러리들에 작업할 경우 여러분은 새 데이터 타입을 상속하지 않고 추가하고 싶을 것이다. 이 상황은 특히 오브젝트가 자동으로 생성되고 그 생성 과정을 대체하는 것이 매우 복잡할 때 그렇다.

오브젝트 파스칼 개발자에게 더 확실한 상황은 컴포넌트를 사용할 때다. 만일 컴포넌트 클래스에 새로운 동작을 하는 메서드를 추가하고 싶을 경우 상속을 이용할 수 있다. 그러나 그렇게 하면 새로운 파생된 타입이 생성되어 패키지를 설치해야 하고 폼에 있는 모든 컴포넌트를 대체해야 하며 다른 디자인 화면들을 (폼 정의와 소스 코드 파일에 모두 영향을 주면서) 새로운 컴포넌트 타입으로 바꿔야 한다.

대안이 되는 접근법은 클래스 헬퍼^{class helper}나 레코드 헬퍼^{record helper}를 이용하는 것이다. 이들 특수 목적 데이터 타입으로 기존에 존재하는 타입을 새로운 메서드와 함께 확장할 수 있다. 몇 가지 제약이 있으나, 클래스 헬퍼로 실제 컴포넌트 타입을 수정할 필요 없이 기존에 존재하는 컴포넌트에 새 메서드만 추가하도록 기획하는 상황을 다룰 수 있다.

참고 이미 우리는 인터포저 클래스라는 용어를 통해 라이브러리 클래스를 같은 이름의 클래스의 상속으로 참조를 바꿀 필요 없이 확장하는 대안을 보았다. 이전 장의 마지막 소단원에서 이 용어를 다루었다. 클래스 헬퍼는 좀 더 깔끔한 모델을 제시한다; 하지만 이전 장에서 보인 대로 가상 메서드를 바꿀 수 없으며 추가적인 인터페이스를 구현할 수 없다.

클래스 헬퍼 Class Helpers

클래스 헬퍼는 여러분이 전혀 손댈 수 없는 클래스(라이브러리 클래스 등)에 메서드와 프로퍼티를 추가하는 방법이다. 클래스 헬퍼를 사용해 여러분이 만든 코드 안에 있는

클래스를 확장하는 것은 정말 흔치 않다. 그런 경우에, 여러분은 그저 그 실제 클래스로 가서 그것을 바꾸는 것이 일반적으로 옳은 방법이다.

클래스 헬퍼라는 아이디어는 델파이의 오브젝트 파스칼에만 존재한다. 하지만, 다른 프로그래밍 언어들에도 확장^{extensions} 메서드 또는 암시적 클래스^{implicit class} 등의 개념이 있어서 이와 유사한 기능을 한다.

클래스 헬퍼 안에서 할 수 없는 일들이 있다. 인스턴스 데이터를 추가하지 못한다. 그 데이터는 실제 오브젝트 안에 존재해야 하고, 그 원래 클래스에 의해 정의되어야 하기 때문이다. 또한 가상 메서드들을 건드리지 못한다. 이것들 역시 원래 클래스의 물리적 구조 안에서 정의되어야 하기 때문이다. 즉, 클래스 헬퍼는 기존 클래스에 있는 가상이 아닌 메서드들만 추가하거나 대체할 수 있다. 이 방식을 통해 여러분은 원래 클래스의 오브젝트에 새 메서드를 적용할 수 있다. 심지어 그 원래 클래스는 그 새 메서드가 있다는 것조차 전혀 알지 못한다.

명확하게 이해되지 않는다면, 그리고 아마도 그럴 거다, ClassHelperDemo 예제를 보도록 하자 - 여러분이 해서는 안 되는 것을 보여주는 예제다. 즉, 클래스 헬퍼를 사용해서 여러분이 만든 클래스를 확장하고 있다:

```
type
  TMyObject = class
    protected
      Value: Integer;
      Text: string;
    public
      procedure Increase;
    end;

  TMyObjectHelper = class helper for TMyObject
    public
      procedure Show;
    end;
```

위 코드는 클래스와 그 헬퍼를 선언한다. 여러분은 TMyObject 타입의 오브젝트에서 원래 클래스의 메서드뿐만 아니라 클래스 헬퍼의 메서드도 호출할 수 있다는 뜻이다.

```
Obj := TMyObject.Create;
Obj.Text := 'Foo';
Obj.Show;
```

헬퍼 클래스의 메서드는 자신이 도와주고 있는 클래스의 일부가 된다(클래스 헬퍼들은 인스턴스화 되지^{instantiated} 않는다). 그래서, 그 클래스를 참조할 때 (그 클래스의 다른 메서드들이 하듯이) self를 사용할 수 있다.

```
procedure TMyObjectHelper.Show;
begin
  Show(Text + ' ' + IntToStr(Value) + ' -- ' +
    ClassName + ' -- ' + ToString);
end;
```


마지막으로, 알아 둘 점이 있다. 헬퍼 클래스의 메서드는 원래의 메서드를 오버라이드 `override`할 수 있다. 위 코드의 `Show` 메서드는 클래스와 클래스 헬퍼에 모두 추가되었다. 하지만, 헬퍼 클래스의 메서드만 호출된다!

물론, 클래스와 그 클래스의 확장(클래스 헬퍼 구문을 사용함)을 같은 유닛 안 또는 더 나아가 같은 프로그램 안에서 선언하는 것은 바람직하지 않다. 이 예제는 단지 이 구문의 기술적 사항 `technicality`을 쉽게 이해할 수 있도록 하기 위해 합쳤을 뿐이다.

클래스 헬퍼는 애플리케이션 설계의 일반적인 언어 구조 `construct`로 사용되면 *안 된다*. 오히려 라이브러리 클래스를 확장하는 목적으로 주로 사용된다. 라이브러리의 소스 코드가 여러분에게 없거나, 있더라도 추후의 충돌 `conflict`을 피하고 싶을 때 쓴다.

클래스 헬퍼, 정확히는 클래스 헬퍼 메서드에 적용되는 추가 규칙 몇 가지가 있다:

- 클래스 안에 있는 원래 메서드와 접근 지정자가 달라도 된다.
- 클래스 메서드, 인스턴스 메서드, 클래스 변수, 프로퍼티의 형태가 될 수 있다.
- 가상 메서드가 될 수 있다. 즉, 파생된 클래스가 오버라이드 하도록 할 수 있다. (용어가 실제로 조금 이상하긴 하다)
- 추가 생성자들을 만들 수 있다.
- 중첩된 상수를 타입 정의에 추가할 수 있다.

설계 측면에서 빠진 유일한 요소는 인스턴스 데이터다. 또한 주목할 점이 있다. 클래스 헬퍼는 범위 안에서 보일 때 활성화된다. 여러분은 그 클래스 헬퍼가 선언된 유닛을 가리키는 `uses` 문을 적어야 한다. 그래야 그 메서드들을 볼 수 있다. 컴파일 단계에서 한번 포함하기만 하는 것만으로는 안 된다.

참고 꽤 오랫동안, 델파이 컴파일러에 오류가 있어서, 클래스 헬퍼들이 자신이 도와주는 클래스의 비공개 `private` 필드에 접근할 수 있었다. 그 클래스가 선언된 유닛과 관계없이 말이다. 이 "꼼수 `hack`"는 기본적으로 객체 지향 프로그래밍의 캡슐화 원칙을 망가뜨렸다. 가시성 `visibility`의 의미를 강제하기 위해, 최근 버전(델파이 10.1 베를린부터)의 오브젝트 파스칼 컴파일러에서는 클래스 헬퍼와 레코드 헬퍼가 자신들이 확장하는 클래스나 레코드의 `private` 멤버에 접근할 수 없다. 즉, 이제는 이 꼼수를 활용하던 기존 코드들이 작동하지 않는다는 뜻이다 (언어 요소가 되도록 의도된 적이 없는 것이기 때문이다)

리스트 박스를 위한 클래스 헬퍼 A Class Helper for a List Box

클래스 헬퍼의 실전 사용은 라이브러리 클래스에게 추가 메서드를 제공하는 것이다. 그 클래스들을 직접 바꾸지 않고 싶을 때 (소스 코드를 갖고 있더라도 핵심 라이브러리 소스를 편집하는 것은 좋지 않다) 또는 그 클래스들을 상속받고 싶지 않을 때 (그렇게 하면, 폼들 안에 있는 컴포넌트들을 디자인할 때 바꿔야 한다) 활용할 수 있다.

간단한 예를 보자. 여러분은 리스트 박스 `list box`에 선택된 내용 `text`를 가져오려고 한다. 그러면, 아래와 같이 간단한 코드를 쓰면 된다:


```
ListBox1.Items[ListBox1.ItemIndex]
```

하지만, 여러분은 클래스 헬퍼를 정의할 수도 있다 (ControlHelper 프로젝트에서 발췌):

```
type
  TListBoxHelper = class helper for TListBox
    function ItemIndexValue: string;
  end;
function TListBoxHelper.ItemIndexValue: string;
begin
  Result := '';
  if ItemIndex >= 0 then
    Result := Items[ItemIndex];
end;
```

그러면, 이제 리스트 박스에 선택된 항목을 가져올 때 다음과 같이 할 수 있다:

```
Show(ListBox1.ItemIndexValue);
```

매우 간단한 경우다. 하지만, 그 아이디어를 매우 실용적으로 보여주고 있다.

클래스 헬퍼와 상속 Class Helpers and Inheritance

헬퍼가 가지는 가장 심각한 한계가 있다. 클래스 하나에 헬퍼를 하나만 붙일 수 있다. 만일 컴파일러가 헬퍼 클래스 두 개를 만나면, 두 번째 헬퍼가 첫 헬퍼를 대체한다. 클래스 헬퍼를 이어 붙이는 방법은 없다. 즉 한 헬퍼에 의해 이미 확장된 클래스를 다시 다른 헬퍼가 확장하지 못한다.

이 한계로 인한 문제에 대한 부분적인 해결책이 있다. 클래스 헬퍼 하나를 클래스에 붙이고 또다른 클래스 헬퍼는 상속된 클래스에 붙이면 된다; 하지만 클래스 헬퍼를 또다른 헬퍼가 상속하도록 할 수는 없다. 여러분이 복잡한 클래스 헬퍼 구조로 들어가는 것을 정말로 장하고 싶지 않다. 그러면 여러분의 코드는 매우 심각하게 뒤엎힌 덩어리가 될 것이기 때문이다.

헬퍼는 라이브러리나 시스템에 정의된 데이터 구조에 여러분이 무언가를 추가할 때 바람직하다. 그 예로 TGUID 레코드를 보자. 이것은 윈도우의 데이터 구조인데, 오브젝트 파스칼에서 헬퍼를 붙여서 흔하게 사용되는 기능들 몇 가지를 추가했다. 사실 여러분은 이것을 여러 플랫폼들에서 사용할 수 있다.

```
type
  TGUIDHelper = record helper for TGUID
    class function Create(const B: TBytes): TGUID; overload; static;
    class function Create(const S: string): TGUID; overload; static;
    // ... 다수의 Create 오버로드 생략됨
    class function NewGuid: TGUID; static;
    function ToByteArray: TBytes;
    function ToString: string;
  end;
```


위에서 TGUIDHelper가 레코드 헬퍼라는 점을 눈치챘을 것이다 (클래스 헬퍼가 아니다). 그렇다. 레코드 역시 클래스처럼 헬퍼를 가질 수 있다.

클래스 헬퍼에 컨트롤 열거형 추가하기 Adding Control Enumeration with a Class Helper

라이브러리에 있는 모든 델파이 컴포넌트들은 열거자 `enumerator`를 자동으로 정의하고 있다. 그래서 여러분은 그것이 소유하고 있는 모든 컴포넌트들 즉 자식 컴포넌트들을 처리할 수 있다. 예를 들어 폼의 메서드 안에서, 여러분은 그 폼이 소유한 컴포넌트들을 열거 `enumerate`할 수 있다. 아래와 같이 작성하면 된다:

```
for var AComp in Self do
    ... // AComp를 사용한다
```

흔한 동작을 하나 더 보자. 폼의 자식 컨트롤들을 탐색하는데, 시각적인 컴포넌트들만 (TMainMenu 등 비주얼 컴포넌트가 아닌 것을 제외) 그리고 그 폼이 직접 담고 있는 것들만(패널 위에 있는 버튼 등 자식 컨트롤 안에 딸린 자식 컨트롤 제외) 해당되도록 할 수 있다. 이런 자식들을 순환하는 작업을 우리가 더 쉽게 할 수 있도록 하려면, 새 열거자를 TWinControl 클래스에 붙이면 된다

```
type
    TControlsEnumHelper = class helper for TWinControl
    type
        TControlsEnum = class
            private
                NPos: Integer;
                FControl: TWinControl;
            public
                constructor Create(AControl: TWinControl);
                function MoveNext: Boolean;
                function GetCurrent: TControl;
                property Current: TControl read GetCurrent;
            end;
            public
                function GetEnumerator: TControlsEnum;
            end;
```

참고 위 헬퍼는 TWinControl에 붙였다. TControl에 붙이지 않았다. 윈도우 핸들이 있는 컨트롤만이 다른 컨트롤의 부모가 될 수 있기 때문이다. 그래서 기본적으로 그래픽 컨트롤은 제외된다.

아래는 이 헬퍼의 전체 코드다. 자신의 메서드 하나 그리고 TControlsEnum 즉 중첩된 클래스의 메서드들 여러 개에 대한 정의다:

```
{ TControlsEnumHelper }
function TControlsEnumHelper.GetEnumerator: TControlsEnum;
begin
    Result := TControlsEnum.Create(Self);
end;

{ TControlsEnumHelper.TControlsEnum }
```



```

constructor TControlsEnumHelper.TControlsEnum.Create(
    AControl: TWinControl);
begin
    FControl := AControl;
    NPos := -1;
end;

function TControlsEnumHelper.TControlsEnum.GetCurrent: TControl;
begin
    Result := FControl.Controls[NPos];
end;

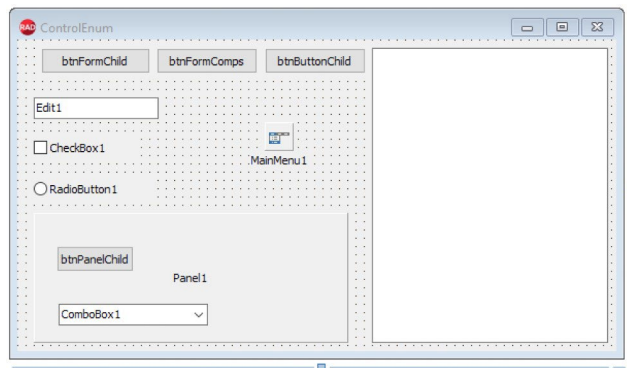
function TControlsEnumHelper.TControlsEnum.MoveNext: Boolean;
begin
    Inc(NPos);
    Result := NPos < FControl.ControlCount;
end;

```

이제 우리가 아래 그림 12.2와 같은 폼을 생성했다고 보자. 그러면, 이 열거를 다양한 상황에서 테스트할 수 있다. 먼저, 우리가 앞에서 코드로 작성했던 상황을 보자. 즉 이 폼의 자식 컨트롤들을 열거해보자:

그림 12.2:

이 폼은 컨트롤 열거 헬퍼를 테스트하는데 사용된다. 델파이 IDE 안에 있는 디자인 타임의 모습이다.



```

procedure TControlEnumForm.BtnFormChildClick(Sender: TObject);
begin
    Memo1.Lines.Add('Form Child');
    for var ACtrl in Self do
        Memo1.Lines.Add(ACtrl.Name);
    Memo1.Lines.Add('');
end;

```

위 동작은 메모 [memo](#) 컨트롤에 다음과 같이 출력한다. 잘 보면, 폼의 자식 컨트롤들이 나열된다. 하지만, 다른 컴포넌트들 또는 패널의 자식 컴포넌트들은 출력되지 않는다:

```

Form Child
Memo1
BtnFormChild
Edit1
CheckBox1
RadioButton1
Panel1
BtnFormComps
BtnButtonChild

```


우리가 모든 컴포넌트들을 열거한다면 전체 목록이 보일 것이다. 이 소단원에서 맨 먼저 보여주었던 코드를 사용하는 데 문제가 있다. 우리는 그 `GetNumerator` 메서드를 새 버전으로 오버라이드 했다. 따라서, 우리는 그 기반 `TComponent` 열거자를 더 이상 직접 접근하지 못한다. 그런데, 우리의 헬퍼는 `TWinControl`을 위해 정의되었으므로, 한 가지 트릭을 쓸 수 있다. 우리가 이 오브젝트를 `TComponent`로 캐스트 하면, 위 코드는 표준인 미리 정의된 열거자를 불러낸다:

```
procedure TControlEnumForm.BtnFormCompsClick(Sender: TObject);
begin
    Memo1.Lines.Add( 'Form Components' );
    for var AComp in TComponent(Self) do
        Memo1.Lines.Add(AComp.Name);
    Memo1.Lines.Add('');
end;
```

그 결과는 아래와 같다. 이전 목록보다 더 많은 컴포넌트들이 출력된다:

```
Form Components
Memo1
BtnFormChild
Edit1
CheckBox1
RadioButton1
Panel1
BtnPanelChild
ComboBox1
BtnFormComps
BtnButtonChild
MainMenu1
```

`ControlsEnum` 예제 안에는 위 패널의 자식 컨트롤들을 나열하는 코드들도 있다. 또한 버튼의 자식들을 나열하는 코드도 있다(이것은 컨트롤 목록이 비어 있을 경우 열거자가 잘 작동하는지 테스트하기 위함이다).

내장된 타입을 위한 레코드 헬퍼 Record Helpers for Intrinsic Types

레코드 헬퍼 개념을 더 확장하면, 네이티브 (즉 *컴파일러에 내장된*^{compiler intrinsic}) 데이터 타입에 메서드를 추가할 수 있다. “레코드 헬퍼”와 같은 구문이 사용되지만, 레코드가 아니라 정규 데이터 타입에 적용된다.

참고 레코드 헬퍼는 현재 네이티브 데이터 타입을 확장하고 메서드 비슷한^{method-like} 연산들을 추가 하는데 사용된다. 그러나 그 역시 앞으로 변할 수 있다. 지금의 런타임 라이브러리는 몇 가지 네이티브 헬퍼^{native helper}들을 정의하고 있는데 그것들은 앞으로 사라질 수 있다. 이 헬퍼들을 사용하는 여러분의 코드에는 문제가 없을 것이다. 하지만 그것들을 정의하는 코드들 안에서는 호환성이 깨진다. 따라서 이 기능을 과도하게 사용하면 안 된다. 상당히 좋고 편해도 말이다.

내장 타입 헬퍼^{intrinsic type helper}들은 어떻게 작동할까? Integer 타입에 대한 헬퍼 정의를 보자:


```

type
  TIntHelper = record helper for Integer
    function AsString: string;
  end;

```

이제 Integer 변수 N이 있다면, 다음과 같이 쓸 수 있다:

```
N.AsString;
```

이 유사 메서드 [pseudo-method](#)를 어떻게 정의하고 어떻게 그 변수의 값을 가리키도록 할 수 있을까? Self 식별자의 의미를 늘려 그 함수가 적용되는 값을 참조하게 할 수 있다:

```

function TIntHelper.AsString: string;
begin
  Result := IntToStr(Self);
end;

```

이 함수는 상수에도 적용할 수 있다는 점을 알아 두자. 예를 들면 다음과 같다:

```
Caption := 400000.AsString;
```

하지만, 상수의 값이 작으면 위와 같이 할 수 없다. 컴파일러는 상수를 다룰 때 그 상수를 담을 수 있는 타입 중 가장 크기가 작은 것으로 해석하기 때문이다. 따라서, 숫자 4는 Byte 타입으로 해석된다. 만약 4에 해당하는 문자열 값을 가져오고 싶다면 아래 코드 중 두 번째 형태를 사용해야 한다:

```

Caption := 4.AsString; // 안됨!
Caption := Integer(4).AsString; // OK

```

또는, 위 첫 번째 줄이 컴파일 되도록 하고 싶다면, 아래와 같이 헬퍼를 하나 더 정의하면 된다:

```

type
  TByteHelper = record helper for Byte...

```

2장에서 이미 봤듯이, 여러분이 위와 같은 코드를 직접 작성할 필요는 없다. 런타임 라이브러리에 이미 상당히 광범위한 클래스 헬퍼들이 있어서 핵심 데이터 타입들을 거의 모두 커버하기 때문이다. 예를 들어, System.SysUtils 유닛 안에 정의된 헬퍼들은 다음과 같다:

```

TGUIDHelper = record helper for TGUID
TStringHelper = record helper for string
TSingleHelper = record helper for Single
TDoubleHelper = record helper for Double
TExtendedHelper = record helper for Extended
TByteHelper = record helper for Byte
TShortIntHelper = record helper for ShortInt
TWordHelper = record helper for Word
TSmallIntHelper = record helper for SmallInt
TCardinalHelper = record helper for Cardinal
TIntegerHelper = record helper for Integer
TUInt64Helper = record helper for UInt64
TInt64Helper = record helper for Int64

```



```
TNativeUIntHelper = record helper for NativeUInt
TNativeIntHelper = record helper for NativeInt
TBooleanHelper = record helper for Boolean
TByteBoolHelper = record helper for ByteBool
TWordBoolHelper = record helper for WordBool
TWordBoolHelper = record helper for WordBool
TLongBoolHelper = record helper for LongBool
TCurrencyHelper = record helper for Currency // 델파이 11에서 추가됨
```

내장된 타입 헬퍼 중 몇 가지는 다음과 같이 다른 유닛 안에 정의되어 있다:

```
// System.Character:
TCharHelper = record helper for Char
// System.Classes:
TUInt32Helper = record helper for UInt32
// System.DateUtils
TDateTimeHelper = record helper for TDateTime // 델파이 11에서 추가됨
```

이 헬퍼들을 사용하는 방법은 이 책의 초반부에서 많은 예제로 다루었으므로, 지금 또 반복하지 않겠다. 이 소단원에서 추가한 것은 내장된 타입 헬퍼를 정의하는 방법에 대한 설명이다.

타입 별칭에 대한 헬퍼 Helpers for Type Aliases

앞에서 봤듯이, 두 개의 헬퍼를 같은 타입에 정의할 수 없으며, 내장된 타입은 말할 것도 없다. 그러면 Integer와 같은 네이티브 타입에 대한 직접 연산을 더 만들어 넣고 싶으면 어떻게 할까? 확실한 해답은 없지만, (내부 클래스 헬퍼의 소스 코드를 복사하여 추가 메서드를 가지도록 복제하는 등) 몇 가지 가능한 방법이 있다.

내가 좋아하는 방법은 타입 별칭 [type alias](#)을 정의하는 것이다. 타입 별칭은 컴파일러에서 볼 때 새로운 타입이다. 그러니, 타입 별칭은 타입 헬퍼를 가질 수 있다. 그 결과, 그 타입 별칭의 원래 타입에 있는 헬퍼를 교체하지 않아도 된다. 이와 같이 별개의 타입이므로, 이 두 클래스 헬퍼의 메서드들을 하나의 동일한 변수에 적용할 수는 없다. 하지만, 타입 캐스트를 하여 둘 중 하나를 배제할 수 있다. 코드를 통해 보자. 다음과 같이 타입 별칭 하나를 만든다고 가정한다:

```
type
  MyInt = type Integer;
```

이제 이 타입에 헬퍼를 정의할 수 있다:

```
type
  TMyIntHelper = record helper for MyInt
    function AsString: string;
  end;

function MyIntHelper.AString: string;
begin
  Result := IntToStr(Self);
end;
```


위에 있는 새 타입으로 변수를 선언하면, 그 변수는 새 타입의 헬퍼를 호출할 수 있다. 그러나, Integer 타입의 헬퍼를 접근하는 것도 가능하다. 캐스트를 하면 된다 - 아래 코드를 보자. (TypeAliasHelper 예제에서 발췌한 코드다):

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    MI: MyInt;  
begin  
    MI := 10;  
    Show(MI.AsString);  
    // Show(MI.ToString); // 작동하지 않음  
    Show(Integer(MI).ToString);  
end;
```


13: 오브젝트와 메모리 Objects and Memory

이 장은 매우 특수하고 꽤 중요한 주제에 초점을 맞춘다. 바로 오브젝트 파스칼 언어의 메모리 관리다. 오브젝트 파스칼 언어와 런타임 환경이 제공하는 해법이 한결 독특하다. C++ 스타일의 수동 메모리 관리와 자바 혹은 C# 스타일의 자동 가비지 컬렉션 [garbage collection/쓰레기 수집](#) 중간에 걸쳐 있다.

중도적 방식을 취하는 이유는 수동 메모리 관리의 불편함을 (전부는 아니지만) 대부분 피할 수 있으면서도, 자동 가비지 컬렉션으로 인한 제약과 문제들이 (메모리가 추가로 할당되는 것부터 파괴를 직접 결정하지 못하는 것까지) 없도록 하기 위해서다.

참고 지금 여기서 GC (가비지 컬렉션 [Garbage Collection](#)) 전략의 문제점들과 GC가 여러 플랫폼에서 어떻게 구현되는지 깊이 탐구할 의도는 없다. 오히려 연구 주제에 가깝기 때문이다. 다만 제약이 있는 장비 (모바일 장비 등)에서 GC는 전혀 이상적이지 않다는 점은 분명하다. 그런데, 똑같은 문제들 중 몇몇은 모든 플랫폼에 해당된다. 윈도우 애플리케이션들은 메모리 소비를 무시하는 추세가 있다. 그래서 우리는 메모리를 엄청나게 소비하는 작은 유틸리티들을 보게 된다.

오브젝트 파스칼이 메모리 관리 면에서 조금 더 복잡한 이유는 변수의 메모리 사용이 그 데이터 타입에 의해 결정된다는 사실 때문이다. 몇몇 타입은 참조 카운팅을 사용하고, 몇몇은 보다 전통적인 방식(예: VCL의 컴포넌트-기반 소유주 [ownership](#) 모델)이다. 그 외에도 다른 선택들이 더 있다. 그래서 메모리 관리는 복잡한 주제다. 이런 이유로 이 장에서는 그것들을 정리한다. 먼저, 현대 프로그래밍 언어들의 메모리 관리에 대한 몇 가지 기초들 그리고 오브젝트 참조 모델 뒤에 있는 개념들부터 시작한다.

참고 지난 몇 년 동안, 델파이 모바일 컴파일러는 ARC 라는 다른 메모리 모델을 제공했다. 즉 자동 참조 카운팅 [Automatic Reference Counting](#)이 제공되었다. ARC는 애플이 자체 언어 안에 넣으면서 부상했다. ARC는 컴파일러에 지원을 추가하여 오브젝트의 참조를 추적하고 카운팅을 할 수 있도록 한다. 그래서, 더 이상 필요하지 않은(즉 참조 카운트가 내려와 0이 되는) 오브젝트를 소멸하게 한다.

이와 매우 “비슷한” 방식을 델파이는 (모든 플랫폼에서) 인터페이스 참조 관리에 사용하고 있다. 10.4부터 델파이의 ARC 지원은 모든 플랫폼에서 제거되었다.

전역 데이터, 스택, 힙 Global Data, the Stack, and the Heap

어떤 플랫폼이든 오브젝트 파스칼 애플리케이션에서 쓰는 메모리 공간은 두 영역으로 나뉜다: 코드 영역과 데이터 영역이다. 코드 영역으로는, 프로그램의 실행 파일 부분, 그 (비트맵, 폼 표현 등) 리소스들, 그것이 사용하는 라이브러리들이 있다. 이것들은 프로그램의 메모리 공간 안에 적재된다. 이 메모리 블록들은 읽기 전용 [read-only](#) 이다. 그리고 (윈도우 등 몇몇 플랫폼에서는) 여러 프로세스들에게 공유될 수 있다.

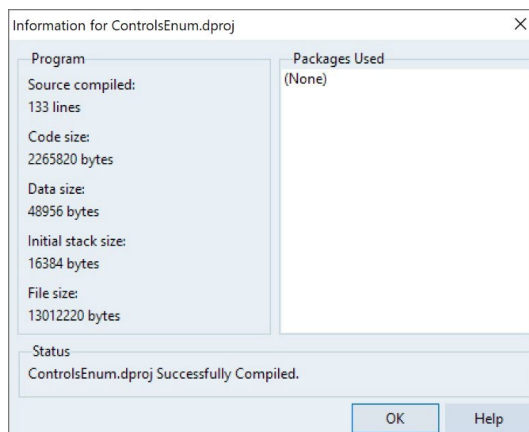
데이터 영역은 더 흥미롭다. (대부분의 다른 언어로 작성된 프로그램들도 그렇듯이) 오브젝트 파스칼 프로그램의 데이터는 명확히 구분된 세 영역 안에 저장된다: 전역 메모리 [global memory](#), 스택 [stack](#), 힙 [heap](#)이다.

전역 메모리 The Global Memory

오브젝트 파스칼 컴파일러는 실행 파일을 만드는 시점에, 그 프로그램이 작동하는 모든 시간 동안 존재할 변수들을 저장하는데 필요한 공간을 결정한다. 유닛의 인터페이스 구역 안, 또는 구현 구역 안,에 선언된 전역 변수들이 이 부류에 해당된다. 참고로, 만약 전역 변수가 클래스 타입(또는 문자열 또는 동적 배열)인 경우, 그저 4 바이트 혹은 8바이트인 오브젝트 참조가 전역 메모리 안에 저장된다.

여러분은 전역 메모리의 크기를 확인할 수 있다. 프로젝트를 컴파일한 다음에 Project | Information 메뉴 항목을 보면 된다. *Data size* 필드에 표시되는 값이 전역 메모리의 크기다. 그림 13.1에서는 48,946바이트라고 보인다. 이것이 여러분의 프로그램과 그 프로그램이 사용하는 라이브러리들이 가지는 전역 데이터의 크기다.

그림 13.1:
컴파일 된 프로그램에 관한 정보



전역 메모리는 종종 정적 메모리 `static memory`라고도 불린다. 여러분의 프로그램이 일단 적재되면, 그 변수들은 자신들의 원래 위치에 계속 존재한다. 전역 메모리는 프로그램의 수명 내내 메모리에서 해제되지 않기 때문이다.

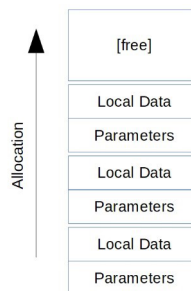
스택 The Stack

스택은 동적 메모리 `dynamic memory` 영역이다. 이 영역은 LIFO, 즉 나중에 들어간 것이 먼저 나오는 `Last In, First Out` 순서를 지키면서 할당 및 해제된다. 즉 여러분이 할당한 마지막 메모리 오브젝트가 가장 먼저 제거된다. 그림 13.2는 스택 메모리를 표현하고 있다.

스택 메모리는 보통 프로시저, 함수, 메서드 호출에서 사용된다. 파라미터나 반환값을 전달하는데 쓰인다. 또한 여러분이 함수나 메서드 안에 선언한 지역 변수를 저장하는데도 사용된다. 루틴 호출이 끝나면, 스택 위의 있던 해당 메모리 영역은 해제된다. 기억해야 할 점이 있다. 오브젝트 파스칼이 기본적 `default`으로 레지스터 호출 규약 `register-calling convention` 방식을 사용한다. 이 방식은 파라미터를 스택 대신 CPU 레지스터 안으로 전달한다. 가능하면 항상 그렇게 한다.

그림 13.2:

스택 메모리 영역에
대한 표현



또 한 가지 주목할 점이 있다. 시간을 아끼기 위해, 일반적으로 스택 메모리는 초기화 `initialize`되지 않으며 청소 `clean up`되지도 않는다. 이런 이유 때문에, 예를 들어 여러분이 `Integer`를 지역 변수로 선언하고 바로 그 값을 읽으면, 그 안에 어떤 값이 있는지 알 수 없다. 따라서, 모든 지역 변수들은 반드시 초기화하고 나서 사용해야 한다.

스택의 크기는 일반적으로 컴파일 과정에서 미리 정해진다. 여러분은 이 파라미터를 설정할 수 있다. Project | Options의 linker 페이지로 가면 된다. 그러나 기본값이면 대체로 무난하다. 만일 “스택 오버플로” 오류 메시지를 만나면, 아마도 어떤 함수가 자신을 무한히 재귀 호출하고 있기 때문일 것이다. 스택 공간 한도가 지나치게 작아서 생기는 경우는 많지 않다. 이 초기 스택 크기도 역시 Project | Information 대화 상자 안에서 볼 수 있다.

힙 The Heap

힙은 메모리의 할당과 해제가 임의의 random 순서로 일어나는 공간이다. 즉 만일 세 개 블록의 메모리를 순서대로 할당 받았다면, 그 후 해제는 아무 순서로든 가능하다. 힙 매니저 heap manager가 모든 세세한 부분을 관리한다. 따라서, 여러분은 그저 새 메모리를 요청하기만 하면 된다. 그 방법은 저-수준 함수인 GetMem을 사용하거나 또는 생성자를 호출하여 오브젝트를 생성하는 것이다. 그러면 시스템이 새 메모리 블록을 여러분을 위해 내어줄 것이다 (아마 이전에 파기된 메모리 블록을 재사용할 것이다). 오브젝트 파스칼이 힙을 사용하여 메모리를 할당하는 대상으로는 각각의 그리고 모든 오브젝트, 문자열의 텍스트, 동적 배열, 기타 대부분의 데이터 구조들이다.

힙 메모리는 동적 dynamic이다. 따라서, 일반적으로 프로그램 문제의 대부분이 발생하는 곳이 바로 힙 메모리 영역이다:

- 모든 오브젝트는 생성되면, 반드시 소멸되어야 한다. 그렇게 하지 않으면, 이른바 "메모리 누수 memory leak"가 생긴다. 메모리 누수는 의외로 너무 심한 문제가 아니다. 하지만, 계속 반복된다면, 결국 힙 메모리가 모두 소진되고 말 것이다.
- 오브젝트가 소멸할 때마다, 여러분이 확실히 해야 할 것이 있다. 그 오브젝트는 더 이상 사용되지 않도록 그리고, 프로그램이 이미 소멸한 오브젝트를 또 다시 소멸 시키려는 시도하지 않도록 해야 한다.
- 위 사실들은 동적으로 생성되는 모든 데이터 구조들의 경우에도 해당된다. 하지만, 이 언어의 런타임은 가장 자동화된 방법으로 문자열과 동적 배열을 돌본다. 따라서 그것들에 대해서는 여러분이 걱정할 필요가 없다.

오브젝트 참조 모델 The Object Reference Model

7장에서 봤듯이, 오브젝트 파스칼 언어의 오브젝트는 참조로 구현된다. 클래스 타입인 변수는 그저 하나의 포인터일 뿐이다. 그 포인터는 그 오브젝트 데이터가 존재하는 힙 heap 안의 메모리 위치를 담는다. 실제로는 위치 이외에도 (클래스 참조도 그렇듯이) 몇 가지 추가 정보 즉 그 오브젝트의 가상 메서드들의 테이블에 접근하는 방법이 더 들어 있다. 하지만, 그것은 이 장의 범위를 벗어나는 내용이다(13장의 "이 포인터는 오브젝트 참조인가?" 소단원에서 짧게 다룰 것이다).

우리는 앞에서, 오브젝트를 다른 하나에게 대입하면 오직 그 참조만 복사된다는 점 그래서, 우리는 메모리 안에 있는 하나의 오브젝트를 가리키는 두 개의 참조를 갖게 된다는 점을 봤다. 완전히 분리된 별개의 오브젝트 두 개를 가지려면, 여러분은 먼저 두 번째 오브젝트를 생성한 다음, 첫 오브젝트의 데이터를 복사해 그 안에 넣어야 한다 (이 동작은 자동으로 되지 않는다. 그 구현 세부 사항이 실제 데이터 구조에 따라 변할 수 있기 때문이다).

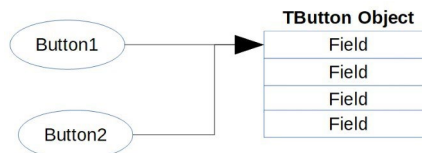
코드를 통해 보자. 만일 아래와 같이 작성한다면, 두 번째 오브젝트를 생성^{create}하는 것이 아니다. 그저 새 참조를 만들어서 기존에 존재하던 오브젝트를 가리키는 것이다:

```
var
  Button2: TButton;
begin
  Button2 := Button1;
```

즉, 메모리 안에 있는 오브젝트는 하나다. 그리고 Button1과 Button2 두 변수가 모두 그 오브젝트를 가리키게 된다. 그림 13.3과 같다.

그림 13.3:

오브젝트 참조의 복사



오브젝트를 파라미터로 전달하기 Passing Objects as Parameters

비슷한 경우가 여러분이 함수나 메서드에 오브젝트를 파라미터로 넘길 때 발생한다. 일반적으로 설명하면, 여러분은 그저 같은 오브젝트에 대한 참조만 복사하는 것이다. 그리고 그 오브젝트를 그 메서드나 함수 안에서 접근하고 그 오브젝트에 대해 연산을 수행하고 그 오브젝트의 데이터를 변경한다. 이는 그 파라미터가 `const` 파라미터로 전달되든 아니든 상관없이 그렇다.

예를 들어, 다음과 같이 이 프로시저를 작성하고 그 아래에서 호출할 경우 여러분은 Button1(이것은 프로시저 안에서는 AButton이다)의 캡션^{caption}을 수정하게 된다.

```
procedure ChangeCaption(AButton: TButton; Text: string);
begin
  AButton.Text := Text;
end;

// 호출...
ChangeCaption(Button1, 'Hello')
```

만약 대신 새 오브젝트를 생성할 필요가 있다면 어떻게 할까? 여러분은 새 오브젝트를 생성해야 한다. 그리고 관련된 각 프로퍼티를 복사해야 한다. 일부 클래스, TPersistent에서 파생된 클래스(TComponent에서 파생된 것들 제외)들 대부분은 보통 Assign 메서드를 정의하고 있어서 오브젝트의 데이터를 복사할 수 있다. 예를 들면 아래와 같다:

```
ListBox1.Items.Assign(Memo1.Lines);
```

이 프로퍼티들을 직접 대입해도, 오브젝트 파스칼이 실행하는 코드는 비슷하다: 사실, 리스트 박스의 Items 프로퍼티에는 SetItems 메서드가 연결되어 있는데, 그 안에서는 TStringList 클래스(리스트 박스의 실제 항목들을 표현함)의 Assign 메서드를 호출한다.

그러니, 파라미터를 넘기는 다양한 제어자(modifiers)가 오브젝트에 적용되었을 때 어떻게 되는지 되짚어 보자.

- **제어자가 없는 경우**, 여러분은 오브젝트와 그것을 가리키는 변수에 무슨 작업이든 할 수 있다. 여러분은 원래 오브젝트를 수정할 수 있다. 그런데, 만약 여러분이 새 오브젝트를 그 파라미터에 대입하면, 새 오브젝트는 원래의 오브젝트 그리고 그것을 참조하는 변수와 아무 상관이 없다.
- **const 제어자가 있는 경우**, 여러분은 값을 변경할 수 있고 오브젝트의 메서드를 호출할 수 있다. 하지만, 여러분은 새 오브젝트를 파라미터에 대입하지 못한다. 알아둘 점이 있다. 오브젝트는 const로 전달하더라도 성능 이득이 없다.
- **var 제어자가 있는 경우**, 여러분은 오브젝트 안의 모든 것을 변경할 뿐만 아니라, 어느 var 파라미터가 그렇듯이 호출 위치에서 원래 오브젝트를 새 오브젝트로 바꿀 수 있다. 그런데 제약이 있다. 여러분은 변수에 대한 참조를 전달해야 한다(일반 표현식은 전달 못함). 또한 참조 타입은 파라미터 타입과 정확히 일치해야 한다.
- 마지막으로 잘 알려지지 않은 선택지가 있다. **상수 참조**라고 불리며 [ref] const로 쓰는 것이다. 상수 참조로 전달되는 파라미터는 참조로 전달(var)과 비슷하게 동작한다. 하지만, 여러분이 전달하는 파라미터의 타입에 더 유연하다. 즉 정확한 타입 일치를 요구하지 않는다(서브클래스인 오브젝트 전달 가능).

메모리 관리 팁들 Memory Management Tips

오브젝트 파스칼의 메모리 관리는 세 가지 간단한 규칙으로 정리할 수 있다: 여러분은 각 오브젝트를 생성하고 필요한 모든 메모리 블록을 할당한다. 여러분이 만들고 할당한 모든 오브젝트와 메모리 블록은 반드시 소멸해야 한다. 여러분은 반드시 각 오브젝트를 단 한 번만 소멸해야 한다. 오브젝트 파스칼은 동적 요소(dynamic elements)(스택과 전역 메모리 영역이 아닌 곳에 있는 요소)에 대한 세 종류의 메모리 관리 방법을 지원한다. 자세한 사항은 이 소단원의 나머지 부분에서 다룰 것이다.

- 여러분이 오브젝트를 생성한다면, 여러분이 반드시 그 오브젝트를 소멸해야 한다. 그렇게 하지 않으면, 오브젝트가 사용하는 메모리는 해제되지 않는다. 프로그램이 끝날 때까지 다른 오브젝트가 사용하지 못한다.
- 여러분은 컴포넌트를 생성할 때 그 컴포넌트의 생성자에게 소유자(owner) 컴포넌트를 명시하여 전달할 수 있다. 소유자 컴포넌트는(종종 폼이나 데이터 모듈임) 자신이 소유하고 있는 모든 오브젝트를 소멸할 책임이 있다. 그래서 자신이 소멸할 때 그것들을 자동으로 소멸한다. 달리 말하면, 여러분이 폼이나 데이터 모듈을 소멸하면, 그것이 소유하고 있는 모든 컴포넌트도 소멸된다. 그래서 여러분이 컴포넌트를 만들고 그 소유자를 지정하면, 더 이상 그 컴포넌트를 소멸할 걱정을 하지 않아도 된다 — 하지만 여러분은 여전히 컴포넌트를 미리 소멸하는 결정을 할 수 있다. 이는 TComponent 클래스에 있는 소멸 공지 메커니즘(destruction notification mechanism) 덕분이다.

- 여러분이 메모리에 문자열, 동적 배열, 인터페이스 변수(11 장에서 설명함)에 의해 참조되는 오브젝트를 할당하면, 오브젝트 파스칼은 그 참조가 범위를 벗어났을 때 자동으로 해당 메모리를 해제한다. 여러분은 문자열을 메모리 해제를 할 필요가 없다. 더 이상 사용할 수 없게 되었을 때, 그 메모리는 자동으로 해제되기 때문이다. 메모리 영역을 확보하기 위해 여러분이 미리 해제할 필요가 있는 경우, 여러분은 문자열이나 동적 배열 변수에 nil을 대입하면 된다. 혹은 문자열 변수에 빈 문자열을 대입하면 된다.

생성한 오브젝트 소멸하기 Destroying Objects You Create

가장 간단한 상황을 보자. 여러분이 생성한 임시 오브젝트는 여러분이 소멸해야 한다. 임시가 아닌 모든 오브젝트에는 소유자가 있어야 한다. 또는 컬렉션 [collection](#)의 일부거나, 또는 몇몇 데이터 구조에 의해 참조되어야 한다. 즉, 뭔가 책임지고 적절한 시간에 그 오브젝트를 소멸할 수 있는 것이 있어야 한다.

임시 오브젝트를 생성 또는 파괴하는 코드는 일반적으로 try-finally 블록으로 둘러싼다. 그 오브젝트를 사용하는 중에 문제가 생겨도 그 오브젝트는 소멸하기 위해서다:

```
MyObj := TMyClass.Create;
try
  MyObj.DoSomething;
finally
  MyObj.Free;
end;
```

다른 흔한 상황이 있다. 오브젝트가 다른 오브젝트에 의해 사용되는 경우다 (그 오브젝트는 자신의 소유자가 된다):

```
constructor TMyOwner.Create;
begin
  FSubObj := TSubObject.Create;
end;

destructor TMyOwner.Destroy;
begin
  FSubObj.Free;
end;
```

몇 가지 더 복잡한 상황이 있다. 하위 오브젝트가 필요할 때까지 생성되지 않는 경우 (나중에 초기화하기 [lazy initialization](#) 등) 또는 소유자보다 먼저 소멸하는 경우(더 이상 필요하지 않는 경우)가 있다.

'나중에 초기화하기'를 구현하려면, 여러분은 소유자의 생성자 안에서 하위 오브젝트를 생성하지 않는다. 필요할 때 생성한다:

```
function TMyOwner.GetSubObject: TSubObject
begin
  if not Assigned(FSubObj) then
```



```

    FSubObj := TSubObject.Create;
    Result := FSubObj;
end;

destructor TMyOnwer.Destroy;
begin
    FSubObj.Free;
end;

```

알아 둘 것이 있다. 위 코드는 오브젝트가 처음 생성될 때, 그 오브젝트의 메모리가 초기화된다는 사실에 기반하고 있는 코드다. 즉, FSubObj은 nil이다. 또 한 가지 알아 둘 것이 있다. 여러분은 오브젝트를 free^{해제}하기 전에 그 오브젝트가 할당되어 있는지 여부를 테스트할 필요가 없다. free는 그것도 수행한다. 다음 소단원에서 보여주겠다.

오브젝트를 한 번만 소멸하기 Destroying Objects Only Once

또다른 문제가 있다. 여러분이 오브젝트의 소멸자를 두 번 호출하면 오류가 발생한다. 소멸자^{destructor}는 오브젝트의 메모리 할당을 해제하는 메서드다. 우리는 소멸자를 위한 코드를 작성할 수 있다. 일반적으로 기본 Destroy 소멸자를 오버라이딩 한다. 그래서 그 오브젝트가 소멸되기 전에 몇몇 코드를 실행하도록 한다.

Destroy는 TObject 클래스의 가상 소멸자다. 오브젝트가 소멸할 때 사용자 지정 정리 코드가 필요한 클래스들은 대부분 이 가상 메서드를 오버라이드 한다. 여러분이 새 소멸자를 정의하면 안 되는 이유가 있다. 오브젝트는 주로 Free 메서드를 호출하여 소멸된다. 그런데 이 메서드는 여러분을 위해 Destroy 가상 소멸자(오버라이드 된 버전일 수 있음)를 호출하기 때문이다.

이미 말했듯이, Free는 모든 클래스의 조상인 TObject 클래스의 메서드다. Free 메서드는 기본적으로 Destroy 가상 소멸자를 호출하기 전에 먼저 현재 오브젝트(Self)가 nil인지 확인한다.

참고 오브젝트 참조가 nil인 경우, Free는 안전하게 호출할 수 있는데, Destroy는 왜 호출할 수 없는지 궁금할 것이다. Free는 알려진 메서드이고 주어진 메모리 위치에 있기 때문에 안전하다. 하지만, 가상 함수인 Destroy는 런타임에 그 오브젝트의 타입을 보면서 결정되므로, 오브젝트가 더 이상 존재하지 않을 경우에는 매우 위험한 동작이다.

여기 Free에 대한 의사-코드^{pseudo-code}가 있다:

```

procedure TObject.Free;
begin
    if Self <> nil then
        Destroy;
end;

```

다음으로, Assigned 함수를 주목하자. 우리가 이 함수에 포인터를 전달하면, 이 함수는 그저 그 포인터가 nil인지만 검사한다. 그래서 아래 두 문장은 똑같다. 적어도 대부분의 경우에는 그렇다:


```

if Assigned(MyObj) then
  ...
  if MyObj <> nil then
    ...

```

유의할 점이 있다. 위 문장들은 그 포인터가 `nil`이 아닌지를 확인한다; 위 문장들은 그 포인터가 유효한 포인터인지는 확인하지 않는다. 만일 다음과 같이 코드를 쓰면:

```

MyObj.Free;
if MyObj <> nil then
  MyObj.DoSomething;

```

테스트 결과는 `True`다. 그리고 여러분은 오브젝트의 메서드를 호출하는 줄에서 오류를 얻는다. 우리가 알아야 할 중요한 점이 있다. `Free` 호출은 오브젝트의 참조를 `nil`로 지정하지는 않는다.

자동으로 오브젝트를 `nil`로 지정하는 것은 불가능하다. 여러분은 여러 참조들로 같은 오브젝트를 가리킬 수 있다. 그리고 오브젝트 파스칼은 그 참조들을 추적하지 않는다. 동시에, (`Free`와 같은) 메서드에서 오브젝트에 동작을 수행하더라도, 오브젝트 참조—그 메서드를 호출하기 위해 우리가 사용한 변수의 메모리 주소—에 대해 우리는 전혀 알지 못한다.

즉 `Free` 메서드나 클래스의 다른 메서드 안에서, 우리는 오브젝트의 메모리 주소 (`Self`)를 알 수 있으나 `MyObj` 같은 오브젝트를 가리키는 변수의 메모리 위치는 알 수 없다. 그러므로 `Free` 메서드는 `MyObj` 변수에 영향을 주지 않는다.

그러나, 우리가 외부 함수를 호출할 때, 오브젝트를 참조로-전달 파라미터로 넘겨주면 그 외부 함수는 변경할 원래의 오브젝트 참조를 선택할 수 있다. 이것이 바로 사용할 수 있도록 미리 준비된 `FreeAndNil` 프로시저가 하는 일이다. `Free` 후에 그 참조 변수를 `nil`로 지정한다. 아래는 현재 `FreeAndNil`의 코드다:

```

procedure FreeAndNil(const [ref] Obj: TObject); inline;
var
  Temp: TObject;
begin
  Temp := Obj;
  TObject(Pointer(@Obj)^) := nil;
  Temp.Free;
end;

```

예전에는, 이 파라미터가 그저 포인터였다. 그래서 단점이 있었다. 여러분이 `FreeAndNil` 프로시저에게 날것인 `raw` 포인터, 인터페이스 참조, 기타 호환되지 않는 데이터 구조를 전달할 수 있었다. 따라서, 종종 메모리 영역 손상(`corruption`)과 찾기 힘든 버그들을 일으켰다. 델파이 10.4부터 이 코드는 위와 같이 변경되었다. `TObject` 타입인 `const reference` 파라미터를 사용한다. 즉, 파라미터를 오브젝트로만 제한한다.

참고 몇몇 델파이 전문가들은 `FreeAndNil`을 사용해서는 안 된다고 주장한다. 그 이유로 오브젝트를

가리키는 변수의 가시성^{visibility}과 그 수명^{lifetime}이 일치해야 한다는 점을 든다. 오브젝트가 다른 오브젝트를 소유하고 있고, 소멸자 안에서 자신이 소유하고 있는 다른 오브젝트를 Free 하는 경우에는 여러분이 그 참조를 nil로 설정할 필요가 없다. 여러분이 더 이상 사용하지 않을 그 소유자 오브젝트의 일부이기 때문이다. 이와 유사하게, try-finally 블록 안에서 Free 하는 로컬^{local} 변수도 nil로 설정할 필요가 없다. 곧 바로 범위를 벗어나기 때문이다.

여담으로, Free 메서드 말고도 TObject에는 DisposeOf 메서드도 있다. 이 언어가 몇 년 간 ARC를 지원하면서 생긴 흔적이다. 현재 DisposeOf 메서드는 그저 Free를 호출한다.

메모리 정리^{clean-up} 동작들 사용에 대해 요약한 몇 가지 가이드라인은 다음과 같다:

- 항상 Free로 오브젝트를 소멸하라. Destroy 소멸자를 호출하는 대신!
- FreeAndNil을 사용하라. 또는 Free 호출 후에는 오브젝트 참조를 nil로 설정하라. 단, 그 참조가 곧바로 범위를 벗어나는 경우가 아니라면 말이다.

메모리 관리와 인터페이스 Memory Management and Interfaces

11장에서 인터페이스를 위한 메모리 관리의 핵심 요소들을 봤다. 그 때 언급했듯이, 오브젝트와 달리, 인터페이스는 관리된다 또한 참조 카운트가 적용된다. 이미 언급한 대로, 인터페이스 참조는 그것이 참조하고 있는 오브젝트의 참조 카운트를 증가시킨다. 하지만, 여러분은 인터페이스 참조를 약함^{weak}으로 선언할 수 있다. 그렇게 선언되는 참조에서는 참조 카운팅이 비활성화 된다(그래도 그 참조를 컴파일러가 관리한다는 점은 여전하다). 또한 여러분은 unsafe 제어자를 사용하여 선언할 수도 있다. 그렇게 선언된 참조에서는 모든 컴파일러 지원이 완전히 비활성화 된다. 이 소단원은 이것에 대해 좀 더 깊이 본다. 그리고 11장에서 소개한 것들에 예시를 몇 가지 추가한다.

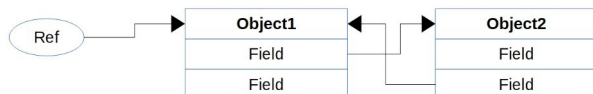
Weak 참조에 대한 더 많은 것들 More on Weak References

델파이가 인터페이스에 사용한 참조 카운팅 모델의 한 문제는, 두 오브젝트가 서로를 참조할 경우 순환 참조^{circular reference}를 이루며, 참조 카운트가 기본적으로 0이 되지 않는다는 점이다. 약한 참조는 이 순환을 끊는^{break} 메커니즘을 제공하여, 참조 카운트를 증가시키지 않는 참조를 정의할 수 있게 한다.

두 인터페이스가 각자의 필드를 사용해 서로를 참조하고 있고, 외부 변수가 첫 번째 오브젝트를 참조한다고 가정하자. 첫 오브젝트의 참조 카운트는 2 다 (외부 변수와 두 번째 오브젝트의 필드가 참조함). 하지만, 두 번째 오브젝트의 참조 카운트는 1이다 (첫 번째 오브젝트의 필드가 참조함). 그림 13.4는 이 상황을 묘사한다.

그림 13.4:

약한 참조로 해결 가능한
오브젝트 간의 순환하는 참조



이제, 외부 변수가 자신의 범위를 벗어나게 되면 어떻게 될까? 이 두 오브젝트 모두 참조 카운트는 1이 되고, 메인 오브젝트인 Object1은 Object2를 소유하고 있는데, 외부 소유자는 없는 상태가 된다. 따라서 이 두 오브젝트들은 메모리에 영원히 남게 된다. 이런 종류의 상황을 풀려면, 여러분은 순환 참조를 끊어야 한다. 결코 간단하지 않다. 참조를 끊어야 하는 시점을 여러분이 모른다면 말이다 (마지막 외부 참조가 범위를 벗어나는 순간에 수행되어야 한다. 그런데, 그 오브젝트들은 그 시점을 알 수 없다).

위 상황 그리고 비슷한 많은 상황에서 해답은, 약한 참조를 사용하는 것이다. 앞에서 말했듯이, 약한 참조는 자신이 참조하는 오브젝트의 참조 카운트를 증가시키지 않는 참조다. 기술적으로, 약한 참조는 [weak] 애트리뷰트 [attribute](#)을 적용하여 정의한다.

참고 애트리뷰트는 오브젝트 파스칼 언어의 고급 기능이다. 16장에서 다룬다. 지금은 한 심볼(symbol)에 대한 런타임 정보 몇 가지를 추가하는 방법이 있고, 그래서 외부 코드는 어떻게 그것을 다루는 지를 결정할 수 있다는 점만 알아 두자.

앞의 상황에서, 만약 두 번째 오브젝트의 참조를 통해 첫 번째 오브젝트로 가는 부분이 약한 참조라면(그림 13.5와 같다), 외부 변수가 범위를 벗어났을 때 두 오브젝트 모두 소멸된다.

그림 13.5:

그림 13.4의 참조
순환이 약한 참조에
의해 끊어짐



이 간단한 상황을 코드로 보자. 먼저, 인터페이스 두 개를 선언한다. 그런데, 하나가 다른 하나를 참조하고 있다 (ArcExperiments 애플리케이션 예제에서 발췌함):

```

type
  IMySimpleInterface = interface
    ['{B6AB548A-55A1-4D0F-A2C5-726733C33708}']
    procedure DoSomething(BRaise: Boolean = False);
    function RefCount: Integer;
  end;

  IMyComplexInterface = interface
    ['{5E8F7B29-3270-44FC-B0FC-A69498DA4C20}']
    function GetSimple: IMySimpleInterface;
    function RefCount: Integer;
  end;
  
```

이 프로그램의 코드에서 클래스 두 개를 정의한다. 각 클래스는 위의 인터페이스를 하나씩 맡아서 구현한다. 교차-참조를 다루는 방식을 주목하자 (FOwnedBy와 FSimple은 인터페이스에 기반하고 있다. 그 중 하나는 weak로 정의되었다):


```

type
  TMySimpleClass = class(TInterfacedObject, IMySimpleInterface)
  private
    [Weak] FOwnedBy: IMyComplexInterface;
  public
    constructor Create(Owner: IMyComplexInterface);
    destructor Destroy; override;
    procedure DoSomething(BRaise: Boolean = False);
    function RefCount: Integer;
  end;

  TMyComplexClass = class(TInterfacedObject, IMyComplexInterface)
  private
    FSimple: IMySimpleInterface;
  public
    constructor Create;
    destructor Destroy; override;
    function GetSimple: IMySimpleInterface;
    function RefCount: Integer;
  end;

```

TMyComplexClass 클래스의 생성자는 아래와 같다(다른 클래스의 오브젝트를 생성한다):

```

constructor TMyComplexClass.Create;
begin
  inherited Create;
  FSimple := TMySimpleClass.Create(Self);
end;

```

FOwnedBy 필드가 약한 참조로 되어있었다는 점을 기억하자. 따라서 자신이 참조하는 오브젝트의 참조 카운트를 늘리지 않는다. 위 코드에서는 현재의 오브젝트(Self)다. 이 코드 구조라면, 우리는 아래와 같이 작성할 수 있다:

```

class procedure TMyComplexClass.CreateOnly;
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.FSimple.DoSomething;
end;

```

위 코드는 메모리 누수가 생기지 않는다. 약한 참조를 사용하기 때문이다. 예를 들어, 이런 코드를 작성할 수 있다:

```

var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  Log('Complex = ' + MyComplex.RefCount.ToString);
  MyComplex.GetSimple.DoSomething(False);

```

각 생성자와 소멸자가 그 실행을 기록한다면, 이런 기록^{log}을 볼 수 있다:

```

Complex class created
Simple class created

```



```
Complex = 1
Simple class doing something
Complex class destroyed
Simple class destroyed
```

만약 weak 애트리뷰트를 코드에서 제거한다면, 여러분은 메모리 누수를 볼 수 있다. 또한 (위 코드를 실행하면) 참조 카운트가 1이 아닌 2가 되는 것을 볼 수 있다.

약한 참조는 관리된다 Weak References are Managed

매우 중요한 점이 있다. 약한 참조는 관리된다^{managed}. 즉 시스템은 약한 참조들의 목록을 메모리 안에 유지하고 있다가, 그 오브젝트가 소멸할 때 그것을 참조하고 있는 약한 참조들이 있는지 확인한다. 만약 있다면, 그 실제 참조들을 nil로 지정한다. 즉 그 실제 외부 참조 포인터들은 nil이 된다. 이는 약한 참조가 런타임 비용^{cost}이 있다는 의미다.

관리되는 약한 참조의 장점은, 전통적 방법에 비해, 인터페이스 참조가 유효한지 아닌지 (즉 참조하는 오브젝트가 소멸되었는지)를 여러분이 확인할 수 있다는 것이다. 하지만, 여러분이 약한 참조를 사용하면, 항상 사용 전에 할당 여부를 확인해야 한다는 의미다.

ArcExperiments 애플리케이션 샘플을 보자. 이 품에는 비공개^{private} 필드가 하나 있는데, IMySimpleInterface 타입이며, 약한 참조로 선언되어 있다:

```
private
[weak] MySimple: IMySimpleInterface;
```

또한 버튼이 하나 있어서 해당 필드에 대해 참조를 대입한다. 그리고 또 다른 버튼은 그 필드가 여전히 유효한지 확인하고 나서 사용한다:

```
procedure TForm3.BtnGetWeakClick(Sender: TObject);
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.GetSimple.DoSomething(False);
  MySimple := MyComplex.GetSimple;
end;

procedure TForm3.BtnUseWeakClick(Sender: TObject);
begin
  if Assigned(MySimple) then
    MySimple.DoSomething(False)
  else
    Log('Nil weak reference');
end;
```

여러분이 코드를 수정하지 않는다면, "if Assigned" 검사는 실패한다. 첫 번째 버튼의 이벤트 핸들러는 이 오브젝트들을 생성하고 바로 해제한다. 따라서 이 약한 참조는 nil이 된다(유효하지 않기 때문이다). 그러나 이 참조는 관리되므로, 컴파일러는 그 실제 상태를 여러분이 추적할 수 있게 도와준다(오브젝트에 대한 참조와 다른 점임).

Unsafe 애트리뷰트 The Unsafe Attribute

특수한 상황(예를 들어, 인스턴스를 생성 중인 상황)에서는 함수가 반환하는 오브젝트의 참조 카운트가 0인 경우가 있다. 그럴 땐, (변수에 대입하여 그 참조 카운트를 1로 만들 기회도 갖기 전에) 컴파일러가 오브젝트를 즉시 제거하지 않도록 막아야 한다. 그러려면, 그 오브젝트에 *unsafe*(안전하지 않음) 표시를 해야 한다.

이 표시는 오브젝트의 참조 카운트를 일시적으로 무시하라는 의미다. 이를 통해 그 코드가 *안전해지도록* 한다. 그렇게 동작하게 하려면, [Unsafe] 애트리뷰트를 사용한다. 여러분은 매우 특수한 상황에서만 이 새 애트리뷰트가 필요할 것이다.

구문은 다음과 같다:

```
var
  [Unsafe] Intf1: IInterface;

[Result: Unsafe] function GetIntf: IInterface;
```

이 애트리뷰트 사용이 합리적인 경우는, (팩토리 [factory](#) 패턴 등) 생성 패턴 [construction pattern](#)을 범용 라이브러리에서 구현할 때를 꼽을 수 있다.

참고 (지금은 단종된) ARC 메모리 모델을 지원하기 위해, System 유닛은 *unsafe* 지시어를 사용했다. 왜냐하면 이 애트리뷰트를 (같은 유닛의 뒤쪽에 있는) 그 정의보다 앞쪽에서는 사용할 수 없기 때문이었다. System 유닛 밖에 있는 어떤 코드도 이것을 사용하지 않을 것이다. 그리고 이제는 이것이 더 이상 사용되지 않는다 (여러분은 `$IFDEF` 지시어 안에서 이것을 볼 수 있을 것이다).

메모리를 추적하고 확인하기 Tracking and Checking Memory

이 장에서는 오브젝트 파스칼의 메모리 관리 기초를 보았다. 대부분의 경우, 여기서 강조한 규칙들만 적용해도 여러분의 프로그램은 충분히 안정적이고, 과도한 메모리 사용을 피하고, 기본적으로 여러분이 메모리 관리에 대해 잊어도 될 것이다. 견고한 애플리케이션을 작성하는 모범 사례들은 더 있다. 이 장의 후반에 살펴볼 것이다.

이 소단원에서는 여러분이 메모리 사용량을 추적하고, 비정상적인 상황을 감시하며, 메모리 누수 [memory leak](#)를 찾을 수 있는 기법들에 집중할 것이다. 이것은 개발자들에게 중요한 지식이다. 비록 이 언어의 한 부분이라고 말할 수는 없고 런타임 지원에 더 가깝지만 말이다. 또한, 메모리 매니저 구현은 타겟 플랫폼과 운영체제에 따라 다르다. 심지어 여러분은 사용자 지정 메모리 매니저를 오브젝트 파스칼 애플리케이션 안에 끼워 넣을 [plug-in](#) 수도 있다(흔한 일은 아니다).

메모리 상태 추적, 메모리 매니저, 누수 탐지에 관한 모든 논의는 *힙 메모리*에만 관련 있음을 기억하자. 스택과 전역 메모리는 다르게 관리되므로 여러분은 거기 개입할 능력이 없지만, 그것들은 거의 문제를 일으키지 않는 영역이다.

메모리 상태 Memory Status

힙 메모리 상태를 어떻게 추적할까? RTL에는 편리한 함수인 `GetMemoryManagerState`와 `GetMemoryMap`이 있다. 메모리 매니저 상태로는 크기가 다양한 여러 가지 할당된 블록들의 개수를 알 수 있다. 그런데, 힙 맵 [heap map](#)은 꽤 멋지다. 시스템 수준에서 애플리케이션의 메모리 상태를 묘사하기 때문이다.

여러분은 각 메모리 블록들의 실제 상태를 다음 코드와 같이 써서 확인할 수 있다:

```
for I := Low(AMemoryMap) to High(AMemoryMap) do
begin
  case AMemoryMap[I] of
    csUnallocated: ...
    csAllocated: ...
    csReserved: ...
    csSysAllocated: ...
    csSysReserved: ...
  end;
end;
```

이 코드는 `ShowMemory` 예제에서 애플리케이션의 메모리 상태를 그림으로 표현하는데 쓰였다.

FastMM4

윈도우 플랫폼에서, 현재 오브젝트 파스칼의 메모리 매니저는 FastMM4라고 불린다. 오픈 소스 [open source](#) 프로젝트이며 Pierre La Riche가 주로 개발한다. 다른 플랫폼에서 델파이는 그 플랫폼의 네이티브 [native](#) 메모리 매니저를 사용한다.

FastMM4은 메모리 할당량을 최적화 [optimize](#)하고, 속도를 높이며 다음 사용을 위한 더 많은 RAM을 해제한다. FastMM4은 효율적인 메모리 정리나, 인터페이스 기반 접근을 포함한 제거된 오브젝트의 부정확한 사용이나, 메모리 덮어쓰기 [overwrite](#)나 버퍼 오버런 [buffer overrun](#)에 대해 시행하는 확장된 메모리 검사에 적합하다. 또 이것은 잔여 [left-over](#) 오브젝트에 대한 피드백 [feedback](#)을 제공하여 메모리 누수 추적에 도움을 준다.

FastMM4에 관한 몇 가지 고급 기능들은 ("완전한 FastMM4에서의 버퍼 오버런" 소단원에서 다룰) 라이브러리의 완전한 버전 [full version](#)에만 들어있으며, 표준 RTL에 들어있는 버전에는 없다. 그래서 완전한 버전의 기능을 원한다면, 여러분은 여기서 완전한 소스 코드를 받아야 한다:

■ <https://github.com/pleriche/FastMM4>

참고 이 라이브러리의 새 버전인 FastMM5가 있다. 이것은 멀티-쓰레드 [multi-threaded](#) 애플리케이션들을 위해 특히 최적화되었다. 그리고 큰 멀티-코어 [multi-core](#) 시스템에서 성능이 훨씬 더 좋다. 이 새 버전 라이브러리는 GPL 라이선스(오픈 소스 프로젝트들용) 그리고 유료 상용 라이선스(충분한 값어치를 한다)로 제공된다. 더 많은 정보는 <https://github.com/pleriche/FastMM5>의 프로젝트 *read me*에 있다.

누수 추적하기와 다른 전역 설정들 Tracking Leaks and Other Global Settings

FastMM4의 RTL 버전은 튜닝할 수 있다. System 유닛의 전역 설정을 사용하면 된다. 알아 둘 점이 있다. 해당 전역 선언들은 System 유닛에 있지만, 실제 메모리 매니저는 getmem.inc RTL 소스 코드 파일 안에 구현되어 있다.

가장 쉽게 사용할 수 있는 설정은 ReportMemoryLeaksOnShutdown 전역 변수다. 이것은 여러분이 메모리 누수를 쉽게 추적할 수 있도록 한다. 여러분은 프로그램 실행의 맨 앞에서 이것을 켜야 한다. 프로그램이 종료되면, 여러분의 코드 (또는 여러분이 사용하는 라이브러리)에 메모리 누수가 있는지 알려줄 것이다.

참고 메모리 매니저의 고급 설정으로는 NeverSleepOnMMThreadContention 전역 변수를 통해 멀티쓰레드로 할당하기; GetMinimumBlockAlignment 함수와 SetMinimumBlockAlignment 함수를 통해 몇몇 SSE 명령^{operation}을 가속하기 (메모리 사용은 더 큼), RegisterExpectedMemoryLeak 전역 프로시저 등을 호출을 통해 예상되는 메모리 누수를 등록하기 등등이 있다.

표준 메모리 누수 리포트^{leak reporting}와 등록을 보여주려고, 여기 간단한 LeakTest 예제를 썼다. 이 OnClick 핸들러는 그 예제 안에 있는 버튼에 연결되어 있다:

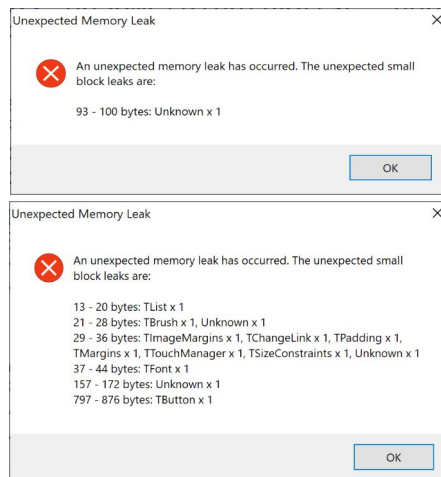
```
var
  P: Pointer;
begin
  GetMem(P, 100); // 메모리 누수 발생!
end;
```

이 코드는 잃어버리고 누수 되는 100바이트의 메모리를 할당한다. 만약 LeakTest 프로그램을 IDE 내에서 실행하고 첫 버튼을 한 번 누른 후, 프로그램을 닫을 경우 그림 13.6의 윗부분과 같은 메시지를 얻는다.

이 프로그램의 다른 “누수”는 TButton 생성 후 메모리 안에 그냥 두면 생긴다. 그런데, 이 오브젝트는 하위 요소들이 많다. 따라서, 그림 13.6의 아래와 같이 누수 리포트가 더 복잡하다. 여전히, 우리는 메모리 누수 자체에 관해 일부 제한된 정보만 가진다.

그림 13.6:

LeakTest 애플리케이션의 종료 시
윈도우 메모리 매니저에 의해
보고된 메모리 누수



프로그램은 또한 해제되지 않을 전역 포인터에 약간의 메모리를 할당하지만, 이곳의 잠재적인 누수를 기대된 것으로 등록하므로 보고되지 않는다.

```
procedure TFormLeakTest.FormCreate(Sender: TObject);
begin
    GetMem(GlobalPointer, 200);
    RegisterExpectedMemoryLeak(GlobalPointer);
end;
```

다시 말하지만 FastMM4를 기본적으로 사용하는 윈도우 플랫폼에서만 기본적인 누수 보고가 가능하다.

완전한 FastMM4에서의 버퍼 오버런 탐지 Buffer Overruns in the Full FastMM4

이것은 고급 단계의 주제이며, 윈도우 플랫폼에 한정한 주제이므로, 가장 숙련된 개발자만 이 소단원을 읽을 것을 추천한다.

누수^{leak} 보고에 대해 더 많이 제어하기를 바라거나(파일에 기록하는 기능 활성화 등), 할당 정책을 세밀하게 조정^{fine-tune}하거나, FastMM4가 제공하는 메모리 검사들을 사용하고 싶다면, 완전한 버전^{full version}을 다운로드해야 한다. 그 버전에는 이 FastMM4.pas 파일 그리고 추가로 해당 설정 파일인 FastMM4Options.inc이 들어 있다.

설정을 세밀하게 조정하기 위해 편집할 파일은 후자로, 많은 수의 지시어를 주석 처리^{commenting}하거나 해제^{uncommenting}하면 된다. 전통적으로 이것은 \$DEFINE 문장 이전에 마침표^{period}를 두어 포함 파일에서 가져온 다음 첫 두 줄처럼 순수한 주석으로 바꾸면 된다.

```
{.$DEFINE Align16Bytes} // 주석
{$DEFINE UseCustomFixedSizeMoveRoutines} // 활성화된 설정
```

이 예제를 위하여, 몇 가지 관련된 설정을 바꾸었으며, 이런 정의로 가능한 아이디어를 주기 위해 표시하였다:

```
{$DEFINE FullDebugMode}
{$DEFINE LogErrorsToFile}
{$DEFINE EnableMemoryLeakReporting}
{$DEFINE HideExpectedLeaksRegisteredByPointer}
{$DEFINE RequireDebuggerPresenceForLeakReporting}
```

이 테스트 프로그램은(FastMMCode 폴더에 있다. 거기에는 여기서 사용한 FastMM4 버전의 전체 소스도 여러분을 위해 넣어 두었다) 이 사용자 지정 메모리 매니저 버전을 활성화하고 있다. 그러기 위해, 첫 번째 유닛으로 지정하고 있다:

```
program FastMMCode;

uses
    FastMM4 in 'FastMM4.pas',
    Forms,
    FastMMForm in 'FastMMForm.pas'; {Form1}
```


이 코드가 작동하려면 FastMM_FullDebugMode.dll 파일의 로컬 복사본도 필요하다. 이 예제 프로그램은 캡션 길이 `Length(Caption)`가 제공된 5자보다 크기 때문에 로컬 버퍼에 들어갈 수 있는 것보다 더 많은 텍스트를 가져와 버퍼 오버런을 발생시킨다.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  PCh1: PChar;
begin
  GetMem(PCh1, 5);
  GetWindowText(Handle, PCh1, Length(Caption));
  ShowMessage(PCh1);
  FreeMem(PCh1);
end;
```

메모리 매니저는 각 메모리 블록의 앞과 끝에 특별한 값을 넣어 추가 바이트들을 할당하며, 각 메모리 블록을 해제할 때 이들 값을 확인한다. 그래서 FreeMem을 호출할 때 오류를 얻는다. (디버거에서) 버튼을 누를 때 여러분은 파일에도 기록되는 매우 긴 오류 메시지를 보게 된다.

FastMMCode_MemoryManager_EventLog.txt

(아래 일부만 표현한) 이것이 현재 스택 흔적 `stack trace` 및 메모리 덤프와 함께 오버런 `overrun` 오류를 할당 및 해제 동작 시의 스택 흔적과 함께 출력한 것이다.

```
FastMM has detected an error during a FreeMem operation.The block
footer has been corrupted.
```

```
The block size is: 5
```

```
Stack trace of when this block was allocated (return addresses):
```

```
40305E [System][System.@GetMem]
44091A [Controls][Controls.TControl.Click]
44431B [Controls][Controls.TWinControl.WndProc]
42D959 [StdCtrls][StdCtrls.TButtonControl.WndProc]
44446C [Controls][Controls.DoControlMsg]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
443A43 [Controls][Controls.TWinControl.MainWndProc]
41F31A [Classes][Classes.StdWndProc]
76281A10 [GetMessageW]
```

```
The block is currently used for an object of class: Unknown
```

```
The allocation number is: 381
```

```
Stack trace of when the block was previously freed (return addresses):
```

```
40307A [System][System.@FreeMem]
42DB8A [StdCtrls][StdCtrls.TButton.CreateWnd]
443863 [Controls][Controls.TWinControl.UpdateShowing]
44392B [Controls][Controls.TWinControl.UpdateControlState]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
44009F [Controls][Controls.TControl.Perform]
43ECDf [Controls][Controls.TControl.SetVisible]
45F770
```



```
76743833 [BaseThreadInitThunk]
```

The current stack trace leading to this error (return addresses):

```
40307A [System][System.@FreeMem]
44091A [Controls][Controls.TControl.Click]
44431B [Controls][Controls.TWinControl.WndProc]
42D959 [StdCtrls][StdCtrls.TButtonControl.WndProc]
44446C [Controls][Controls.DoControlMsg]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
443A43 [Controls][Controls.TWinControl.MainWndProc]
41F31A [Classes][Classes.StdWndProc]
76281A10 [GetMessageW]
```

Current memory dump of 256 bytes starting at pointer address 133DEF8:
46 61 73 74 4D 4D 43 6F 64 [... omitted...]

아주 잘 보여주는 정보는 아니다. 하지만, 버그bug 추적을 시작하기에 충분한 정보다.

메모리 매니저에 이런 설정들이 되어 있지 않으면, 여러분은 아무 오류도 볼 수 없을 것이다. 또한 프로그램은 계속 실행될 것이다 — 심지어 여러분이 무작위 버그들을 겪게 될 수 있다고 해도 그렇다. 즉, 다른 것들이 저장된 메모리 영역에 버퍼 오버런이 영향을 주는 경우에 생기는 버그들이다. 그런 상황이 되면 좀 이상하고 추적하기가 매우 어려운 오류들이 발생한다.

예를 들어, 나는 오브젝트의 데이터의 앞부분을 부분적으로 덮어쓴 것을 본 적이 있다. 그 클래스 참조가 저장된 위치였다. 이 메모리 영역이 손상되면, 그 클래스는 정의되지 않음undefined 상태가 된다. 그리고 그 가상 함수에 대한 모든 호출은 충돌crash이 생긴다 — 이런 것들은 프로그램의 완전히 다른 영역에 있는 메모리 쓰기 명령과 관련되어 있다고 연결해보기가 매우 어렵다.

윈도우가 아닌 플랫폼에서의 메모리 관리 Memory Management on Platforms Other than Windows

오브젝트 파스칼 컴파일러에서 메모리 관리를 하는 법을 생각할 때, 다른 플랫폼에서도 모든 것이 제대로 되는지 확인할 수 있는 선택지를 생각할 필요가 있다. 이에 관해 진행하기 전에, 비 윈도우 플랫폼에서는 델파이가 FastMM4를 사용하지 않으므로 프로그램이 닫힐 때 메모리 누수가 있는지 확인하기 위해 ReportMemoryLeak-sOnShutdown 전역 플래그를 설정하는 것은 쓸모 없음을 아는 것이 중요하다. 유거나 운영체제가 강제로 메모리에 있는 앱을 닫는 것 외에 모바일 애플리케이션을 닫는 일반적인 방법이 없는 또 다른 이유가 있다.

macOS와 iOS 및 안드로이드Android 플랫폼에서 오브젝트 파스칼 RTL은 네이티브 libc 라이브러리에서 malloc과 free 함수를 직접 호출한다. 이 플랫폼에서 메모리 사용량을 감시하는 방법은 외부 플랫폼 툴에 의존하는 것이다. iOS(와 macOS)의 예를 들면 여러분은 애플Apple의 인스트루먼트즈 도구Instruments tool을 통해 물리 장치에서 작동하는 여러분의 애플리케이션의 모든 측면을 감시하는 완전한 추적 시스템을 쓸 수 있다.

클래스별로 할당을 추적하기 Tracking Per-Class Allocations

끝으로, 특정 클래스를 추적하는 오브젝트 파스칼식 방법이 있다 (메모리 관리 전반에 관한 내용은 아님). 오브젝트에 대한 메모리 할당은 `NewInstance` 가상 클래스 메서드 호출을 통해 실현된다. 그리고 메모리 정리는 `FreeInstance` 가상 메서드가 수행한다. 이것들은 가상 메서드다. 따라서 여러분은 주어진 클래스가 특정 메모리 할당 전략에 대해 맞춤 정의를 하도록 오버라이드 `override`를 통해 정의할 수 있다.

장점은 생성자(이것은 여러 개일 수도 있다)와 소멸자에 상관없이, 메모리 추적 코드를 오브젝트의 표준 초기화 및 종료화 코드로부터 깔끔하게 분리할 수 있다는 것이다.

매우 희귀한 경우지만 (아마 매우 큰 메모리 구조에서만 쓸모가 있겠지만). 여러분은 이 메서드들을 오버라이드 해서, 주어진 클래스의 오브젝트들이 생성하고 소멸하는 수를 세고, 활성 상태의 인스턴스 수를 계산하고, 최종적으로 그 숫자가 0이 되는지 확인할 수 있다.

견고한 애플리케이션 작성하기 Writing Robust Applications

이 장에서, 그리고 이전 많은 장의 이 소단원에서, 견고한 애플리케이션 작성하기와 메모리 할당 및 해제를 올바르게 관리하기에 초점을 맞춘 몇 가지 기법들을 보았다.

이 장은 메모리 관리에 집중했는데, 이제 그 마지막 소단원이다. 여기에서는 조금 더 수준 높은 몇 가지 주제들을 나열하고자 한다. 앞에서 다뤘던 내용에서 조금 더 뺀어 나가는 것들이다. `try-finally` 블록을 사용하기, 소멸자를 호출하기 등은 이미 앞에서 다뤘다. 하지만, 여기서 강조해서 보여주려는 상황은 조금 더 복잡하다. 그리고 언어 요소들 여러 가지를 함께 사용하게 된다.

이 소단원은 그 수준이 그다지 높지 않다. 하지만, 오브젝트 파스칼 개발자가 완전하게 익혀야 하는 내용이다. 그래야 견고한 애플리케이션을 작성할 수 있다. 이 소단원에서 더 수준 높은 주제라고 분명히 말할 수 있는 부분은 맨 뒤에 있는 포인터와 오브젝트 참조 부분뿐이다. 오브젝트와 클래스 참조의 메모리 내부 구조를 다루기 때문이다.

생성자, 소멸자, 그리고 예외 Constructors, Destructors, and Exceptions

생성자와 소멸자는 정확한 메모리 관리를 위한 강력한 도구이지만, 잘못 사용할 경우 문제의 근원이 될 수 있다. 가상 생성자는 거의 항상 부모의 생성자를 *먼저* (`inherit` 호출을 통해) 호출해야 한다. 소멸자는 일반적으로 부모의 소멸자를 *마지막에* (역시 `inherit` 호출을 통해) 호출한다.

참고 좋은 코딩법을 따르려면, 필수 요소가 아니고 추가적인 호출이 소용없더라도 (`TObject.Create`를 상속할 때 같이) 일반적으로 기반 클래스 생성자를 여러분의 오브젝트 파스칼 코드의 모든 생성자에서 호출해야 한다.

이 소단원에서는 다음 고전적인 상황에서 생성자가 실패할 경우 무엇이 일어나는지 특별히 집중하겠다:

```
MyObj := TMyClass.Create;
try
  MyObj.DoSomething;
finally
  MyObj.Free;
end;
```

만일 오브젝트가 생성되고 MyObj 변수에 대입되면, finally 블록이 오브젝트 소멸을 담당한다. 그러나 만일 Create 호출이 예외를 일으키면, 정확한 대응으로 try-finally 블록에 진입하지 않는다. 생성자가 예외를 일으키면, 그에 대응하는 소멸자 코드가 부분적으로 초기화된 오브젝트에 대해 자동으로 실행된다. 예를 들어 생성자가 두 하위 오브젝트를 생성하면, 그에 맞는 소멸자를 호출하여 정리되어야 한다. 그러나, 이것은 여러분이 오브젝트가 적절히 초기화되었다고 단정할 경우 소멸자에서 잠재적인 문제를 일으킬 수 있다.

이것은 이론적으로 이해하기 간단하지 않으므로, 실제 코드 예제를 보겠다. SafeCode 예제는 일반적으로 - 생성자가 스스로 실패하지 않는 경우 - 맞는 생성자와 소멸자가 있는 클래스를 가지고 있다:

```
type
  TUnsafeDestructor = class
  private
    FList: TList;
  public
    constructor Create(PositiveNumber: Integer);
    destructor Destroy; override;
  end;

constructor TUnsafeDestructor.Create(PositiveNumber: Integer);
begin
  inherited Create;

  if PositiveNumber <= 0 then
    raise Exception.Create('Not a positive number');
  FList := TList.Create;
end;

destructor TUnsafeDestructor.Destroy;
begin
  FList.Clear;
  FList.Free;
  inherited;
end;
```

문제는 오브젝트가 완전히 생성되었을 때 있지 않고, 생성자가 음수 값을 만나 FList 필드가 여전히 nil로 설정되어 있을 때 발생한다. 이 경우, 생성자는 실패하고, 소멸자가 발동해 생성자가 호출되기 전에 nil로 된 FList에 대한 Clear 호출을 시도한다. nil에 대한 Clear의 접근 시도는 "접근 위반"[access violation](#) 예외를 일으킨다.

위와 같은 코드를 안전하게 작성하는 방법은 다음과 같다:

```
destructor TUnsafeDestructor.Destroy;
begin
  if Assigned(FList) then
    FList.Clear;
    FList.Free;
  inherited;
end;
```

다시 말하지만 이 이야기의 교훈은, 소멸자 안에서, 그 소멸자에 대응하는 생성자가 당연히 그 오브젝트를 완벽하게 초기화했다고 가정하면 안 된다는 것이다. 여러분은 이런 가정을 다른 메서드에서는 할 수 있다. 그렇지만, 소멸자에서는 그러면 안 된다.

중첩된 Finally 블록들 Nested Finally Blocks

Finally 블록은 여러분의 프로그램을 안전하게 만드는 가장 중요하고 흔한 기법일 것이다. 고급 주제는 아니다. 하지만, finally를 여러분이 온 사방에서 쓰고 있지는 않는가? 중첩된 연산 등 특수한 상황에 맞게 쓰고 있는가? 종료화 finalization 문장 여러 개를 모아 하나의 finally 블록 안에 넣지는 않는가? 아래는 완벽한 코드와 한참 동떨어진 코드다:

```
procedure TForm1.BtnTryFClick(Sender: TObject);
var
  A1, A2: TAClass;
begin
  A1 := TAClass.Create;
  A2 := TAClass.Create;
  try
    A1.Whatever := 'One';
    A2.Whatever := 'Two';
  finally
    A2.Free;
    A1.Free;
  end;
end;
```

아래는 같은 코드의 더 안전하고 정확한 버전이다(SafeCode 예제에서 다시 발췌함):

```
procedure TForm1.BtnTryFClick(Sender: TObject);
var
  A1, A2: TAClass;
begin
  A1 := TAClass.Create;
  try
    A2 := TAClass.Create;
    try
      A1.Whatever := 'One';
      A2.Whatever := 'Two';
    finally
      A2.Free;
    end;
  finally
    A1.Free;
  end;
end;
```


동적으로 타입을 확인하기 Dynamic Type Checking

일반적으로 타입들 사이의 동적 캐스팅 연산은, 특히 클래스 타입인 경우에는, 또다른 함정의 근원이 될 수 있다. 특히 `is`와 `as` 연산을 쓰지 않고 하드 캐스트 hard cast를 하는 경우가 그렇다. 직접적인 타입 캐스트는 모두, 사실, 오류의 잠재적인 근원이다(미리 `is`로 확인한 다음에 하지 않는다면 말이다).

오브젝트를 포인터로, 클래스 참조를 또는 클래스 참조로, 오브젝트를 인터페이스로, 문자열을 또는 문자열로 하는 모든 타입 캐스트는 잠재적으로 매우 위험하다. 하지만, 몇몇 특별한 환경에서는 피하기 어려운 문제다. 예를 들어, 여러분은 컴포넌트의 정수 값인 `Tag` 프로퍼티의 오브젝트 참조를 저장하고 싶을 것이며, 이 경우 하드 캐스트를 사용하게 된다. 다른 경우로 (다음 장에서 다룰 타입-안전 type-safe한 일반 리스트가 아닌) 구식의 `TList`를 사용한 포인터 리스트에 오브젝트를 저장할 때가 있다.

아래는 상당히 바보 같은 예시다:

```
procedure TForm1.BtnCastClick(Sender: TObject);
var
    List: TList;
begin
    List := TList.Create;
    try
        List.Add(Pointer(Sender));
        List.Add(Pointer(23422));
        // 직접 캐스트
        TButton(List[0]).Caption := 'Ouch';
        TButton(List[1]).Caption := 'Ouch';
    finally
        List.Free;
    end;
end;
```

이 코드를 실행하면 일반적으로 접근 위반을 일으킨다.

참고 여기 *일반적*이라고 쓴 것은 메모리에 임의로 접근하기 때문에 실제 어떤 효과가 있을지 모르기 때문이다. 프로그램이 즉시 오류를 내지 않고 단순히 메모리를 덮어쓰는 경우 다른 데이터가 손상된 이유를 찾아내는 고된 시간을 보내야 한다.

여러분은 유사한 상황을 가능한 한 피해야 하지만, 코드를 고치지 않으면 방법이 없는 경우에는 어떻게 해야 하는가? 자연스러운 접근법은 다음 코드 조각과 같이 `as`로 안전한 캐스트를 하거나 `is`로 타입 검사를 사용하는 것이다:

```
// "as" 캐스트
(TObject(List[0]) as TButton).Caption := 'Ouch';
(TObject(List[1]) as TButton).Caption := 'Ouch';

// "is" 캐스트
if TObject(List[0]) is TButton then
    TButton(List[0]).Caption := 'Ouch';
if TObject(List[1]) is TButton then
    TButton(List[1]).Caption := 'Ouch';
```


그러나, 이것은 해답이 *아니다*. 여러분은 계속 접근 위반을 얻는다. 문제는 is와 as 모두 TObject.InheritsFrom를 호출하는데, 숫자에 대해서 적용하기 어려운 동작이다!

해법은 무엇일까? 진짜 답은 처음부터 (코드의 타입이 잘 맞지 않는) 유사한 상황을 피하여, 예를 들어 TObjectList나 다른 안전한 기법을 쓰는 것이다(일반 컨테이너 클래스들을 다음 장에서 다룰 것이다). 만일 저수준의 꼼수^{hack}을 부리며 포인터를 가지고 놀고자 한다면, 주어진 "숫자값"이 오브젝트의 참조인지 아닌지 판단할 수 있다. 그러나 이것은 사소한 동작인 아니다. 여기 흥미로운 측면이 있는데, 다음 예제로 오브젝트와 클래스 참조의 내부 구조를 설명하는데 양해를 구하고자 한다.

이 포인터는 오브젝트 참조인가? *Is this Pointer an Object Reference?*

이 소단원은 오브젝트와 클래스 참조의 내부구조를 설명하며, 이 책의 대부분에서 다루는 주제의 수준을 넘어선다. 그래도, 전문가인 독자들에게는 더 많은 내용과 통찰력을 제공해 줄 것이다. 그래서, 내가 예전에 전문지에 기고했던 메모리 관리 부분을 여기에 남기기로 했다. 다만, 아래 구현은 오직 윈도우만을 위한 메모리 검사라는 점을 알아두기 바란다.

예전에, 포인터를 여기저기에서 사용해야 했던 때가 있었다 (포인터는 그저 숫자 값이며, 데이터가 실제로 있는 물리적인 메모리 위치를 가리킨다). 이 포인터들은 실제로 오브젝트에 대한 참조일 수 있다. 대체로 우리는 그것들이 있다는 것을 알고 사용한다. 하지만, 저수준의 캐스트^{cast/변환}를 한다면 그때마다 우리는 프로그램 전체를 망가뜨릴 위기를 마주하게 된다. 이런 종류의 포인터 관리를 조금 더 안전하게 할 수 있는 기법들이 있다. 그 기법들이 비록 100퍼센트 안전을 보장하지는 않지만 말이다.

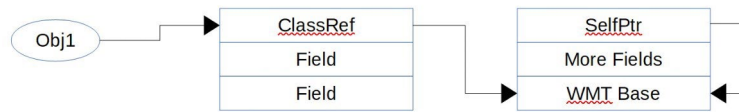
포인터를 다루기 전에 고려해야 하는 첫 시작은 그 포인터가 유효한 포인터인지 아닌지를 확인하는 것이다. Assigned 함수는 그저 포인터가 nil이 아니라는 점만 점검한다. 따라서, 이 경우에는 도움이 되지 않는다. 그러나, FindHInstance 함수는 파라미터로 오브젝트를 전달받으면 그것이 담긴 힙 블록의 시작 주소^{base address}를 반환한다. 만약 0을 반환한다면, 그 포인터는 유효하지 않은 페이지^{page}를 가리키고 있을 것이다(자주 있지는 않지만 이 경우 메모리 페이지 오류 추적이 매우 어렵다. 따라서 이런 상황을 미리 방지하는데 도움이 된다). 만약 반환하는 숫자가 거의 임의의 숫자라면, 아마도 그것이 가리키고 있는 것이 유효하지 않은 메모리 페이지일 가능성이 크다. 참고로, FindHInstance 함수는 오브젝트 파스칼 RTL 안에 있다(System 유닛 안에 있으며, 윈도우 플랫폼에서 사용 가능).

이렇게 시작하면 좋다. 하지만, 더 잘 할 수 있다. 왜냐하면 위 기법은 포인터의 값이 문자열 참조인지 또는 다른 유효한 포인터이지만 오브젝트 참조는 아닌지 등을 파악하는데는 도움이 되지 않기 때문이다. 포인터가 오브젝트의 참조임을 어떻게 알 수 있을까? 다음과 같이 경험적으로 테스트를 해보자. 오브젝트의 첫 4바이트는 그 클래스에 대한 포인터다. 그 클래스 참조의 내부 데이터 구조를 보자. vmtSelfPtr 위치 안에

자기 자신을 가리키는 포인터를 담는다. 대략 그림 13.7의 그림과 같다.

그림 13.7:

오브젝트와 클래스
참조의 내부 구조에
대한 대략적인 표현



즉, 클래스 참조 포인터로부터 `vmtSelfPtr`만큼의 바이트의 메모리 위치 값(메모리 하위 방향 음수 오프셋^{offset}인)을 역참조하면, 같은 클래스 참조 포인터를 다시 찾을 수 있다. 또한, 인스턴스 크기 정보를 (`vmtInstanceSize` 위치에서 있다) 읽을 수 있다. 그래서 그곳에 **합리적인** 숫자가 있는지 확인할 수 있다.

여기 실제 코드가 있다:

```
function IsPointerToObject(Address: Pointer): Boolean;
var
  ClassPointer, VmtPointer: PByte;
  InstSize: Integer;
begin
  Result := False;
  if FindHInstance(Address) > 0 then
  begin
    VmtPointer := PByte(Address^);
    ClassPointer := VmtPointer + vmtSelfPtr;
    if Assigned(VmtPointer) and
      (FindHInstance(VmtPointer) > 0) then
    begin
      InstSize := (PInteger(
        VmtPointer + VmtInstanceSize))^;
      // Self 포인터를 확인한다. 그리고 인스턴스 크기가 "합리적인지" 확인한다
      if Pointer(Pointer(ClassPointer)^) =
        Pointer(VmtPointer) and
        (InstSize > 0) and (InstSize < 10000) then
        Result := True;
    end;
  end;
end;
```

참고 매우 높은 확률로 이 함수는 올바른 값을 반환한다. 하지만, 100% 확실하다고 보장하지 못한다. 아쉽지만, 메모리 안에 있는 임의의 값이 이 검사를 통과할 수도 있다.

위 함수가 있다면, 이전의 `SafeCode` 예제로 돌아가서, 우리는 안전한 캐스트를 하기 전에 그 포인터가 오브젝트를 가리키고 있는지를 미리 검사할 수 있다.

```
if IsPointerToObject(List[0]) then
  (TObject(List[0]) as TButton).Caption := 'Ouch';
if IsPointerToObject(List[1]) then
  (TObject(List[1]) as TButton).Caption := 'Ouch';
```

똑같은 방법을 사용해 클래스 참조 사이에 안전한 캐스트를 구현할 수 있다. 다시 말

하지만, 이런 문제 상황을 애초에 만들지 않기 위해 보다 안전하고 보다 깔끔한 코드를 작성하는 것이 가장 좋다. 하지만, 혹시라도 피할 수 없는 경우를 대비한다면, `IsPointerToObject` 함수를 쓰면 편할 것이다. 어쨌든, 이 소단원의 내용은 이 시스템 데이터 구조들의 내부에 대해 일부만 설명한 것임을 알아두자.

파트 III 고급 기능들

이제 우리는 언어의 기초와 객체 지향 프로그래밍 패러다임을 보았으므로, 오브젝트 파스칼 언어의 몇몇 최신 고급 기능들을 살펴볼 차례다. 제네릭 [Generics](#), 익명 메서드 [anonymous method](#), 리플렉션 [reflection](#) 이 객체 지향 프로그래밍을 중요한 방법으로 확장하는 새로운 패러다임으로 코드를 개발하고자 하는 이에게 열려 있다.

사실, 이들 고급 언어 기능 중 일부는, 개발자에게 코드를 작성하는 새로운 방법을 받아들이고, 몇몇 타입과 코드 추상화를 제공하며, 코딩에 관한 보다 동적인 접근법을 준다.

이 단원의 마지막 부분은 이들 언어 기능들을 핵심 런타임 라이브러리 요소의 개요를 보면서 확장할 것이다. 그리고 오브젝트 파스칼 개발 모델의 핵심 중의 핵심이라 언어와 라이브러리의 구분을 어렵게 하는 요소를 볼 것이다. 예를 들어, 우리는 이전에 보았던 `TObject` 클래스를 살펴보며 그것이 여러분이 작성하는 모든 클래스의 기반 클래스이며, 라이브러리 구현의 세부 사항으로 역할을 한정하기보다 더 중요한 역할을 가지는 것을 볼 것이다.

파트 III 요약

14장: 제네릭스 [Generics](#)

15장: 익명 메서드 [Anonymous Methods](#)

16장: 리플렉션과 애트리뷰트 [Reflection and Attributes](#)

17장: TObject 클래스 [The TObject Class](#)

18 장: 런타임 라이브러리 [The Runtime Library](#)

14: 제네릭스 Generics

오브젝트 파스칼의 엄격한^{strong} 타입 검사^{type checking}는 코드의 정확성을 높여준다. 이미 이 책에서 여러차례 강조한 사항이다. 그런데, 엄격한 타입 검사가 성가실 때도 있다. 프로시저(또는 클래스)가 그 동작이 비슷하면서, 다루는 데이터 타입만 다른 경우에, 그런 프로시저들을 새로 작성하려면 매우 귀찮다. 이런 문제를 해소할 방법이 있다. 오브젝트 파스칼 언어에 있는 기능이며 C#이나 자바 등 다른 유사한 언어에도 있는 방법이다. 바로 제네릭^{generics}을 사용하는 것이다.

제네릭, 혹은 템플릿 클래스^{template class}라는 개념은 사실 C++ 언어로부터 도입되었다. 아래는 1994년에 내가 C++에 대해 쓴 책에 있는 내용이다:

여러분은 클래스 선언에서 (하나 혹은 그 이상의) 데이터 멤버에 대한 타입을 명시하지 않고 생략하는 것이 가능하다: 그 동작은 그 클래스의 오브젝트가 실제로 선언될 때까지 미뤄질 수 있다. 이와 마찬가지로, 함수를 정의할 때 (하나 혹은 그 이상의) 파라미터에 대한 타입을 명시하지 않고 생략하는 것도 가능하다. 즉, 그 함수가 호출되기 전까지 미뤄질 수 있다.

참고 위 내용은 90년대 초에 Steve Tendon와 함께 저술한 “볼랜드 C++ 4.0 객체 지향 프로그래밍” (Borland C++ 4.0 Object-Oriented Programming) 책에 있다.

이 장에서는 이 주제를 살펴본다. 기초에서부터 시작해서, 몇 가지 고급 사용 사례를 다루면서, 제네릭이 어떻게 표준 비주얼 프로그래밍에 적용될 수 있는지를 알려준다.

제네릭 키-값 쌍 Generic Key-Value Pairs

제네릭 클래스 [generic class](#), 일반 클래스 첫 예제로, 키-값 쌍 데이터 구조를 보자. 아래는 전통적인 방식으로 작성된 데이터 구조다. 이 오브젝트는 값을 담는데 사용된다.

```
type
  TKeyValue = class
  private
    FKey: string;
    FValue: TObject;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: TObject);
  public
    property Key: string read FKey write SetKey;
    property Value: TObject read FValue write SetValue;
  end;
```

위 클래스를 사용하는 코드는 아래와 같다. 오브젝트를 만들고, 키와 값을 설정하고, 사용한다. (KeyValueClassic 예제의 메인 폼에 있는 다양한 메서드들 중에서 발췌).

```
// FormCreate
Kv := TKeyValue.Create;

// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender; // 이 버튼(Button1)

// Button2Click
Kv.Value := Self; // 이 폼(form)

// Button3Click
ShowMessage([' ' + Kv.Key + ', ' + Kv.Value.ClassName + '']);
```

만약 위와 비슷한 클래스가 필요한데 값에는 오브젝트 대신 Integer를 담아야 한다면 어떻게 할까? 아마도 매우(위험하고) 부자연스러운 타입 캐스트를 하거나, 새 클래스를 더 만들어 키에는 문자열을 값에는 숫자를 담는 방법 중에서 선택할 수도 있을 것이다. 원래 있던 클래스를 복사/붙여넣기 [copy-and-paste](#) 해서 해결하면 된다고 생각할 수도 있을 것이다. 하지만, 그렇게 되면 결국 코드가 기본적으로 같은 '두 개의 코드 복사본'을 갖게 된다. 이것은 좋은 프로그래밍 원리에 어긋난다. 또한 유지보수 [maintenance](#)의 악몽에 시달리게 될 것이다. 새 기능을 추가하거나 버그를 고칠 때마다 둘, 셋 혹은 스무 개의 거의 똑 같은 복사본 하나하나를 모두 수정해야 하기 때문이다.

제네릭을 사용하면, 이 값에 대해 더 폭넓은 정의를 할 수 있다. 제네릭 [generic/구체적이지 않은](#) 클래스 하나만 작성하면 된다. 그 키-값 제네릭 클래스를 여러분이 인스턴스화 [instantiate](#) 하면 바로 그 때서야, 주어진 데이터 타입에 묶이는 구체적인 클래스가 된다. 따라서, 컴파일 되어 여러분의 애플리케이션 안에 들어가는 클래스는 둘, 셋, 혹은 스무 개가 될 수 있다. 그러나 여러분이 작성하는 소스 코드 정의는 단 하나다. 그 정의로부터 만들어지는 클래스들은 모두 저마다 타입 검사를 네이티브 하게 한다. 게다가 런타임에 추가 오버헤드 [overhead](#)를 주지 않는다.

설명이 너무 앞서 나간 것 같다. 이제 위 코드 즉 키-값 쌍을 담는 제네릭 클래스를 정의하는 구문에서부터 시작해보자.

```
type
  TKeyValue<T> = class
  private
    FKey: string;
    FValue: T;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: T);
  public
    property Key: string read FKey write SetKey;
    property Value: T read FValue write SetValue;
  end;
```

위 첫 줄의 클래스 정의를 보면, 명시되지 않은 `unspecified` 타입이 하나 있다. 꺾쇠 괄호 `angle brackets`, `<>`로 감싸진 자리 표시자 `T`가 그것이다. 대체로 약속처럼 심볼 `T`가 사용된다. 하지만, 컴파일러는 여러분이 어떤 심볼을 사용하든 다 인식한다. 그렇지만, `T`를 사용하면 코드를 읽기가 더 쉽다. 제네릭 클래스가 파라미터 타입을 하나만 사용한다면 말이다. 구체적이지 않은 클래스 즉 제네릭 클래스에 필요한 파라미터 타입이 여러 개라면, 각자의 실제 역할에 맞춘 이름을 지정하자. 연속한 글자(예: `T`, `U`, `V`)를 쓰지는 말자. C++에서는 초창기에 그렇게 했지만 말이다.

참고 “T”가 제네릭 타입의 표준 이름, 혹은 자리 표시자가 된 것은 C++ 언어에 *템플릿*이 도입된 1990년대 초다. 작성자들에 의하면, “T”는 “타입” 또는 “템플릿 타입”을 뜻한다. 델파이 세계에서 이 관습은 통한다. 타입 이름에 `T` 접두사를 붙이는 것이 일반적이므로, “T”라고 하면 “Type”이라고 알아듣기 때문이다.

위 `TKeyValue<T>` 제네릭 클래스는 이 명시되지 않은 타입을 두 필드 중 하나의 값에 사용한다. 또한 그 필드를 가리키는 프로퍼티 값과 세터 `setter` 메서드의 파라미터로도 사용한다. 세터 메서드에 대한 정의는 평소와 같다. 제네릭 타입이든 아니든 상관없이 메서드 정의에는 그 클래스의 전체 이름 즉 제네릭 타입까지 명시된 이름을 적어야 한다는 점을 명심하자:

```
procedure TKeyValue<T>.SetKey(const Value: string);
begin
  FKey := Value;
end;

procedure TKeyValue<T>.SetValue(const Value: T);
begin
  FValue := Value;
end;
```

이 클래스를 사용하려면, 여러분은 그 클래스가 충분히 자격을 갖추도록 해야 한다. 즉, 제네릭 타입의 실제 타입을 제공해야 한다. 예를 들어, 다음과 같이 작성하면, 그 키-값 오브젝트에는 값에 버튼을 담는다고 선언하는 것이다:


```
var
    Kv: TKeyValue<TButton>;
```

인스턴스를 생성할 때에도 이와 같이 완전한 이름이 필요하다. 이때의 이름이 실제 타입 이름이 되기 때문이다 (이와 달리, 제네릭 타입 이름, 즉 인스턴스화 되기 전의 타입 이름,은 타입을 생성하는 하나의 메커니즘 같은 것이라고 보면 된다).

키-값 쌍의 값에 타입을 명시하여 사용하면, 코드는 훨씬 더 견고해진다. 이제 여러분은 TButton (또는 그 자손의) 오브젝트를 이 키-값 쌍에 넣을 수 있다. 그리고 그렇게 뽑아낸 오브젝트에 속한 다양한 메서드들과 프로퍼티들에 우리는 접근할 수 있다.

아래 코드 조각을 보자. KeyValueGeneric 예제의 메인 폼에서 가져온 것이다:

```
// FormCreate
Kv := TKeyValue<TButton>.Create;

// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender as TButton;

// Button2Click
Kv.Value := Sender as TButton; // 원래는 "Self"였는데, 이제는 유효하지 않다!

// Button3Click
ShowMessage ( '[' + Kv.Key + ', ' + Kv.Value.Name + ' ]');
```

이전 버전의 코드에서는 우리가 제네릭 오브젝트를 할당할 때 버튼이든 폼이든 뭐든 넣을 수 있었다. 하지만, 이제는 오직 버튼만 넣을 수 있도록 컴파일러가 강제한다. 이와 마찬가지로, 앞 버전에서는 출력할 때 제네릭한 Kv.Value.ClassName을 사용했지만, 이제는 이 (버튼) 컴포넌트의 Name프로퍼티를 사용할 수 있다. 즉, TButton 클래스에 속한 모든 프로퍼티들을 사용할 수 있다.

이전 버전의 프로그램처럼 만들려면, 이 키-값 선언을 오브젝트 타입으로 하면 된다:

```
var
    Kvo: TKeyValue<TObject>;
```

제네릭 키-값 쌍이 위와 같으면, 그 값에는 어떤 오브젝트든 넣을 수 있다. 하지만, 이것을 사용해서 뽑아낸 클래스를 가지고 우리가 할 수 있는 것은 많지 않다. 보다 구체적인 타입으로 캐스트 하지 않는다면 말이다. 버튼과 오브젝트의 중간에서 균형을 잘 잡아주는 뭔가를 원한다면, 여러분은 그 값을 TComponent를 컴포넌트 타입으로 명시하면 된다:

```
var
    Kvc: TKeyValue<TComponent>;
```

위에 해당하는 코드 조각들 역시 KeyValueGeneric 예제에 있다. 끝으로, 이 제네릭 키-값 쌍 클래스의 인스턴스가 이제 오브젝트 값이 아니라 정수를 담도록 정의해보자.


```

var
  Kvi: TKeyValue<Integer>;
begin
  Kvi := TKeyValue<Integer>.Create;
  try
    Kvi.Key := 'Object';
    kvi.Value := 100;
    Kvi.Value := Left;
    ShowMessage ('[' + Kvi.Key + ', ' +
      IntToStr (Kvi.Value) + ']');
  finally
    Kvi.Free;
  end;

```

인라인 변수와 제네릭 타입 추론 Inline Variables and Generics Type Inference

제네릭 타입의 변수를 선언할 때, 여러분이 작성해야 하는 선언문이 상당히 길 수도 있다. 해당 타입의 오브젝트를 생성할 때마다, 똑같은 선언을 반복해야 하기 때문이다. 하지만, 인라인 변수 [inline variable](#) 선언과 그것의 변수 타입 추론 능력을 사용한다면 한결 편하다. 위 코드는 다음처럼 쓸 수도 있다.

```

begin
  var Kvi := TKeyValue<Integer>.Create;
  try
    ...

```

이 코드에서 여러분은 제네릭 타입의 선언 전체를 두 번 적지 않아도 된다. 이것은 컨테이너를 다룰 때 특히 편하다. 뒤에서 살펴보게 될 것이다.

제네릭의 타입 규칙 Type Rules on Generics

여러분이 제네릭 타입의 인스턴스를 선언하는 시점에, 그 타입은 특정 버전을 가지게 된다. 이는 컴파일러에 의해 강제되며 이후 모든 동작에 적용된다. 따라서, 예를 들어 다음 제네릭 클래스가 있다면:

```

type
  TSimpleGeneric<T> = class
    Value: T;
  end;

```

이제 여러분이 특정 오브젝트를 특정 타입으로 선언하기 때문에, Value 필드에 다른 타입을 대입하지 못한다. 즉, 아래의 두 오브젝트에서, 대입들 중 몇 개가 잘못되었다 (TypeCompRules 예시에서 발췌함):

```

var
  Sg1: TSimpleGeneric<string>;
  Sg2: TSimpleGeneric<Integer>;
begin
  Sg1 := TSimpleGeneric<string>.Create;
  Sg2 := TSimpleGeneric<Integer>.Create;

```



```
Sg1.Value := 'Foo';
Sg1.Value := 10; // 오류
// E2010 Incompatible types: 'string' and 'Integer'

Sg2.Value := 'Foo'; // 오류
// E2010 Incompatible types: 'Integer' and 'string'
Sg2.Value := 10;
```

일단 제네릭 선언에서 특정 타입을 여러분이 정의하면, 그 타입은 컴파일러에 의해 고정된다. 이는 오브젝트 파스칼 같은 강한 타입 [strong-typed](#) 언어라면 당연히 예상되는 결과다. 타입 검사 역시 모든 제네릭 오브젝트에 대해 적용된다. 오브젝트를 생성할 때 제네릭 파라미터를 명시한 경우, 여러분은 유사한 제네릭 타입일지라도 타입 호환이 되지 않는 인스턴스에 그 오브젝트를 대입할 수 없다. 헷갈린다면, 아래 예제를 보면 명확히 이해할 수 있을 것이다.

```
Sg1 := TSimpleGeneric<Integer>.Create; // 오류
// E2010 Incompatible types:
// 'TSimpleGeneric<System.string>'
// and 'TSimpleGeneric<System.Integer>'
```

“제네릭 타입 호환성 규칙” 소단원에서 다루겠지만, 이 경우는 특이하다. 타입 호환성 규칙은 타입 이름이 아닌 그 구조에 의해 결정된다. 일단 제네릭 타입이 구체적으로 선언되고 나면, 그와 호환되지 않는 타입을 거기에 대입할 수 없다.

오브젝트 파스칼의 제네릭 [Generics in Object Pascal](#)

이전 예제에서 우리는 오브젝트 파스칼로 어떻게 제네릭 클래스를 정의하고 사용하는지 보았다. 이 기능에 대해 예제와 함께 소개를 했으니 이제 그 기술적인 측면으로 들어가보자. 매우 복잡하면서도 동시에 매우 중요한 내용이다. 제네릭을 언어 면에서 살펴보고 나서, 보다 많은 예제들을 보기로 하자. 그 예제들 중에는 제네릭 컨테이너 클래스를 사용하고 정의하는 예제도 있다. 제네릭 컨테이너는 이 언어에서 제네릭이 주로 사용되는 영역이기도 하다.

앞에서 여러분은 클래스를 정의할 때 추가 “파라미터”를 꺾쇠 괄호 안에 넣어서 그 타입이 나중에 제공될 수 있도록 한다는 것을 보았다:

```
type
  TMyClass<T> = class
  end;
```

전달되는 제네릭 타입은 필드의 타입으로(앞의 예제에서 봤다), 프로퍼티의 타입으로, 함수의 파라미터 혹은 반환 값의 타입 등등으로 사용된다. 이 타입들을 반드시 로컬에서만 필드(또는 배열)로 사용해야 하는 것은 아니다. 제네릭 타입을 그저 결과값이

나 파라미터로 사용하는 경우도 있다. 즉 클래스의 선언이 아니라, 클래스 안에 있는 메서드의 정의 안에서만 써도 된다.

이런 확장된 즉 *제네릭* 타입 선언 형태는 클래스뿐만 아니라 레코드에서도 사용할 수 있다(5장에서 봤듯이, 레코드도 메서드, 프로퍼티, 오버로드된 연산자를 가질 수 있다). 제네릭 클래스는 파라미터화 *parameterized* 된 타입을 여러 개 가질 수도 있다. 아래 클래스는 메서드의 입력 파라미터와 결과 값을 서로 다른 타입으로 명시할 수 있도록 한다.

```
type
TPWGeneric<TInput, TReturn> = class
public
  function AnyFunction(Value: TInput): TReturn;
end;
```

다른 정적 언어 *static language*들이 그렇듯, 오브젝트 파스칼의 제네릭 구현은 런타임 지원에 기반하지 않는다. 이것을 다루는 것은 컴파일러와 링커다. 따라서 런타임 메커니즘에 거의 아무것도 남기지 않는다. 가상 함수 호출과는 다르다. 그것은 런타임에 묶인다. 하지만, 제네릭 메서드는 여러분이 제네릭 타입을 인스턴스화하는 그 시점에 단 한 번만 생성된다. 컴파일 중에 말이다! 우리는 이 방식에 따른 단점들을 보게 될 것이다. 하지만, 장점을 보자면 제네릭 클래스들은 평범한 클래스만큼 효율적이다. 심지어 더 효율적이다. 런타임 검사의 필요성이 줄기 때문이다. 그 내부를 살펴보기 전에 먼저, 파스칼 언어의 전통적인 타입 호환성 규칙을 벗어난 몇 가지 매우 중요한 규칙을 보자.

제네릭 타입 호환성 규칙 Generic Types Compatibility Rules

전통적인 파스칼과 오브젝트 파스칼 언어에서, 핵심 타입 호환성 규칙은 타입 이름 동치 *type name equivalence*에 기반한다. 즉, 두 변수가 타입 호환 *type compatible*이 되려면, 오로지 타입 이름이 똑같아야 한다. 그것들이 실제로 가리키는 데이터 구조와는 상관이 없다.

아래는 타입 호환이 안되는 것을 보여주는 고전적인 예시로써 정적 배열을 사용해서 보여준다 (TypeCompRules 예제에서 발췌함).

```
type
TArrayOf10 = array[1..10] of Integer;

procedure TForm30.Button1Click(Sender: TObject);
var
  Array1: TArrayOf10;
  Array2: TArrayOf10;
  Array3, Array4: array[1..10] of Integer;
begin
  Array1 := Array2;
  Array2 := Array3; // 오류
  // E2010 Incompatible types: 'TArrayOf10' and 'Array'

  Array3 := Array4;
  Array4 := Array1; // 오류
  // E2010 Incompatible types: 'Array' and 'TArrayOf10'
end;
```


위 코드에 있는 배열 네 개는 모두 구조가 똑같다. 하지만 컴파일러는 타입 호환이 되는 변수에만 대입을 허용한다. 즉, 명시적 타입 이름이 똑같거나(예: TArrayOf10) 또는 (컴파일러가 생성하는) 암시적 타입 이름이 똑같아야 대입할 수 있다.

이 타입 호환 규칙에 예외는 거의 없다. 파생 클래스에 속한 것들 정도뿐이다. 그런데 이 규칙에는 매우 중요한 예외 하나가 더 있다. 바로 제네릭 타입에 대한 타입 호환이다. 이 예외 규칙은 내부적으로 컴파일러에서도 사용된다. 즉 제네릭 타입으로부터 새 타입(과 그것이 가진 모든 메서드들)을 언제 생성하는지를 결정하는데도 사용된다.

이 제네릭 타입에 대한 새 호환 규칙은 다음과 같다. 제네릭 타입들 중에 그 클래스 정의가 똑같고 인스턴스 타입도 똑같다면, 그 제네릭 타입들은 서로 호환된다. 그리고 제네릭 클래스 정의에 사용된 타입 이름은 호환과 전혀 상관이 없다. 다르게 설명하면, 제네릭 타입 인스턴스의 전체 이름은 제네릭 타입과 인스턴스 타입을 합친 것이다.

다음 예시에 있는 변수 네 개는 모두 타입 호환된다:

```
type
  TGenericArray<T> = class
    AnArray: array[1..10] of T;
  end;
  TIntGenericArray = TGenericArray<Integer>;

procedure TForm30.Button2Click(Sender: TObject);
var
  Array1: TIntGenericArray;
  Array2: TIntGenericArray;
  Array3, Array4: TGenericArray<Integer>;
begin
  Array1 := TIntGenericArray.Create;
  Array2 := Array1;
  Array3 := Array2;
  Array4 := Array3;
  Array1 := Array4;
end;
```

표준 클래스용 제네릭 메서드 Generic Methods for Standard Classes

제네릭 타입은 제네릭 클래스를 정의하는데 가장 많이 쓰인다. 하지만, 제네릭이 아닌 클래스 안에서도 사용될 수 있다. 즉, 일반 클래스들도 제네릭 메서드를 가질 수 있다. 이런 경우, 여러분이 그 클래스의 인스턴스를 만들 때는 그저 제네릭 자리 표시자를 명시하지 않고 그냥 두면 된다. 그 대신 그 제네릭 메서드를 불러낼 ^{invoke} 때 명시한다. 아래의 예시 클래스 안에는 제네릭 메서드가 있다 (GenericMethod 예제에서 발췌함).

```
type
  TGenericFunction = class
  public
    function WithParam<T>(T1: T): string;
  end;
```


참고 이 코드를 처음에 작성할 때, 나는 파라미터를 (t: T)와 같이 표기했었다. C++ 경험의 추억이 남아있어서 그랬던 것 같다. 두 말할 필요도 없이, 오브젝트 파스칼처럼 대소문자 구분이 없는 언어에서 그렇게 하려는 것은 좋은 생각이 아니다. 컴파일러는 그런 문장을 허용한다. 하지만, 제네릭 타입 T를 가리킬 때마다 오류를 낼 것이다.

이런 클래스 메서드 안에서 여러분이 할 수 있는 것은 많지 않다. (제약^{constraint}을 사용하지 않는다면 말이다. 제약은 이 장 뒷부분에서 설명한다), 그래서, 특별한 제네릭 타입 함수(역시, 뒤에서 설명한다)들 그리고 타입을 문자열로 바꿔주는 특별한 함수를 사용해서 코드를 만들었다:

```
function TGenericFunction.WithParam<T>(T1: T): string;
begin
    Result := GetTypeName(TypeInfo(T));
end;
```

위 메서드는 파라미터 안에 전달된 실제 값을 전혀 사용하지 않는다. 그저 파라미터의 타입 정보 몇 가지를 읽고 반환한다. 다시 말하지만, 전달되는 T1의 타입을 전혀 알지 못한 채, 그것을 사용하는 코드를 이 함수 안에서 작성하기란 꽤 복잡하다.

여러분은 이 “전역 제네릭 함수”의 다양한 버전들을 호출할 수 있다:

```
var
    GF: TGenericFunction;
begin
    GF := TGenericFunction.Create;
    try
        Show(GF.WithParam<string>('Foo'));
        Show(GF.WithParam<Integer>(122));
        Show(GF.WithParam('Hello'));
        Show(GF.WithParam(122));
        Show(GF.WithParam(Button1));
        Show(GF.WithParam<TObject>(Button1));
    finally
        GF.Free;
    end;
```

위의 호출은 모두 올바르다. 호출 시, 타입을 지정하는 파라미터 전달이 암묵적이어도 되기 때문이다. 모두 (명시된 대로, 또는 추론된 대로) 해당 제네릭 타입을 표현한다. 파라미터에 담겨 전달되는 값의 실제 타입을 표현하는 게 아니라는 점을 알아야 한다. 그러면 아래 결과를 이해할 수 있을 것이다.

```
String
Integer
string
ShortInt
TButton
TObject
```

타입을 꺾쇠 괄호 사이에 명시하지 않은 채 메서드를 호출하면, 파라미터 값에서 그 실제 타입을 추론한다. 하지만, 타입과 파라미터를 모두 명시해서 메서드를 호출하는

경우에는, 명시한 제네릭 타입 선언과 파라미터로 전달되는 값의 타입이 반드시 일치해야 한다. 따라서, 아래 세 줄은 컴파일이 되지 않는다:

```
Show(GF.WithParam<Integer>('Foo'));
Show(GF.WithParam<string>(122));
Show(GF.WithParam<TButton>(Self));
```

제네릭 타입의 인스턴스화 Generic Type Instantiation

이것은 한결 더 수준 높은 소단원이다. 제네릭의 일부 내부와 잠재적 최적화 optimization에 집중한다. 제네릭에 관해 두 번째로 읽을 때에 좋다. 처음 보는 독자에게는 적합하지 않다.

몇 가지 최적화에 의한 예외가 있긴 하지만, 여러분이 제네릭 타입을 인스턴스화할 때마다, 그것이 메서드든 클래스든, 컴파일러는 새 타입을 만들어낸다. 그 새 타입은 같은 제네릭 타입에서 나온 다른 인스턴스들(또는 같은 제네릭 메서드의 다른 버전들)과 코드를 전혀 공유하지 않는다.

예시를 보자(GenericCodeGen 예제에서 발췌함). 다음과 같은 제네릭 클래스가 있다:

```
type
  TSampleClass<T> = class
  private
    FData: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT(Value: T);
  end;
```

아래는 이 메서드들의 구현이다 (One 메서드는 제네릭 타입에서 완전히 독립적이라는 점을 눈 여겨 보자):

```
procedure TSampleClass<T>.One;
begin
  Form30.Show('OneT');
end;

function TSampleClass<T>.ReadT: T;
begin
  Result := FData;
end;

procedure TSampleClass<T>.SetT(Value: T);
begin
  FData := Value;
end;
```

이제 메인 프로그램은 제네릭 타입을 사용해 그 메서드의 메모리 내 주소를 알아낸다. 그 주소는 컴파일러에 의해 인스턴스가 생성될 때 확보된다. 여기 그 코드가 있다:


```

procedure TForm30.Button1Click(Sender: TObject);
var
  T1: TSampleClass<Integer>;  T2: TSampleClass<string>;
begin
  T1 := TSampleClass<Integer>.Create;
  T1.SetT(10);
  T1.One;

  T2 := TSampleClass<string>.Create;
  T2.SetT('Hello');
  T2.One;

  Show( 'T1.SetT: ' +
    IntToHex(PInteger(@TSampleClass<Integer>.SetT)^, 8));
  Show( 'T2.SetT: ' +
    IntToHex(PInteger(@TSampleClass<string>.SetT)^, 8));

  Show( 'T1.One: ' +
    IntToHex(PInteger(@TSampleClass<Integer>.One)^, 8));
  Show( 'T2.One: ' +
    IntToHex(PInteger(@TSampleClass<string>.One)^, 8));
end;

```

결과는 다음과 같다(실제 값은 환경에 따라 달라진다):

```

T1.SetT: C3045089
T2.SetT: 51EC8B55
T1.One: 4657F0BA
T2.One: 46581CBA

```

컴파일러가 각 데이터 타입을 위해 생성하는 메모리 위치를 보면, SetT 메서드만 다른 버전을 갖게 되는 게 아니라, One 메서드도 그렇다. 똑같은데도 불구하고 말이다.

게다가, 동일한 제네릭 타입을 다시 선언하면, 여러분은 또다시 새 함수들의 묶음을 얻게 된다. 이와 비슷한 이유로, 하나의 제네릭 타입에서 나온 똑같은 인스턴스들이 서로 다른 유닛들 안에서 쓰인다면, 컴파일러는 똑같은 코드를 끊임없이 만든다. 그 결과, 심각한 코드 팽창을 일으킬 수 있다. 그러므로, 만약 제네릭 클래스 안에 있는 메서드들 중 그 제네릭 타입에 의존하지 않는 것들이 많다면, 비-제네릭 클래스 안에 그런 메서드들을 담아 기반 클래스를 만든 다음, 그것을 상속받아서 제네릭 클래스를 만들고 제네릭 메서드들은 넣는 방법을 권장한다. 그러면, 그 비-제네릭 기반 클래스 안에 담긴 메서드들은 단 한 번만 컴파일 되어 실행파일 안에 들어간다.

참고 현재 컴파일러, 링커, 저수준 RTL에서는, 제네릭으로 인한 크기 증가를 줄이려는 작업이 진행되고 있다. <http://delphisorcery.blogspot.it/2014/10/new-language-feature-in-xe7.html>에 게시된 내용 역시 그 사례이다.

제네릭 타입 함수들 Generic Type Functions

우리가 지금까지 본 제네릭 타입 정의에는 가장 큰 문제가 있다. 그 제네릭 타입의 요소들을 가지고 여러분이 할 수 있는 게 매우 적다. 두 가지 기술을 사용하면 극복

할 수 있다. 첫째, 런타임 라이브러리에 있는 몇몇 특수한 함수들(제네릭을 특별하게 지원하는 함수들)을 사용하는 것; 둘째, (더 강력한 방법인) 제네릭 클래스를 정의할 때 여러분이 사용하고 싶은 타입으로 제약^{constraint}을 붙이는 것이다.

지금은 첫 번째 기법에 집중한다. 제약에 대해서는 다음 소단원에서 다룬다. 말했듯이, 몇몇 RTL 함수들은 제네릭으로 정의된 타입 (T)을 파라미터로 받아서 처리한다.

- **Default (T)**: 제네릭과 함께 새로 도입된 함수다. 빈^{empty} 값 즉 "영^{zero} 값"을 또는 현재 타입에 맞는 널^{null} 값을 반환한다; 0, 빈 문자열, nil 등등이다; 0으로 초기화된 메모리에 들어가는 값은 해당 타입의 전역 변수가 가지는 값과 똑같다 (로컬 변수와는 다르게 전역 변수는 컴파일러가 "0"으로 초기화한다).
- **TypeInfo (T)**: 제네릭 타입의 현재 버전에 맞는 런타임 정보를 가리키는 포인터를 반환한다; 타입 정보에 대한 더 많은 정보들은 16장에 설명되어 있다.
- **SizeOf (T)**: 타입의 바이트 수를 반환한다(문자열이나 오브젝트처럼 참조 타입인 경우에는 참조 크기를 반환한다. 참조는 32비트 컴파일러에서는 4바이트, 64비트 컴파일러에서는 8바이트가 사용된다)
- **IsManagedType (T)**: 메모리가 관리되는 타입인지 알려준다(문자열, 동적 배열 등).
- **HasWeakRef (T)**: ARC가 활성화된 컴파일러에서 타겟 타입이 약한 참조인지 알려준다; 이는 특별한 메모리 관리 지원이 필요하다.
- **GetTypeKind (T)**: 타입 정보에서 타입 종류에 접근하는 단축 경로다; TypeInfo에서 반환한 정보보다 더 고-수준^{high-level}의 타입 정의를 준다.

참고 이들 메서드들은 컴파일러가 평가^{evaluate}한 상수를 반환한다. 실제 함수를 런타임에 호출하는 게 아니다. 이것의 중요성은 이들의 연산이 매우 빠르다는 사실에 있지 않다. 오히려 컴파일러와 링커가 생성되는 코드를 최적화할 수 있게 한다는 점이다. 사용하지 않는 분기^{branch}를 제거하기 때문이다. 만일 여러분의 case나 if 문이 이 함수들이 반환하는 값에 기반한다면, 컴파일러는 그 문장을 이해할 수 있다. 주어진 타입만 보고 분기 중 하나가 실행될 것을 안다. 그래서 쓸모 없는 코드를 제거한다. 같은 제네릭 메서드가 다른 타입을 위해 컴파일된 경우에는, 결국 다른 분기를 사용한다. 하지만, 마찬가지로 컴파일러는 먼저 알고 메서드의 크기를 최적화한다.

GenericTypeFunc의 예시 안에 제네릭 클래스가 하나 있다. 이것은 제네릭 타입 함수 세 개를 실제로 보여준다:

```
type
  TSampleClass<T> = class
  private
    FData: T;
  public
    procedure Zero;
    function GetDataSize: Integer;
    function GetDataName: string;
  end;

function TSampleClass<T>.GetDataSize: Integer;
begin
  Result := SizeOf(T);
```



```

end;

function TSampleClass<T>.GetDataName: string;
begin
    Result := GetTypeName(TypeInfo(T));
end;

procedure TSampleClass<T>.Zero;
begin
    FData := Default(T);
end;

```

GetDataName 메서드는 GetTypeName 함수(System.TypeInfo 유닛 안에 있음)를 사용하고 있다. 해당 타입 이름을 담은 인코딩된^{encoded} 문자열을 알맞게 변환해주기 때문에 굳이 해당 데이터 구조에 직접 접근하지 않아도 된다.

위 선언이 주어졌으니, 여러분은 아래의 테스트 코드를 컴파일 할 수 있다. 이 코드는 서로 다른 세 가지 제네릭 타입 인스턴스들을 위해 세 번 반복된다. 반복되는 코드는 생략하고, data 필드에 접근하는데 사용하는 문장들만 남겨놓았다. 실제 타입에 따라 달라지는 문장들이기 때문이다:

```

var
    T1: TSampleClass<Integer>;
    T2: TSampleClass<string>;
    T3: TSampleClass<Double>;
begin
    T1 := TSampleClass<Integer>.Create;
    T1.Zero;
    Show( 'TSampleClass<Integer>' );
    Show( 'Data: ' + IntToStr(T1.FData));
    Show( 'Type: ' + T1.GetDataName);
    Show( 'Size: ' + IntToStr(T1.GetDataSize));

    T2 := TSampleClass<string>.Create;
    Show( 'Data: ' + T2.FData);

    T3 := TSampleClass<Double>.Create;
    Show( 'Data: ' + FloatToStr(T3.FData));

```

(GenericTypeFunc 예제에서) 이 코드를 실행하면 다음과 같이 출력된다:

```

TSampleClass<Integer>
Data: 0
Type: Integer
Size: 4
TSampleClass<string>
Data:
Type: string
Size: 4
TSampleClass<Double>
Data: 0
Type: Double
Size: 8

```


제네릭 타입 함수들은 구체적인 타입에 대해서도 사용할 수 있다는 점을 알아두자. 제네릭 클래스 문맥에서만 사용할 수 있는 게 아니다. 예를 들어, 이렇게 쓸 수 있다:

```
var
  I: Integer; s: string;
begin
  I := Default(Integer);
  Show( 'Default Integer: ' + IntToStr(I));

  s := Default(string);
  Show( 'Default String: ' + s);

  Show( 'TypeInfo String: ' + GetTypeName(TypeInfo(string)));
```

출력 결과는 뻔하다:

```
Default Integer: 0
Default String:
TypeInfo String: string
```

참고 여러분은 TypeInfo를 변수에 대해 호출할 수 없다. 즉, 위 코드 안에 있는 TypeInfo(s)처럼 사용할 수 없다. 오직 데이터 타입에 대해서만 호출할 수 있다.

제네릭 클래스를 위한 클래스 생성자 Class Constructors for Generic Classes

매우 흥미로운 상황이 여러분이 제네릭 클래스의 클래스 생성자를 정의할 때 생긴다. 사실, 제네릭 템플릿을 사용해 정의되는 실제 타입마다 각자의 생성자를 컴파일러가 만들어 넣는다. 그리고 각 제네릭 클래스 인스턴스는 그 생성자를 호출하여 생성된다. 상당히 흥미롭다. 왜냐하면 여러분은 여러분의 프로그램 안에 제네릭 클래스의 실제 인스턴스들을 생성하게 될 텐데, 클래스 생성자가 없다면 그 인스턴스를 위한 초기화 코드를 실행하기가 매우 복잡하기 때문이다.

예를 들어, 제네릭 클래스가 하나 있는데, 클래스 데이터 몇 개를 가지고 있다고 보자. 여러분이 얻게 되는 각 제네릭 클래스 인스턴스마다 그 안에는 그 클래스 데이터들의 인스턴스가 있게 된다. 만약 이 클래스 데이터들에 초기값을 대입하고 싶다면 어떻게 할까? 유닛의 초기화 코드를 사용할 수는 없다. 제네릭 클래스가 정의된 유닛에서는 실제로 어떤 클래스가 생겨날 지 알 수 없기 때문이다.

다음은 제네릭 클래스가 자신의 클래스 생성자를 사용하여 DataSize 클래스 필드를 초기화하는 골격을 보여주는 예시다 (GenericClassCtor 예제에서 발췌함)

```
type
  TGenericWithClassCtor<T> = class
  private
    FData: T;
    procedure SetData(const Value: T);
  public
    class constructor Create;
    property Data: T read FData write SetData;
    class var
      DataSize: Integer;
  end;
```


아래는 제네릭 클래스 생성자의 코드다. 어느 클래스 생성자가 실제로 호출되는지를 클래스 내부의 스트링리스트를 사용해 기록한다 (세부 구현은 전체 소스 코드 참조):

```
class constructor TGenericWithClassCtor<T>.Create;
begin
    DataSize := SizeOf(T);
    ListSequence.Add(ClassName);
end;
```

이 예제 프로그램은 이 제네릭 클래스의 인스턴스 몇 개를 생성하고 사용한다. 또한 데이터 타입을 세 번재를 위해 선언한다. 이것은 링커에 의해 제거된다:

```
var
    GenInt: TGenericWithClassCtor<SmallInt>;
    GenStr: TGenericWithClassCtor<string>;
type
    TGenDouble = TGenericWithClassCtor<Double>;
```

프로그램에게 ListSequence 스트링리스트의 내용을 보여주라고 하면, 실제 초기화된 타입들만 보여줄 것이다:

```
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

그러나, 같은 데이터 타입에 기반한 제네릭 인스턴스들을 서로 다른 유닛들 안에서 생성하면, 링커는 기대대로 작동하지 않고 같은 타입에 대해 제네릭 클래스 생성자 여러 개가 생기게 된다.

참고 이 문제를 다루는 것은 쉽지 않다. 초기화 반복을 피하려면, 여러분은 클래스 생성자가 이미 실행되었는지를 확인하면 된다. 그러나 일반적으로, 이 문제는 제네릭 클래스의 보다 광범위한 제한의 일부이다. 그리고 링커가 최적화를 못한다는 것이 문제다.

이 예제의 두 번째 유닛에 주석 해제할 경우 다음 초기화 순서처럼 문제가 보이는 Useless이라는 프로시저를 만들었다:

```
TGenericWithClassCtor<System.string>
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

제네릭 제약들 Generic Constraints

이미 보았듯이, 여러분이 제네릭 클래스의 메서드 안에서 제네릭 타입 값을 가지고 할 수 있는 것이 거의 없다. 여러분은 그 값을 다른 곳으로 넘겨주기 (즉 대입하기) 또는 제네릭 타입 함수(앞에서 설명함)가 제공하는 제한된 동작 정도만 할 수 있다.

클래스인 제네릭 타입의 실제 동작들을 수행할 수 있으려면, 여러분이 타입에 제약

`constraint`을 붙여야 한다. 예를 들어, 여러분은 제네릭 타입이 클래스여야 한다는 제한을 붙일 수 있다. 그러면, 컴파일러는 여러분이 `TObject`의 모든 메서드를 사용할 수 있게 해준다. 여러분은 제한을 더 크게 할 수도 있다. 그 클래스를 계층 상 더 하위에 있는 클래스를 지정하여 제한하면 된다. 또는 특정 인터페이스를 구현하도록 제한할 수도 있다. 그러면, 지정한 클래스나 인터페이스의 메서드들을 제네릭 타입의 인스턴스에서 호출할 수 있다.

클래스 제약들 Class Constraints

여러분이 쓰기에 가장 간단한 제약은 클래스 제약 `class constraint`이다. 클래스 제약을 사용하려면 제네릭 타입을 다음과 같이 선언한다:

```
type
  TSampleClass<T: class> = class
```

클래스 제약을 명시함으로써, 여러분은 그 제네릭 타입을 오직 오브젝트 타입으로만 사용하겠다고 알려주는 것이다. 아래 선언을 보자 (`ClassConstraint` 예제에서 발췌함):

```
type
  TSampleClass<T: class> = class
  private
    FData: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT(T1: T);
  end;
```

여러분은 아래에서 첫 두 인스턴스를 만들 수 있으나 세 번째 것은 만들지 못한다:

```
Sample1: TSampleClass<TButton>;
Sample2: TSampleClass<TStrings>;
Sample3: TSampleClass<Integer>; // 오류
```

위에서 마지막 선언이 발생시키는 컴파일러 에러는 다음과 같다:

```
E2511 Type parameter 'T' must be a class type
```

이렇게 제약을 명시하면 무슨 이득이 있을까? 이제 그 제네릭 클래스의 메서드들 안에서 여러분은 `TObject`의 모든 메서드를 호출할 수 있다(가상 메서드까지도)! `TSampleClass` 제네릭 클래스의 `One` 메서드를 보자:

```
procedure TSampleClass.One;
begin
  if Assigned(FData) then
    begin
      Form30.Show('ClassName: ' + FData.ClassName);
      Form30.Show('Size: ' + IntToStr(FData.InstanceSize));
      Form30.Show('ToString: ' + FData.ToString);
    end;
  end;
```


참고 두 가지 언급할 것이 있다. 첫째로, `InstanceSize`는 실제 오브젝트의 크기를 반환한다. 이와 달리 앞에서 사용했던 일반적인 `SizeOf` 함수는 참조 타입의 크기를 반환한다. 둘째로, `TObject` 클래스의 `ToString` 메서드를 사용하고 있다.

이제 프로그램을 가지고 그 실제 효과를 보자. 제네릭 타입의 인스턴스를 몇 개 정의하고 사용해보자. 다음 코드 조각과 같다:

```
var
  Sample1: TSampleClass<TButton>;
begin
  Sample1 := TSampleClass<TButton>.Create;
  try
    Sample1.SetT(Sender as TButton);
    Sample1.One;
  finally
    Sample1.Free;
  end;
```

주목할 점이 있다. 맞춤 정의한 `customized ToString` 메서드를 가진 클래스를 선언했다면, 데이터 오브젝트의 클래스가 실제 그 클래스인 경우, 호출되는 `ToString`은 그 클래스 버전의 메서드다. 인스턴스를 생성하기 위해 제네릭 타입에게 전달한 실제 타입이 무엇이든 관계없이 그렇다. 만일 `TButton`의 자식이 이렇다면:

```
type
  TMyButton = class(TButton)
  public
    function ToString: string; override;
  end;
```

여러분은 이 오브젝트를 `TSampleClass<TButton>`의 값으로 전달하거나 또는 그 제네릭 타입의 해당 타입 인스턴스를 정의하고 거기에 전달할 수도 있다. 두 경우 모두 `One`을 호출하면 이 버전의 `ToString`이 실행된다:

```
var
  Sample1: TSampleClass<TButton>;
  Sample2: TSampleClass<TMyButton>;
  Mb: TMyButton;
begin
  ...
  Sample1.SetT(Mb);
  Sample1.One;
  Sample2.SetT(Mb);
  Sample2.One;
```

클래스 제약과 비슷하게, 다음과 같이 레코드 제약 `record constraint`을 선언할 수 있다:

```
type
  TSampleRec<T: record> = class
```

그러나, 레코드들 사이에는 공통점이 거의 없다 (공통 조상이 없다). 따라서 이 선언은 다소 한계가 있다.

특정 클래스 제약들 Specific Class Constraints

만일 여러분의 제네릭 클래스가 (특정 계층 구조의) 특정 클래스들의 하위 집합^{subset}을 다루는 경우, 여러분은 그 기반 클래스를 제약으로 명시하고 싶을 것이다. 예를 들어, 다음과 같이 선언하면:

```
type
  TCompClass<T: TComponent> = class
```

이 제네릭 클래스의 인스턴스들은 컴포넌트 클래스들, 즉 TComponent의 자손 클래스로만 될 수 있다. 그러면 여러분은 매우 특정한 제네릭 타입(좀 이상하게 들리겠지만 그렇게 말할 수밖에 없다)을 쓸 수 있다. 그리고 컴파일러는 여러분이 제네릭 타입을 사용할 때 TComponent 클래스의 모든 메서드를 사용할 수 있도록 한다.

이것이 매우 강력해 보인다면, 다시 생각하라. 상속과 타입 호환성 규칙들만 가지고 해낼 수 있는 것이라면, 여러분은 그 문제를 전통적인 객체 지향 기법으로 해결할 수 있을 것이다. 제네릭 클래스를 쓰지 않고도 말이다. 제네릭에 클래스 제약을 명시하는 것이 쓸모 없다는 의미가 아니다. 하지만, 분명한 점이 있다. 제네릭 클래스 제약은 보다 더 수준 높은 클래스 제약 즉 인터페이스 기반 제약만큼 강력하지는 않다(매우 흥미롭다).

인터페이스 제약들 Interface Constraints

제네릭 클래스에게 클래스를 명시해 제약하는 것보다 대체로 더 유연한 방법이 있다. 주어진 인터페이스를 구현한 클래스들만 타입 파라미터로 받도록 제한하는 것이다. 그러면, 전달되는 제네릭 타입의 인스턴스들이 구현한 인터페이스를 불러 낼 수 있게 된다. 이와 같은 인터페이스 제약^{interface constraints} 사용은 C# 언어에서 매우 보편적이다. 예시를 보자 (IntfConstraint 예제에서 발췌함) 먼저, 인터페이스를 선언해야 한다:

```
type
  IGetValue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
    procedure SetValue(Value: Integer);
    property Value: Integer read GetValue write SetValue;
end;
```

그 다음, 그것을 구현하는 클래스를 정의한다:

```
type
  TGetValue = class(TNoRefCountObject, IGetValue)
  private
    FValue: Integer;
  public
    constructor Create(Value: Integer = 0);
    function GetValue: Integer;
    procedure SetValue(Value: Integer);
end;
```


주어진 인터페이스를 구현하는 타입들로만 제한하도록 제네릭 클래스를 정의해보자. 재미있어지기 시작한다:

```
type
  TInftClass<T: IGetValue> = class
  private
    FVal1, FVal2: T; // 즉 IGetValue
  public
    procedure Set1(Val: T);
    procedure Set2(Val: T);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
end;
```

이제 우리는 이 클래스의 제네릭 메서드 코드 안에 다음과 같은 코드를 쓸 수 있다:

```
function TInftClass<T>.GetMin: Integer;
begin
  Result := Min(FVal1.GetValue, FVal2.GetValue);
end;

procedure TInftClass<T>.IncreaseByTen;
begin
  FVal1.SetValue(FVal1.GetValue + 10);
  FVal2.Value := FVal2.Value + 10;
end;
```

이 모두가 정의되어 있으므로, 우리는 제네릭 클래스를 사용할 수 있다. 다음과 같다:

```
procedure TFormIntfConstraint.BtnValueClick(Sender: TObject);
var
  IClass: TInftClass<TGetValue>;
begin
  IClass := TInftClass<TGetValue>.Create;
  try
    IClass.Set1(TGetValue.Create(5));
    IClass.Set2(TGetValue.Create(25));
    Show('Average: ' + IntToStr(IClass.GetAverage));
    IClass.IncreaseByTen;
    Show('Min: ' + IntToStr(IClass.GetMin));
  finally
    IClass.FVal1.Free;
    IClass.FVal2.Free;
    IClass.Free;
  end;
end;
```

이 제네릭 클래스의 유연함 보여주고자, 그 인터페이스를 전혀 다르게 구현하는 것을 새로 만들어 보았다:

```
type
  TButtonValue = class(TButton, IGetValue)
  public
    function GetValue: Integer;
    procedure SetValue(Value: Integer);
```



```

    class function MakeTButtonValue(Owner: TComponent;
    Parent: TWinControl): TButtonValue;
end;

function TButtonValue.GetValue: Integer;
begin
    Result := Left; // 기반 클래스의 프로퍼티를 사용
end;

procedure TButtonValue.SetValue(Value: Integer);
begin
    Left := Value; // 기반 클래스의 프로퍼티를 사용
end;

```

(위에서 정의가 생략된) 클래스 함수는 버튼 하나를 Parent 컨트롤 안에 생성한다. 그 위치는 무작위로 결정된다. 그 클래스 함수를 사용하는 코드는 아래와 같다:

```

procedure TFormIntfConstraint.BtnValueButtonClick(Sender: TObject);
var
    IClass: TInftClass<TButtonValue>;
begin
    IClass := TInftClass<TButtonValue>.Create;
    try
        IClass.Set1(TButtonValue.MakeTButtonValue(Self, ScrollBox1));
        IClass.Set2(TButtonValue.MakeTButtonValue(Self, ScrollBox1));
        Show( 'Average: ' + IntToStr(IClass.GetAverage));
        Show( 'Min: ' + IntToStr(IClass.GetMin));
        IClass.IncreaseByTen;
        Show( 'New Average: ' + IntToStr(IClass.GetAverage));
    finally
        IClass.Free;
    end;
end;

```

인터페이스 참조 [Interface References](#) vs 제네릭 인터페이스 제약 [Generic Interface Constraints](#)

앞 예제에서 정의한 제네릭 클래스는 주어진 인터페이스를 구현하는 모든 오브젝트에 대해 작동하는 것이었다. 인터페이스 참조에 기반한 (제네릭 클래스가 아니라) 표준 클래스를 만들었다고 해도 그와 비슷한 효과를 얻을 수 있었을 것이다. 사실, 다음과 같이 클래스를 정의할 수도 있었다 (IntfConstraint 예제에서 발췌함):

```

type
    TPlainInftClass = class
    private
        FVal1, FVal2: IGetValue;
    public
        procedure Set1(Val: IGetValue);
        procedure Set2(Val: IGetValue);
        function GetMin: Integer;
        function GetAverage: Integer;
        procedure IncreaseByTen;
    end;

```


이 두 방식은 어떤 차이가 있을까? 첫 번째 차이점은, 위 클래스의 경우, 세터 메서드 각각에 전달되는 오브젝트들은 타입이 달라도 된다. 그 두 오브젝트 모두 `IGetValue` 라는 주어진 인터페이스를 구현하기만 하다면 말이다. 이와 달리, 제네릭 버전의 경우, (그 제네릭 클래스의 인스턴스에게) 전달되는 오브젝트는 오직 주어진 타입이나 그 타입의 자손 타입이어야만 한다. 그러므로, 제네릭 버전이 더 보수적^{conservative}이며 타입 검사 측면에서 더 엄격하다.

내가 볼 때, 핵심 차이점은 다음과 같다. 인터페이스-기반 버전을 사용하면, 오브젝트 파스칼의 참조 카운팅 메커니즘이 작동한다. 그런데, 제네릭 버전을 사용하면, 주어진 타입의 평범한 오브젝트를 클래스가 다루기 때문에 참조 카운팅은 관여하지 않는다.

거기에 더해, 제네릭 버전은 (생성자 제약 등) 제약을 여러 개 붙일 수도 있다. 또한 다양한 제네릭 함수(그 제네릭 타입의 실제 타입을 물어보기 등)들을 사용할 수 있다. 이런 것들은 인터페이스를 사용하는 경우에는 불가능하다. (인터페이스를 사용한다면, 사실 기반인 `TObject`의 메서드를 참조할 방법이 없다).

다른 말로 하면, 제네릭 클래스를 인터페이스 제약과 함께 쓰면 인터페이스의 성가신 점들 없이 그 장점을 취할 수 있다. 그러나 여전히 대부분의 경우 두 접근법은 동등하다는 점 그리고 다른 경우에는 인터페이스-기반의 해법이 더 유연할 수도 있다는 점을 알아 두면 좋다.

기본 생성자 제약 Default Constructor Constraint

사용할 수 있는 또 다른 제네릭 타입이 있다. 기본 생성자 즉 파라미터 없는 생성자라고 부르는 것이다. 만약 기본 생성자를 불러내어 그 제네릭 타입인 새 오브젝트를 생성해야 한다면, 예를 들어 목록을 채우려 한다면, 이 제약을 쓰면 된다. 이론적으로, 문서에 따르면, 컴파일러는 기본 생성자가 있는 타입들만 이것을 허용한다. 그렇지만 실제로는, 기본 생성자가 없는 경우에도 컴파일러는 그냥 허용한다. 그리고 `TObject`의 기본 생성자를 호출한다.

생성자 제약이 붙는 제네릭 클래스를 작성하려면 다음과 같이 한다 (`IntfConstraint` 예제에서 발췌함):

```
type
  TconstrClass<T: class, constructor> = class
  private
    FVal: T;
  public
    constructor Create;
    function Get: T;
  end;
```

참고 여러분은 클래스 제약을 생략하고 생성자 제약만 명시해도 된다. 생성자가 있는 타입은 당연히 클래스이기 때문이다. 하지만, 둘 다 나열해 놓으면 코드를 읽기가 더 쉽다.

위와 같이 선언되어 있으면, 이제 여러분은 실제 타입이 무엇인지 미리 알지 못해도, 그 생성자를 사용하여 내부 제네릭 오브젝트를 생성할 수 있다. 이렇게 적으면 된다:

```
constructor TConstrClass<T>.Create;
begin
  FVal := T.Create;
end;
```

이 제네릭 클래스를 어떻게 사용할까? 실제 무슨 규칙들이 있을까? 다음 예제를 보자. 클래스 두 개를 정의한다. 하나는 파라미터가 없는 기본 생성자를 가지고 있고, 다른 하나는 파라미터가 하나인 생성자를 가지고 있다:

```
type
  TSimpleConst = class
    public
      FValue: Integer;
      constructor Create; // 값을 10으로 설정한다
    end;

  TParamConst = class
    public
      FValue: Integer;
      constructor Create(I: Integer); // 값을 I로 설정한다
    end;
```

앞에서 말했듯이, 이론적으로는 첫 번째 클래스만 쓸 수 있지만, 실제로는 둘 다 쓸 수 있다:

```
var
  ConstructObj: TConstrClass<TSimpleConst>;
  ParamConstObj: TConstrClass<TParamConst>;
begin
  ConstructObj := TConstrClass<TSimpleConst>.Create;
  Show('Value 1: ' + IntToStr(ConstructObj.Get.FValue));

  ParamConstObj := TConstrClass<TParamConst>.Create;
  Show('Value 2: ' + IntToStr(ParamConstObj.Get.FValue));
```

이 코드의 출력 결과는 다음과 같다:

```
Value 1: 10
Value 2: 0
```

사실, 두 번째 오브젝트는 초기화되지 않는다. 애플리케이션을 디버깅^{debug}하여 코드를 타고 들어가면 `trace into TObject.Create` 가 호출된다는(잘못된 것이라고 생각된다) 것을 알 수 있다. 만약 직접적으로 호출을 한다면:

```
with TParamConst.Create do
```

컴파일러는 (올바르게) 오류를 일으킨다:

```
[DCC Error] E2035 Not enough actual parameters
```


참고 TParamConst.Create를 직접적으로 호출하면 (지금 설명했듯이) 컴파일 시간에 실패한다. 그렇지만, 클래스 참조를 사용하는 유사한 호출 즉 모든 간접적인 형식은 성공한다. 이는 생성자 제약의 효과가 동작하는 방식을 설명해 준다.

제약에 대한 요약 및 조합해서 사용하기 Constraints Summary and Combining them

제네릭 타입에 붙일 수 있는 제약들은 매우 많다. 짧은 요약해서 코드로 적어 보았다.

```
type
    TSampleClass<T: class> = class
    TSampleRec<T: record> = class
    TCompClass<T: TButton> = class
    TInftClass<T: IGetValue> = class
    TConstrClass<T: constructor> = class
```

위 요약 코드만으로 바로 알 수 없는 것이 있다 (난 익숙해지는데 시간이 좀 걸렸다) 이 제약들을 조합할 수 있다는 점이다. 예를 들어, 여러분은 제네릭 클래스 정의에서, 그것을 하위 계층 [sub-hierarchy](#)으로 한정할 수 있으며 그와 동시에 특정 인터페이스를 요구할 수 있다. 다음과 같이 작성하면 된다:

```
type
    TInftComp<T: TComponent, IGetValue> = class
```

아무 조합이나 합리적이지는 않다: 예를 들어 클래스와 레코드를 동시에 명시할 수 없다. 또한 클래스 제약을 사용하면서 동시에 특정 클래스 제약을 붙이는 것은 과잉이다. 끝으로, 메서드 제약 같은 것은 없다. 그런 것이 필요하다면, 단일 메서드 [single-method](#)를 가진 인터페이스 제약을 붙이면 된다 (표현하려면 훨씬 더 복잡하긴 하지만).

미리 정의된 제네릭 컨테이너들 Predefined Generic Containers

C++ 언어에 템플릿이 생겼을 때부터 C++ 언어가 표준 템플릿 라이브러리(즉 STL)를 정의하기 전까지, 템플릿 클래스들은 템플릿 컨테이너 [container](#)들 즉 리스트 [list](#)들을 정의하는 용도로 매우 많이 사용되었다.

오브젝트 파스칼에 있는 TObjectList와 같은 오브젝트의 리스트를 여러분이 정의하면, 그 리스트 안에 모든 종류의 오브젝트를 담을 수 있다. 여러분은 상속 또는 조합을 사용해 특정 타입을 담는 사용자 지정 컨테이너를 정의할 수도 있을 것이다. 그러나 그건 지루한 (그리고 오류가 나기 쉬운)방식이다.

오브젝트 파스칼은 컴파일러와 함께 제네릭 컨테이너 클래스들(Generics.Collections 유닛 안에 있음)을 제공한다. 네 개의 핵심 컨테이너 클래스들은 그 구현 방법이 서로 독립적이다(클래스 간 상속 관계가 없다) 하지만, 구현 스타일은 비슷하다(동적 배열을 사용한다). 그리고 모두가 각자에게 대응되는 (Contrrs 유닛 안에 있는) 제네릭이 아닌 컨테이너 클래스에게 매핑된다.


```

type
TList<T> = class
TQueue<T> = class
TStack<T> = class
TDictionary<K, V> = class
TObjectList<T: class> = class(TList<T>)
TObjectQueue<T: class> = class(TQueue<T>)
TObjectStack<T: class> = class(TStack<T>)
TObjectDictionary<K, V> = class(TDictionary<K, V>)

```

이 클래스들의 논리적 차이는 그 이름에 분명하게 드러난다. 이것들을 시험해보려면, 제네릭이 아닌 컨테이너 클래스 안에 기존과 다른 데이터 타입을 담고 싶을 때 기존 코드를 여러분이 얼마나 변경해야 하는지를 제네릭 버전과 비교해보는 것이다.

참고 이어서 볼 프로그램인 ListDemoMd2005는 메서드를 조금만 사용한다. 따라서 인터페이스 호환성 차이를 제네릭과 제네릭 아닌 리스트 사이에서 비교하기에 그다지 적합하지 않다. 하지만, 새로 코드를 짜지 않고 기존의 프로그램을 쓰기로 했다. 또 다른 이유도 있다. 여러분의 기존 프로그램들은 제네릭 컬렉션 [collection](#) 클래스들을 사용하지 않고 있을 수도 있을 텐데, 그것들이 언어 기능을 사용해 더 향상시키도록 독려하기 위해서다.

TList<T> 사용하기 [Using TList<T>](#)

이 ListDemoMd2005 프로그램에는, TDate 클래스를 정의하는 유닛과 메인 [main](#) 폼이 있다. 메인 폼은 날짜들의 TList를 참조한다. 맨 먼저, Generics.Collections을 참조하도록 `uses` 문을 추가한다. 그리고 메인 폼의 필드 선언을 아래와 같이 변경한다.

```

private
FListDate: TList<TDate>;

```

물론, 메인 폼의 OnCreate 이벤트 핸들러 역시 변경하여 리스트를 생성하도록 해야 한다. 다음과 같다:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  FListDate := TList<TDate>.Create;
end;

```

이제 나머지 코드를 그대로 컴파일할 수 있다. 그러면 프로그램은 “원했던” 버그를 얻는다. 리스트 안에 TButton 오브젝트를 추가하려고 시도하기 때문이다. 아래 코드는 컴파일에 사용되었으며, 지금은 실패한다:

```

procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // 리스트에 버튼을 추가
  FListDate.Add(Sender); // Error:
  // E2010 Incompatible types: 'TDate' and 'TObject'
end;

```

새 날짜 리스트는 타입 검사 면에서 원래의 포인터 제네릭 리스트보다 훨씬 견고하다.

문제의 줄을 제거하면, 프로그램은 컴파일 되고 작동한다. 그러나 여전히 개선할 수 있는 부분들이 있다.

아래는 원래 있던 코드다. 리스트 안의 모든 날짜들을 ListBox 컨트롤 안에 표현한다:

```
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    ListBox1.Items.Add((TObject(FListDate[I]) as TDate).Text);
```

타입 캐스트를 주목하자. 이 프로그램에서는 포인터의 리스트(TList)를 사용하고 있다. 오브젝트의 리스트(TObjectList)를 사용하고 있지 않기 때문이다.

손쉽게 개선할 수 있다. 다음과 같이 쓰면 된다:

```
for I := 0 to FListDate.Count - 1 do
  ListBox1.Items.Add(FListDate[I].Text);
```

또다른 코드 조각을 개선해보자. 일반 for 루프를 열거형으로 교체한다 (이 미리 정의된 제네릭 리스트는 열거형을 완전히 지원한다):

```
var
  ADate: TDate;
begin
  for ADate in FListDate do
    begin
      ListBox1.Items.Add(ADate.Text);
    end;
```

마지막으로, TDate 오브젝트를 담은 제네릭 TObjectList을 사용하도록 개선할 수 있다. 그것은 다음 소단원에서 다룬다.

이전에 말한 대로, TList<T> 제네릭 클래스는 호환성이 매우 높다. 이 클래스에는 Add, Insert, Remove, IndexOf 등의 클래스 메서드들이 있다. Capacity, Count 프로퍼티 역시 있다. 이상하게도, Items 가 Item으로 되었다. 하지만, 기본 프로퍼티이므로 (프로퍼티 이름을 적지 않고 대괄호 [square brackets](#)를 사용해 접근함), 직접 명시할 일은 거의 없다.

TList<T> 정렬하기 [Sorting a TList<T>](#)

TList<T>의 정렬 [sorting](#)이 어떻게 작동하지 이해하는 것은 흥미롭다 (여기서 목표는 ListDemoMd2005 예제에 정렬 지원을 추가하는 것이다). 그 Sort 메서드 정의는 이렇다:

```
procedure Sort; overload;
procedure Sort(const AComparer: IComparer<T>); overload;
```

IComparer<T> 인터페이스가 선언된 곳은 Generics.Defaults 유닛이다. 만약 여러분이 첫 번째 버전을 호출한다면 프로그램은 기본 비교를 사용한다. TList<T> 기본 생성자

에 의해 초기화되는 것인데 우리의 경우에는 쓸모가 없다.

우리에게 필요한 건 `IComparer<T>` 인터페이스에 대한 알맞은 구현을 정의하는 것이다. 타입 호환성을 위해, 우리는 이 특정 `TDate` 클래스에 효력이 있는 구현을 정의해야 한다.

이를 달성하는 방법은 여러 가지다. 익명 메서드도 그 중 하나다 (간단히 다음 소단원에서 설명하고 다음 장에서 전부 설명한다). 제네릭의 몇 가지 사용 패턴^{pattern}을 보여 줄 기회가 되기도 하는 흥미로운 기법이 있다. (내가 붙인 이름으로) **구조적 클래스** 즉 `Generics.Defaults` 유닛에 들어 있는 `TComparer`를 활용하는 것이다.

참고 내가 이 클래스를 **구조적**이라 부르는 이유는 코드의 구조 즉 그 아키텍처를 정의하는 데 도움이 되지만 실제 구현 측면에서는 많은 것을 추가하지 않기 때문이다. 이와 유사한 클래스로는 종종 프레임워크 클래스라는 것이 있는데, 대체로 설계가 보다 복잡한 것들을 그렇게 부른다.

이 클래스는 추상 클래스로 정의되었으며 인터페이스의 일반적인 구현이다:

```
type
    TComparer<T> = class(TInterfacedObject, IComparer<T>)
    public
        class function Default: IComparer<T>;
        class function Construct(
            const Comparison: TComparison<T>): IComparer<T>;
        function Compare(
            const Left, Right: T): Integer; virtual; abstract;
    end;
```

우리는 이제 이 제네릭 클래스를 특정 데이터 타입(예를 들어 여기에서는 `TDate`)으로 인스턴스화하고 그 특정 타입을 상속하고 그 안에 `Compare` 메서드를 구현할 구체적인 클래스를 만들어야 된다. 이 두 동작을 동시에 할 수 있다. 관용적 코드 표현^{idiom}을 사용하면 된다. 소화하는데 시간이 조금 걸릴 수 있다:

```
type
    TDateComparer = class(TComparer<TDate>)
    function Compare(
        const Left, Right: TDate): Integer; override;
    end;
```

이 코드가 매우 낯설게 보인다면, 여러분만 그런 것이 아니다. 이 새 클래스는 제네릭 클래스의 특정 인스턴스로부터 상속을 받는다. 즉, 위 코드는 두 단계로 나누어 아래와 같이 작성될 수도 있다:

```
type
    TAnyDateComparer = TComparer;
    TMyDateComparer = class(TAnyDateComparer)
    function Compare(
        const Left, Right: TDate): Integer; override;
    end;
```


참고 두 개의 구분된 선언은 생성된 코드의 크기를 줄이는 데 도움이 되는데 이는 기반 타입인 `TAnyDateComparer`를 같은 유닛에서 재사용하기 때문이다.

여러분은 소스 코드에서 `Compare` 함수의 실제 구현을 찾을 수 있는데, 여기서 강조하고자 하는 핵심 요소는 아니다. 그러나 리스트를 정렬하더라도, (`TStringList`와 달리) `IndexOf`메서드는 이 이점을 활용하지 못함을 명심하라.

익명 메서드를 사용해 정렬하기 Sorting with an Anonymous Method

앞 소단원에 본 정렬 코드는 상당히 복잡하게 보이며 실제로 그렇다. 정렬 함수를 직접 `Sort` 메서드에 전달하는 것이 더 쉽고 깔끔할 것이다. 과거에는 일반적으로 함수 포인터를 전달하여 수행했다. 오브젝트 파스칼에서 이것은 익명 메서드(다음 장에서 다룰 여러 추가 기능들이 있는 메서드 포인터의 일종)를 전달하여 수행할 수 있다.

참고 익명 메서드에 대해 잘 모르더라도 이 소단원을 읽은 뒤, 그 후에 자세하게 익명 메서드를 다루는 다음장을 읽은 후 한 번 더 살펴보는 것을 추천한다.

`TList<T>` 클래스의 `Sort` 메서드에 있는 `IComparer<T>` 파라미터는, 사실 `TComparer<T>`의 `Construct` 메서드를 호출하는 데 사용할 수 있다. 그래서 익명 메서드를 파라미터로 전달할 수 있다. 그 정의는 아래와 같다:

```
type
  TComparison<T> = reference to function(
    const Left, Right: T): Integer;
```

실전에서, 여러분은 타입이 호환되는 함수를 작성한다. 그리고 파라미터로 전달한다:

```
function DoCompare(const Left, Right: TDate): Integer;
var
  LDate, RDate: TDateTime;
begin
  LDate := EncodeDate(Left.Year, Left.Month, Left.Day);
  RDate := EncodeDate(Right.Year, Right.Month, Right.Day);
  if LDate = RDate then
    Result := 0
  else if LDate < RDate then
    Result := -1
  else
    Result := 1;
end;
procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  FListDate.Sort(TComparer.Construct(DoCompare));
end;
```

참고 위 `DoCompare` 메서드는 익명 메서드처럼 작동한다. 이름이 있는데 말이다. 이름이 필요 없는 코드 조각은 뒤에서 보게 된다. 오브젝트 파스칼 언어 구조에 관한 더 많은 정보는 다음 장까지 기다리자. 알아 둘 점이 있다. `TDate` 레코드를 가지고도 크기 비교 연산자들을 정의할 수 있다. 그러면 코드가 더 간단하다. 그런데, 심지어 클래스에서도 비교 코드를 메서드에 넣을 수 있다.

이것이 상당히 전통적인 것으로 보인다면, 구분된 함수의 정의를 회피하고 소스 코드 형태로 함수 구현을 Construct 메서드의 파라미터로 다음과 같이 전달할 수 있다:

```
procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort(TComparer<TDate>.Construct(
    function(const Left, Right: TDate): Integer
    var
      LDate, RDate: TDateTime;
    begin
      LDate := EncodeDate(Left.Year, Left.Month, Left.Day);
      RDate := EncodeDate(Right.Year, Right.Month, Right.Day);
      if LDate = RDate then
        Result := 0
      else if LDate < RDate then
        Result := -1
      else
        Result := 1;
      end));
end;
```

이 예제는 익명 메서드를 배우는 것에 대한 맛보기로 있다. 확실히 이 마지막 버전은 이전 소단원에서 다룬 원래 비교 코드보다 작성하기 훨씬 간단하지만, 많은 오브젝트 파스칼 개발자에게는 파생 클래스의 사용이 더 깔끔하고 이해하기 쉬울 수 있다(상속을 사용한 버전은 로직을 더 잘 분리하므로 잠재적인 코드 재사용이 더 쉽지만 결국 재사용하지 않는 경우가 많다).

오브젝트 컨테이너들 Object Containers

이 단원의 처음에 다룬 제네릭 클래스에 더해, Contnrs(*컨테이너*^{container}를 뜻함) 유닛의 기존 클래스들을 따라하는 Generics.Collections 유닛에 정의된 기반 클래스로부터 파생된 네 개의 상속된 제네릭 클래스들이 있다:

```
type
  TObjectList<T: class> = class(TList<T>)
  TObjectQueue<T: class> = class(TQueue<T>)
  TObjectStack<T: class> = class(TStack<T>)
```

그 기반 클래스와 비교했을 때, 두 가지 중요한 차이점이 있다. 하나는 이들 제네릭 타입이 오브젝트에만 사용 가능하다는 것이다; 두번째는, 이들이 오브젝트가 리스트에서 Free 될 때 (선택적으로 OnNotify 이벤트 핸들러를 호출하면서) 오브젝트를 해제하는 사용자 지정된 customized Notification 메서드를 정의한다.

다른 말로 하면, TObjectList<T> 클래스는 OwnsObjects 프로퍼티가 설정되었을 때 제네릭이 아닌 오브젝트 리스트와 똑같이 행동한다. 왜 이것이 더 이상 선택 가능한 사항이 아닌지 궁금하다면, 제네릭이 아닌 오브젝트 리스트와 달리 TList<T>가 이제는 오브젝트 타입에 대해 직접 쓰이기 때문임을 생각하자.

TObjectDictionary<K, V>라 불리는 네 번째 클래스도 있는데, 조금 다른 방법으로 정의되었으며, 키 오브젝트나 값 오브젝트, 혹은 둘 다 가질 수 있다. TDictionary-Ownerships 세트^{set}와 생성자로 자세한 사항을 참조하자.

제네릭 딕셔너리 사용하기 Using a Generic Dictionary

미리 정의된 제네릭 컨테이너 클래스 중 좀 더 자세히 살펴볼 가치가 있는 것은 제네릭 딕셔너리인 TObjectDictionary<K, V>다.

참고 여기서 말하는 *딕셔너리*란 요소들의 컬렉션이고, 각 요소마다 자신을 가리키는 고유한 키의 값을 가지고 있는 것을 뜻한다. 딕셔너리는 연관된 배열^{associative array}이라고도 알려져 있다. 전통적인 딕셔너리, 즉 사전에는 단어가 키의 역할을 하여 그 단어의 정의를 찾을 수 있다. 하지만, 프로그래밍 용어에서 딕셔너리의 ‘키’는 반드시 문자열일 필요는 없다 (그런 경우가 꽤 많지만 말이다). 과거 버전의 델파이 제네릭 컬렉션들 안에서는, 딕셔너리 타입의 자리 표시자로 TKey와 TValue를 사용했다. 그러나, TValue가 (16장에서 다루겠지만) 관련 없는 RTL 데이터 타입이라는 점때문에, 혼동을 막기 위해 델파이 11부터는 K와 V로 이름이 바뀌었다. 이에 따라 이 책도 위와 같이 수정되었다.

다른 클래스도 중요하지만, 그들은 사용하고 이해하기 쉽다. 딕셔너리의 사용 예제로, 데이터베이스 테이블로부터 데이터를 가져와 각 레코드를 위한 오브젝트를 만들고, 소비자 ID와 설명을 키로 합성^{composite}한 인덱스를 사용하는 애플리케이션을 작성하였다. 이것을 따로 구분한 이유는 유사한 아키텍처^{architecture}가 프록시^{proxy}(나중에 초기화하기)를 만드는 데 쉽게 쓰일 수 있으며, 데이터베이스에서 *적재*되는 실제 오브젝트의 경량화 된 버전으로 키를 대신할 수 있다.

아래에는 클래스 두 개가 있다(CustomerDictionary 예제에서 발췌). 각각 딕셔너리의 키와 실제 값으로 사용된다. 첫번째 클래스 안에는 해당 데이터베이스 테이블의 필드들 중 필요한 필드 두 개만 있다. 그리고 두번째 클래스 안에는 그 데이터베이스 테이블의 전체 필드들이 들어 간다 (아래 코드에서는 비공개^{private} 필드들, 게터^{getter} 메서드들, setter 메서드들은 생략함):

```
type
  TCustomerKey = class
  private
    ...
  published
    property CustNo: Double read FCustNo write SetCustNo;
    property Company: string read FCompany write SetCompany;
  end;

  TCustomer = class
  private
    ...
    procedure Init;
    procedure EnforceInit;
  public
    constructor Create(ACustKey: TCustomerKey);
```



```

    property CustKey: TCustomerKey
    read FCustKey write SetCustKey;
published
    property CustNo: Double
    read GetCustNo write SetCustNo;
    property Company: string
    read GetCompany write SetCompany;
    property Addr1: string
    read GetAddr1 write SetAddr1;
    property City: string
    read GetCity write SetCity;
    property State: string
    read GetState write SetState;
    property Zip: string
    read GetZip write SetZip;
    property Country: string
    read GetCountry write SetCountry;
    property Phone: string
    read GetPhone write SetPhone;
    property Fax: string
    read GetFax write SetFax;
    property Contact: string
    read GetContact write SetContact;
class var
    RefDataSet: TDataSet;
end;

```

첫 클래스는 굉장히 간단하지만(각 오브젝트가 생성될 때 초기화됨), TCustomer 클래스는 *나중에 초기화*되는 모델(프록시 모델)을 사용하고 모든 오브젝트가 (class var로) 공유하는 소스 데이터베이스의 참조를 저장한다.

참고 이 예제에서 TDataSet에서 파생된 오브젝트로 데이터베이스 내 데이터에 접근하여, 좀 더 현실 세계의 상황에 가깝게 하였다. 델파이에서 데이터베이스 접근에 대한 논의는 이 책의 범위를 한참 벗어나므로, 이 TDataSet 클래스의 실제 동작 성능은에 관한 추가적인 설명은 하지 않는다.

오브젝트가 생성되면 대응하는 TCustomerKey의 참조에 대입된다. 한편, 클래스 데이터 필드는 소스 데이터셋dataset을 참조한다. 각 게터getter 메서드 안에서, 클래스는 실제로 그 오브젝트가 초기화되었는지 확인한다. 그리고 나서 데이터를 반환한다. 다음과 같다:

```

function TCustomer.GetCompany: string;
begin
    EnforceInit;
    Result := FCompany;
end;

```

EnforceInit 메서드는 로컬 플래그 [local flag](#)를 확인한다. 그리고 최종적으로 Init 메서드를 호출한다. 그래서 데이터를 데이터베이스로부터 메모리 내 오브젝트로 적재한다:

```

procedure TCustomer.EnforceInit;
begin
    if not FInitDone then
        Init;
end;

```



```

procedure TCustomer.Init;
begin
    RefDataSet.Locate( 'CustNo', CustKey.CustNo, []);

    // RTTI를 통해 각 published 필드에 적재
    FCustNo := RefDataSet.FieldByName( 'CustNo').AsFloat;
    FCompany := RefDataSet.FieldByName( 'Company').AsString;
    FCountry := RefDataSet.FieldByName( 'Country').AsString;
    ...
    FInitDone := True;
end;

```

두 클래스가 주어졌으므로, 특별한 목적의 딕셔너리를 애플리케이션에 추가했다. 이 사용자 지정 딕셔너리 클래스는 제네릭 클래스로부터 상속받는다. 알맞은 타입을 사용해 인스턴스가 되는 클래스다. 거기에 특정한 메서드를 하나 추가했다:

```

type
    TCustomerDictionary = class(
        TObjectDictionary<TCustomerKey, TCustomer>)
    public
        procedure LoadFromDataSet(DataSet: TDataSet);
    end;

```

적재하는 메서드는 딕셔너리를 채우며, 키 오브젝트에만 데이터를 복사한다:

```

procedure TCustomerDictionary.LoadFromDataSet(DataSet: TDataSet);
var
    CustKey: TCustomerKey;
begin
    TCustomer.RefDataSet := DataSet;
    DataSet.First;
    while not DataSet.EOF do
        begin
            CustKey := TCustomerKey.Create;
            CustKey.CustNo := DataSet[ 'CustNo' ];
            CustKey.Company := DataSet[ 'Company' ];
            Self.Add(CustKey, TCustomer.Create(CustKey));
            DataSet.Next;
        end;
    end;

```

이 예제 프로그램에는 메인 폼과 데이터 모듈이 있다. 데이터 모듈은 ClientDataSet 컴포넌트를 담고 있다. 메인 폼에는 ListView 컨트롤이 있다. 이것은 사용자가 폼에 있는 하나밖에 없는 버튼을 누르면 채워진다.

참고 여러분은 예제를 확장해 실용적으로 쓰기 위해 ClientDataSet 컴포넌트를 실제 데이터셋으로 대체하고 싶을 것이다. 여러분은 키를 위한 쿼리^{query}와 각 TCustomer 오브젝트의 실제 데이터를 위한 질의^{query}를 별도로 가질 수 있다. 나에게 그런 코드가 있기는 하나, 여기 추가하지 않았다. 제네릭 딕셔너리 클래스를 실험한다는 이 예제의 목표로부터 너무 벗어날 수 있기 때문이다.

딕셔너리 안에 데이터를 적재한 다음, BtnPopulateClick 메서드는 딕셔너리의 키들에 대한 열거자^{enumerator}를 사용한다:


```

procedure TFormCustomerDictionary.BtnPopulateClick(Sender: TObject);
var
    CustKey: TCustomerKey;
    ListItem: TListItem;
begin
    DataModule1.ClientDataSet1.Active := True;
    CustDict.LoadFromDataSet(DataModule1.ClientDataSet1);
    for CustKey in CustDict.Keys do
        begin
            ListItem := ListView1.Items.Add;
            ListItem.Caption := CustKey.Company;
            ListItem.SubItems.Add(FloatToStr(CustKey.CustNo));
            ListItem.Data := CustKey;
        end;
    end;

```

이것은 ListView 컨트롤의 첫 두 열 [columns](#)을 채운다. 키 오브젝트 안에 있는 데이터가 들어간다. 사용자가 ListView 컨트롤의 아이템을 선택할 때마다, 프로그램은 세 번째 열을 채운다:

```

procedure TFormCustomerDictionary.ListView1SelectItem(
    Sender: TObject; Item: TListItem; Selected: Boolean);
var
    ACustomer: TCustomer;
begin
    ACustomer := CustDict.Items[Item.Data];
    Item.SubItems.Add(
        IfThen(
            ACustomer.State <> '',
            ACustomer.State + ', ' + ACustomer.Country,
            ACustomer.Country));
    end;

```

위 메서드는 해당 키에 매핑되는 오브젝트를 얻어 낸다. 그리고 그 데이터를 사용한다. 그 뒤편에서는, 특정 오브젝트가 처음으로 사용될 때, 해당 프로퍼티 접근 메서드가 TCustomer 오브젝트를 위한 전체 데이터 적제를 발동한다.

딕셔너리 vs 스트링 리스트 [Dictionaries vs. String Lists](#)

몇 년 간 많은 오브젝트 파스칼 개발자들은, 나 자신을 포함해, TStringList 클래스를 남용했다. 여러분은 이것을 문자열들에 대한 평범한 목록이나 이름/값 쌍들의 목록으로 사용할 수 있을 쓸 뿐만 아니라, 문자열에 연결된 오브젝트들의 목록으로도 사용할 수 있다. 그래서 그 오브젝트들을 탐색할 수도 있다. 그러나, 제네릭이 도입된 후에는, 제네릭 기반 클래스를 사용하는 것이 더 좋은 방법이다. TStringList를 스위스 아미 나이프 같은 만능 방식으로 쓰는 대신 말이다.

특화되고 집중된 컨테이너 클래스들을 쓰는 것이 훨씬 더 좋은 선택이다. 예를 들어, 제네릭 TDictionary에 문자열 키와 오브젝트 값을 넣어 쓰는 것이 TStringList를 두 번 쓰는 것보다 대체로 더 좋다: 더 깔끔하고 더 안전한 코드다. 타입 캐스트가 더 적게 관여한다. 또한, 더 빠르게 실행된다. 딕셔너리는 해시 테이블 [hash table](#)을 쓰기 때문이다.

이 차이점들을 보여 주기 위해, StringListVsDictionary라는 간단한 애플리케이션을 만들었다. 그 메인 폼은 똑같이 두 개의 리스트를 담는다. 그 선언은 다음과 같다:

```
private
    FList: TStringList;
    FDict: TDictionary<string, TMyObject>;
```

이 두 리스트는 임의의 동일한 엔트리를 아래 코드를 반복하는 루프로 채워진다:

```
FList.AddObject(AName, AnObject);
FDict.Add(AName, AnObject);
```

두 버튼은 리스트의 각 요소를 추출한다. 그리고 각각 이름으로 내용을 검색한다. 이 메서드 둘 다 값을 스트링 리스트를 훑어서 값을 찾는다. 하지만, 첫 번째 것은 스트링 리스트 안에서 오브젝트를 찾아 낸다. 그런데, 두번째 것은 딕셔너리를 사용한다. 잘 보면, 첫 번째 경우에는서는 여러분이 as 캐스트를 사용해야만 주어진 타입을 되찾을 수 있다. 한편, 딕셔너리는 그 클래스에 이미 묶여 있다. 여기 두 메서드의 메인 루프가 있다:

```
TheTotal := 0;
for I := 0 to SList.Count - 1 do
begin
    AName := FList[I];
    // 이제 탐색한다
    AnIndex := FList.IndexOf(AName);
    // 오브젝트를 가져온다
    AnObject := FList.Objects[AnIndex] as TMyObject;
    Inc(TheTotal, AnObject.Value);
end;

TheTotal := 0;
for I := 0 to FList.Count - 1 do
begin
    AName := FList[I];
    // 오브젝트를 가져온다
    AnObject := FDict.Items[AName];
    Inc(TheTotal, AnObject.Value);
end;
```

나는 문자열들에 순서대로 접근하고 싶지 않다. 오히려, 정렬된 스트링 리스트 안에서 검색(이진 탐색 [binary search](#)을 수행한다) 하는 것이 딕셔너리의 해시된 [hashed](#) 키와 비교했을 때 얼마나 많은 시간이 걸리는 지 알고 싶다. 놀랍지 않겠지만 딕셔너리가 더 빠르다. 테스트 결과를 밀리초 단위 숫자로 보면 아래와 같다:

```
Total: 99493811
StringList: 2839
Total: 99493811
Dictionary: 686
```

Total은 분명히 똑같다. 하지만, 소요 시간은 상당히 다르며, 딕셔너리가 100만 개에 가까운 엔트리에 대해 *사분의 일* 수준의 시간이 걸린다.

제네릭 인터페이스들 Generic Interfaces

“TList<T> 정렬하기” 소단원에서 여러분은 미리 정의된 인터페이스를 다소 이상하게 사용한다는 점을 눈치챘을 것이다. 그것은 제네릭 선언을 가지고 있었다. 이 기법을 자세히 볼 가치가 있다. 상당한 기회를 열어 줄 것이다.

첫 번째로 알아야 할 기술적 요소는 제네릭 인터페이스를 정의하는 것이 적법하다는 것이다. `GenericInterface` 예제에서 했던 것과 같다:

```
type
  IGetValue = interface
    function GetValue: T;
    procedure SetValue(Value: T);
end;
```

참고 이것은 IGetValue 인터페이스의 제네릭 버전이다. `IntfContraints` 예제에 있다. 이 장의 초반 소단원인 “인터페이스 제약”에서 다뤘다. 그 때는 인터페이스가 정수 값을 가졌었다. 하지만, 지금은 제네릭 값을 갖는다.

알아 둘 점이 있다. 표준 인터페이스와 달리, 제네릭 인터페이스는 인터페이스 ID(IID)로 사용하기 위해 여러분이 GUID를 명시할 필요가 없다. 컴파일러가 제네릭 인터페이스의 인스턴스마다 IID를 만들어준다. 심지어 암시적으로 선언이 되었더라도 그렇다. 사실, 여러분은 제네릭 인터페이스의 특정 인스턴스를 생성해서 구현할 필요가 없다. 하지만, 여러분은 그 제네릭 인터페이스를 구현하는 제네릭 클래스를 정의할 수 있다:

```
type
  TGetValue<T> = class(TInterfacedObject, IGetValue<T>)
  private
    FValue: T;
  public
    constructor Create(Value: T);
    destructor Destroy; override;
    function GetValue: T;
    procedure SetValue(Value: T);
end;
```

생성자는 그 오브젝트의 초기값을 대입한다. 한편, 소멸자의 유일한 목적은 오브젝트가 소멸했다는 기록을 남기는 것이다. 우리는 이 제네릭 클래스의 인스턴스를 생성할 수 있다 (그 뒤편에서는 그 인터페이스 타입의 특정 인스턴스를 생성한다). 다음과 같다:

```
procedure TFormGenericInterface.BtnValueClick(Sender: TObject);
var
  AVal: TGetValue<string>;
begin
  AVal := TGetValue<string>.Create(Caption);
  try
    Show( 'TGetValue value: ' + AVal.GetValue);
  finally
    AVal.Free;
  end;
end;
```


대안으로, 이전 `IntfConstraint` 예제에서 봤듯이, 대응하는 타입의 인터페이스 변수를 사용할 수 있다. 그래서 그 특정 인터페이스 타입 정의가 명시적이 되도록 한다(이전 코드 조각 안과 같이 암시적으로 하지 않는다).

```
procedure TFormGenericInterface.BtnIValueClick(Sender: TObject);
var
    AVal: IGetValue<string>;
begin
    AVal := TGetValue<string>.Create(Caption);
    Show( 'IGetValue value: ' + AVal.GetValue);
    // 참조 카운팅이 되므로 자동으로 해제된다.
end;
```

물론, 우리는 그 제네릭 인터페이스를 구현하는 구체적인 `specific` 클래스를 정의할 수도 있다. 다음 상황이 그렇다 (`GenericInterface` 예제에서 발췌함):

```
type
    TButtonValue = class(TButton, IGetValue<Integer>)
    public
        function GetValue: Integer;
        procedure SetValue(Value: Integer);
        class function MakeTButtonValue(Owner: TComponent;
            Parent: TWinControl): TButtonValue;
    end;
```

주의할 점이 있다. `TGetValue<T>` 제네릭 클래스가 `IGetValue<T>` 제네릭 인터페이스를 구현한다. 하지만, 구체적인 클래스인 `TButtonValue`가 `IGetValue<Integer>`라는 구체적 형태의 인터페이스를 구현하고 있다. 특히, 이전 예제에서 그랬듯이, 이 인터페이스는 그 컨트롤의 `Left` 프로퍼티에게 다시 매핑된다:

```
function TButtonValue.GetValue: Integer;
begin
    Result := Left;
end;
```

위 클래스에서, `MakeTButtonValue` 클래스 함수는 그 클래스의 오브젝트를 만들도록 미리 준비된 메서드다. 이 메서드는 메인 폼의 세 번째 버튼이 사용한다. 다음과 같다:

```
procedure TFormGenericInterface.BtnValueButtonClick(Sender: TObject);
var
    IVal: IGetValue<Integer>;
begin
    IVal := TButtonValue.MakeTButtonValue(Self, ScrollBox1);
    Show( 'Button value: ' + IntToStr(IVal.GetValue));
end;
```

제네릭 클래스와 전혀 관계없지만, 여기 `MakeTButtonValue` 클래스 함수의 구현이 있다:

```
class function TButtonValue.MakeTButtonValue(
    Owner: TComponent; Parent: TWinControl): TButtonValue;
begin
    Result := TButtonValue.Create(Owner);
    Result.Parent := Parent;
```



```

    Result.SetBounds(Random(Parent.Width),
        Random(Parent.Height), Result.Width, Result.Height);
    Result.Text := 'Btnv';
end;

```

미리 정의된 제네릭 인터페이스 Predefined Generic Interfaces

이제 우리는 제네릭 인터페이스를 정의하고 제네릭 및 구체화한 클래스와 조합하는 방법을 알아봤다. 이제 `Generics.Defaults` 유닛으로 돌아가서 다시 보자. 이 유닛은 두 개의 제네릭 비교 인터페이스를 정의한다:

- `IComparer<T>`에는 `Compare` 메서드가 있다
- `IEqualityComparer<T>`에는 `Equals`와 `GetHashCode` 메서드가 있다

이 클래스들은 제네릭 클래스들과 구체적인 클래스들 몇 가지로 구현되었다. 아래에 나열해 놓았다 (구현 세부 사항은 나열하지 않음):

```

type
    TComparer<T> = class(TInterfacedObject, IComparer<T>)
    TEqualityComparer<T> = class(
        TInterfacedObject, IEqualityComparer<T>)
    TCustomComparer<T> = class(TSingletonImplementation,
        IComparer<T>, IEqualityComparer<T>)
    TStringComparer = class(TCustomComparer<string>)

```

위 목록을 보면, 인터페이스의 제네릭 구현에 사용되는 기반 클래스는 고전적인 참조 카운팅이 되는 `TInterfacedObject` 클래스이거나 `TSingletonImplementation` 클래스 (`TNoRefCountObject` 클래스의 별칭 alias이다)임을 알 수 있다. 이름이 이상하게 지어진 클래스다. 이 클래스는 `IInterface`에 대한 기본 구현이면서 참조 카운팅이 없도록 한 것이다.

참고 싱글톤 singleton이라는 용어는 인스턴스를 오직 하나만 만들 수 있는 클래스를 정의할 때 사용되는 것이 일반적이다. 참조 카운팅이 없는 클래스에 쓰는 용어가 아니다. 위 이름은 상당히 잘못 붙여졌다고 생각한다.

이 장의 초반에서, 이미 “`TList<T>` 정렬하기” 소단원에서 본 대로, 이 비교 클래스들은 제네릭 컨테이너들이 사용한다. 그런데, `Generics.Default` 유닛은 익명 메서드에 크게 의존하기 때문에 더 복잡해진다. 그러니 이것은 다음 장을 읽고 난 뒤에 보는 것이 좋다.

오브젝트 파스칼의 스마트 포인터 Smart Pointers in Object Pascal

`generics`를 접근하다 보면, 자칫 잘못된 첫 인상 즉 제네릭 언어 구조는 주로 컬렉션 collection을 위해 사용된다고 생각할 수도 있다. 제네릭 클래스를 사용하는 가장 간단한

경우가 컬렉션이고, 제네릭을 다루는 책이나 문서도 맨 먼저 컬렉션 예제를 보여주는 경우가 많아서 그럴 것이다. 하지만, 제네릭은 컬렉션(즉 컨테이너) 클래스들의 세상 밖에서도 유용하다. *컬렉션이 아닌* 제네릭 타입들 중 한 가지 예를 보기로 하자. 바로 스마트 포인터 [smart pointer](#) 정의 하나를 이 장의 마지막 예시로 살펴보겠다.

오브젝트 파스칼만 사용해왔다면, 스마트 포인터에 관해 들어본 적이 없을 것이다. 그 아이디어는 C++ 언어에서 왔기 때문이다. C++에서는, 여러분이 오브젝트를 가리키는 포인터를 가질 수 있다. 포인터들은 여러분이 직접 수작업으로 관리해야 한다. 그리고 로컬 오브젝트 변수들은 자동으로 관리되지만, 다른 제한들이 많다 (다형성 [polymorphism](#) 결여 등). 스마트 포인터는 로컬에서 [locally](#) 관리되는 오브젝트를 사용해 포인터의 수명 [lifetime](#)을 보살피겠다는 개념이다. 그 포인터는 여러분이 사용하려는 진짜 오브젝트를 가리킨다. 너무 복잡하게 들린다면, 오브젝트 파스칼 버전으로 스마트 포인터를 보면 명확하게 이해할 수 있을 것이다.

참고 OOP 언어 용어로 다형성 [polymorphism](#)이란, 기반 클래스 변수에 그 자손 클래스의 오브젝트를 할당하고, 그 변수에서 기반 클래스의 가상 메서드를 호출하는 경우, 결국 호출되는 가상 메서드는 하위클래스 [subclass](#)에서 정의한 버전이 되는 상황을 일컫는다.

레코드를 스마트 포인터용으로 사용하기 [Using Records for Smart Pointers](#)

오브젝트 파스칼에서, 오브젝트는 참조에 의해 관리된다. 이와 달리, 레코드의 수명은 자신을 선언한 메서드의 수명 범위에 묶인다. 메서드가 끝나는 시점에 그 레코드를 위한 메모리 영역이 비워진다. 따라서 레코드를 사용해 오브젝트 파스칼 오브젝트의 수명을 관리하는 것이 가능하다.

텔파이 10.4 이전에는, 오브젝트 파스칼의 레코드가 사용자 지정 코드를 소멸 시점에 실행할 수 있는 방법이 없었다. 매니지드 레코드 [managed record](#)가 도입될 때 함께 생겨난 기능이기 때문이다. 그 전에 사용되던 방식은 인터페이스 필드를 레코드 안에 넣는 것이었다. 인터페이스 필드는 관리된다 [managed](#)는 점 그리고 그 인터페이스를 구현하는데 사용된 오브젝트는 참조 카운트가 감소한다는 점을 이용하는 방식이었다.

또 다른 고려 사항은 표준 레코드와 제네릭 레코드 중 무엇을 쓸 것인가이다. 표준 레코드 안에 TObject 타입 필드를 넣으면, 여러분은 필요할 때 그 오브젝트를 제거할 수 있다. 그러나 일반적인 상황에서 충분하다. 하지만, 제네릭 버전으로는, 두 가지 이점이 생긴다:

- 제네릭 스마트 포인터는 자신이 담고 있는 오브젝트에 대한 참조를 반환할 수 있다. 따라서 여러분은 참조 두 개를 유지할 필요가 없다
- 제네릭 스마트 포인터는 자동으로 컨테이너 오브젝트를 만든다. 이때 파라미터가 없는 생성자를 사용한다.

이제 제네릭 레코드를 사용해 구현되는 스마트 포인터 예시 두 개를 보자. 조금 더 복잡해지기는 할 것이다. 오브젝트 제약^{constraint}이 달린 제네릭 레코드부터 시작한다:

```
type
  TSmartPointer<T: class> = record
    strict private
      FValue: T;
      function GetValue: T;
    public
      constructor Create(AValue: T);
      property Value: T read GetValue;
  end;
```

위 레코드의 Create와 GetValue 메서드는 단순히 값을 대입하고 도로 읽을 뿐이다. 이것을 사용하는 코드 조각은 아래와 같다. 오브젝트 하나를 만들고, 그것을 감싸는 스마트 포인터를 만든다. 그리고, 그 스마트 포인터를 사용해 그 안에 담긴 오브젝트를 참조해서 그 오브젝트의 메서드를 호출한다 (마지막 줄을 보라):

```
var
  SL: TStringList;
begin
  SL := TStringList.Create;
  var SmartP: TSmartPointer<TStringList>.Create(SL);
  SL.Add( 'Foo' );
  SmartP.Value.Add( 'Bar' );
```

아마 예상했을 텐데, 위 코드는 메모리 누수를 발생시킨다. 스마트 포인터가 없는 것과 완전히 똑같다. 왜냐하면 이 레코드는 이 메서드의 범위를 벗어날 때 소멸한다. 하지만, 그 안에 담긴 오브젝트는 해제^{free}되지 않기 때문이다.

제네릭 매니지드 레코드 ^{Generic Managed Record}를 사용해 스마트 포인터 구현하기

위 스마트 포인터 레코드에게 가장 절실한 동작^{operation}은 종료화^{finalization}다. 그렇지만, 아래 코드에는 초기화 연산자도 추가했다. 오브젝트 참조를 nil로 설정하기 위해서다. 이상적으로는, (내부 오브젝트에 대한 참조가 여러 개인 경우 참조 카운트 매커니즘을 훨씬 복잡하게 구성해야 하기 때문에) 우리가 모든 대입 연산을 방지하면 좋겠지만, 그건 불가능하다고 가정하고, 연산자를 추가하고, 그 연산자가 발동하면 예외를 발생 시키도록 구현했다.

그 제네릭 매니지드 레코드의 전체 코드는 다음과 같다:

```
type
  TSmartPointer<T: class, constructor> = record
    strict private
      FValue: T;
      function GetValue: T;
    public
      class operator Initialize(out ARec: TSmartPointer<T>);
      class operator Finalize(var ARec: TSmartPointer<T>);
      class operator Assign(var ADest: TSmartPointer<T>);
```



```

    const [ref] ASrc: TSmartPointer<T>;
    constructor Create(AValue: T);
    property Value: T read GetValue;
end;

```

이 제네릭 레코드에는 class 제약 외에 constructor 제약도 있음을 주목하자. 그래야 해당 제네릭 데이터 타입의 오브젝트를 생성할 수 있기 때문이다. GetValue 메서드가 호출되는 경우에 그 오브젝트가 생성된다. 하지만 그 필드에 대한 초기화는 수행되지 않은 상태다:

```

constructor TSmartPointer<T>.Create(AValue: T);
begin
    FValue := AValue;
end;

class operator TSmartPointer<T>.Initialize(
    out ARec: TSmartPointer<T>);
begin
    ARec.FValue := nil;
end;

class operator TSmartPointer<T>.Finalize(
    var ARec: TSmartPointer<T>);
begin
    ARec.FValue.Free;
end;

class operator TSmartPointer<T>.Assign(
    var ADest: TSmartPointer<T>;
    const [ref] ASrc: TSmartPointer<T>);
begin
    raise Exception.Create('Cannot copy or assign a TSmartPointer<T>');
end;

function TSmartPointer<T>.GetValue: T;
begin
    if not Assigned(FValue) then
        FValue := T.Create;
    Result := FValue;
end;

```

이 코드는 SmartPointersMR 프로젝트에서 발췌했다. 거기에는 스마트 포인터를 어떻게 사용하는지에 대한 예시도 들어 있다. 첫 번째 방식은 이미 몇 페이지 앞에서 다뤘던 예시 코드와 매우 닮았다:

```

procedure TFormSmartPointers.BtnSmartClick(Sender: TObject);
var
    SL: TStringList;
begin
    SL := TStringList.Create;
    var SmartP := TSmartPointer<TStringList>.Create(SL);
    SL.Add('Foo');
    SmartP.Value.Add('Bar');
    Log('Count: ' + SL.Count.ToString);
end;

```


그런데, 이제 이 제네릭 스마트 포인터는 명시된 타입의 오브젝트를 자동 생성하도록 지원한다. 따라서, TStringList를 참조하는 명시적인 변수 그리고 그 TStringList를 생성하는 코드는 이제 없어도 된다.

```
procedure TFormSmartPointers.BtnSmartShortClick(Sender: TObject);
var
  SmartP: TSmartPointer<TStringList>;
begin
  SmartP.Value.Add( 'Foo' );
  SmartP.Value.Add( 'Bar' );
  Log( 'Count: ' + SmartP.Value.Count.ToString );
end;
```

프로그램에서, 여러분은 모든 오브젝트가 실제로 소멸했으며 메모리 누수가 없음을 확인할 수 있다. 프로그램의 초기화 코드 안에 전역 ReportMemoryLeaksOnShutdown을 True로 지정하면 된다. 그 반대로 테스트할 수 있도록 프로그램에는 누수를 일으키는 버튼을 만들어 두었다. 누수는 프로그램이 종료될 때 잡힐 것이다.

제네릭 레코드 [Generic Record](#)와 인터페이스 [Interface](#)를 사용해 스마트 포인터 구현하기

이미 언급한 대로, 매니지드 레코드가 생긴 델파이 10.4 이전에는 스마트 포인터를 구현하려면 인터페이스를 써야 했다. 레코드는 자신의 인터페이스 필드가 참조하는 오브젝트를 자동으로 해제한다는 점을 활용한 방식이다. 이제 이 방법은 덜 흥미롭게 되었지만, 암시적 변환 연산자 [Implicit conversion operator](#) 등 두세 가지 추가 기능을 제공한다.

여전히 흥미롭고 복잡한 예시가 남아 있으니 계속 가보자. 다만 설명은 좀 줄이겠다 (SmartPointers 프로젝트의 소스 코드다).

스마트 포인터를 인터페이스를 사용해 구현하려면, 여러분은 내부적인 지원 클래스를 작성해 그것을 인터페이스에 묶는다. 그리고, 그 인터페이스 참조 카운팅 메커니즘을 사용해 언제 오브젝트를 해제할지를 결정한다. 내부 클래스는 다음과 같이 생겼다:

```
type
  TFreeTheValue = class(TInterfacedObject)
  private
    FObjectToFree: TObject;
  public
    constructor Create(AnObjectToFree: TObject);
    destructor Destroy; override;
  end;

constructor TFreeTheValue.Create(AnObjectToFree: TObject);
begin
  FObjectToFree := AnObjectToFree;
end;

destructor TFreeTheValue.Destroy;
begin
  FObjectToFree.Free;
  inherited;
end;
```


이 클래스는 이 제네릭 스마트 포인터 타입에 대한 중첩된 타입으로 선언했다. 이제 이 스마트 포인터 제네릭 타입에서 우리가 해야 할 일은 이 기능을 활성화하는 것이다. 그렇게 하기 위해, 인터페이스 참조 하나를 추가하고 `TFreeTheValue` 오브젝트를 통해 초기화한다. 참고로 그 `TFreeTheValue` 오브젝트는 자신이 담고 있는 오브젝트를 참조하고 있다:

```
type
  TSmartPointer<T: class> = record
    strict private
      FValue: T;
      FFreeTheValue: IInterface;
      function GetValue: T;
    public
      constructor Create(AValue: T); overload;
      property Value: T read GetValue;
    end;
```

유사 생성자는 다음과 같다:

```
constructor TSmartPointer<T>.Create(AValue: T);
begin
  FValue := AValue;
  FFreeTheValue := TFreeTheValue.Create(FValue);
end;
```

이 코드가 있기 때문에, 이제 우리는 아래 코드를 프로그램 안에서 작성할 수 있다. 메모리 누수를 일으키지 않는 코드다 (이 코드는 처음에 나열했고 사용했던 매니지드 레코드 버전과 비슷하다):

```
procedure TFormSmartPointers.BtnSmartClick(Sender: TObject);
var
  SL: TStringList;
  SmartP: TSmartPointer<TStringList>;
begin
  SL := TStringList.Create;
  SmartP.Create(SL);
  SL.Add('Foo');
  SL.Add('Bar');
  Show('Count: ' + IntToStr(SL.Count));
end;
```

메서드의 마지막 부분에서 `SmartP` 레코드는 버려진다. 그러면 그 내부의 인터페이스 오브젝트가 소멸된다. 그러면서 `TStringList` 오브젝트를 해제^{free}한다.

참고 위 코드는 예외가 생겨도 실행된다. 위에서 인터페이스 필드를 가진 레코드를 사용했는데 사실, 여러분이 매니지드 타입을 사용하면, 컴파일러는 암시적 `try-finally` 블록을 사방에 추가한다.

암시적 변환 추가하기 Adding Implicit Conversion

매니지드 레코드 해법을 사용할 때, 우리는 레코드 복사 동작이 발생하지 않도록 더

주의해야 한다. 그런 경우에는 참조 카운팅 메커니즘을 수작업으로 추가할 필요가 있을 뿐만 아니라 레코드 구조가 훨씬 더 복잡해지기 때문이다. 하지만, 인터페이스 기반 해법으로 구축되어 있기 때문에, 우리는 이 모델을 활용해 변환 연산자들 [conversion operators](#)을 추가할 수 있다. 이런 연산자들이 있으면 데이터 구조를 초기화하고 생성하기가 더 간편하다. 구체적으로, 변환 연산자인 `Implicit`를 추가하기로 하자. 이 연산자는 타겟 오브젝트를 스마트 포인터에게 대입할 때 사용할 수 있다.

```
class operator TSmartPointer<T>.Implicit(AValue: T): TSmartPointer<T>;
begin
    Result := TSmartPointer<T>.Create(AValue);
end;
```

위 코드가 있으면 (그리고 `Value` 필드의 이점도 활용하면) 우리는 더 간결한 버전의 코드를 작성할 수 있다. 다음과 같다:

```
var
    SmartP: TSmartPointer<TStringList>;
begin
    SmartP := TStringList.Create;
    SmartP.Value.Add( 'Foo' );
    SmartP.Value.Add( 'Bar' );
    Show( 'Count: ' + IntToStr(SmartP.Value.Count));
```

다른 대안으로, `TStringList` 변수를 사용할 수 있다. 대신 이 스마트 포인터 레코드를 초기화하기 위해서 조금 더 복잡한 생성자를 사용한다. 명시적인 참조는 없어도 된다:

```
var
    SL: TStringList;
begin
    SL := TSmartPointer<TStringList>.Create(TStringList.Create).Value;
    SL.Add( 'Foo' );
    SL.Add( 'Bar' );
    Show( 'Count: ' + IntToStr(SL.Count));
```

여기까지 왔으니, 이제 우리는 그 반대 변환도 정의할 수 있다. 그리고 그 캐스트 [cast](#) 표기를 사용하면 된다. `Value` 프로퍼티 대신 말이다:

```
class operator TSmartPointer<T>.Implicit(AValue: T): TSmartPointer<T>;
begin
    Result := TSmartPointer<T>.Create(AValue);
end;

var
    SmartP: TSmartPointer<TStringList>;
begin
    SmartP := TStringList.Create;
    TStringList(SmartP).Add( 'Bar' );
```

이제, 위 코드에서 항상 유사 생성자를 사용하는 것을 눈치챘을 것이다. 레코드에서는 이것이 필요 없다. 우리가 필요한 것은 그저 내부 오브젝트를 초기화할 수 있는 방법이다. 아마 처음 사용할 때 그 생성자를 호출하는 것이 그 방법 중 하나일 것이다.

우리는 그 내부 오브젝트가 Assigned 되었는지 테스트하지 못한다. 레코드는 (클래스와 다르게) 그냥 0으로 초기화되지 않기 때문이다. 하지만 인터페이스 변수에 대해서는 그 테스트를 할 수 있다. 초기화가 되기 때문이다. 또 다른 대안으로, 우리는 추가 생성자 코드를 써서 레코드를 0으로 초기화할 수도 있다.

스마트 포인터 사용법 간의 비교 Comparing Smart Pointer Solutions

위 스마트 포인터에서는 매니지드 레코드 버전이 더 간단하고 꽤 효과적이다. 하지만, 인터페이스 버전은 변환 연산자라는 장점이 있다. 둘 다 저마다 장점이 있지만, 나는 개인적으로 매니지드 레코드 버전을 선호한다.

참고 (이 책의 범위를 벗어나는) 보다 명확한 분석과 정교한 해법은 Erik van Bilsen의 블로그 게시글을 추천한다: <https://blog.grijjy.com/2020/08/12/custommanaged-records-for-smart-pointers/>.

제네릭을 사용한 공변 반환 타입 Covariant Return Types with Generics

일반적으로 오브젝트 파스칼(과 대부분의 다른 정적 객체 지향 언어들)에서, 메서드는 클래스의 오브젝트를 반환할 수 있다. 하지만 여러분은 그 메서드를 자손 클래스에서 재정의(override)해서, 자손 클래스 오브젝트를 반환하도록 하지는 못한다. “공변 반환 타입”이라 불리는 흔한 기법인데, C++ 등 몇몇 언어들은 명시적으로 지원한다.

Animals, Dogs, Cats에 관하여 About Animals, Dogs, and Cats

TDog가 TAnimal를 상속받는다면, 아마 아래 메서드들을 원할 것이다:

```
function TAnimal.Get (AName: string): TAnimal;
function TDog.Get (AName: string): TDog;
```

오브젝트 파스칼에서는 반환 값이 다른 가상 함수를 가질 수 없다. 또한 오버로드(overload)도 그 반환 타입을 가지고는 하지 못하고, 오직 파라미터를 다르게 사용하는 것만 가능하다. 이 간단한 예제의 전체 코드를 보자. 다음과 같이 세 개의 클래스가 있다:

```
type
  TAnimal = class
    private
      FName: string;
      procedure SetName(const Value: string);
    public
      property Name: string read FName write SetName;
    public
      class function Get(const AName: string): TAnimal;
      function ToString: string; override;
    end;

  TDog = class(TAnimal)
    end;
```



```
TCat = class(TAnimal)
end;
```

위의 두 메서드는 구현이 상당히 간단하다. 그 중 클래스 함수는 사실 새 오브젝트를 만들 것이고, 그러기 위해 내부에서 생성자를 호출할 것이라는 것만 안다면 말이다.

생성자를 직접 만들지 않은 이유는 이것이 보다 일반적인 기법이기 때문이다. 여기 이 클래스 메서드는 다른 클래스(즉 클래스 계층)에 속하는 오브젝트를 만들 수 있다.

구현 코드는 다음과 같다:

```
class function TAnimal.Get(const AName: string): TAnimal;
begin
    Result := Create;
    Result.FName := AName;
end;

function TAnimal.ToString: string;
begin
    Result := 'This ' + Copy(ClassName, 2, MaxInt) +
        ' is called ' + FName;
end;
```

이제 우리는 이 클래스를 사용할 수 있다. 아래와 같이 코드를 적으면 된다. 하지만, 마음에 들지 않는다. 결과를 타입 캐스트 해서 바르게 되돌려 놓아야 하기 때문이다:

```
var
    ACat: TCat;
begin
    ACat := TCat.Get('Matisse') as TCat;
    Memo1.Lines.Add(ACat.ToString);
    ACat.Free;
```

위 코드 상, Get이 TCat 오브젝트를 반환하는 점은 확실하다. 따라서 위 코드 첫 줄을 다음과 같이 써도 된다 (실행 속도가 조금 더 빠르다):

```
ACat := TCat(TCat.Get('Matisse'));
```

다시 말하지만, 내가 하고 싶은 것은 TCat.Get이 반환하는 값을 TCat의 참조에 바로 대입하는 것이다. 명시적인 캐스트 없이 말이다. 어떻게 하면 될까?

제네릭 결과를 가지는 메서드 A Method with a Generic Result

제네릭이 이 문제를 해결하도록 도울 수 있다. 제네릭 타입(제네릭을 사용하는 가장 흔한 형식)이 아니라, 제네릭 메서드(이 장의 초반부에서 설명했다)를 사용하면 된다. TAnimal 클래스에 *제네릭 타입* 파라미터를 가지는 메서드를 다음과 같이 추가했다:

```
class function GetAs<T: class>(const AName: string): T;
```


이 메서드는 제네릭 타입 파라미터를 적도록 되어 있다. 클래스(또는 인스턴스의 타입)를 파라미터로 받고, 그 타입의 오브젝트를 반환한다. 구현 예문은 다음과 같다:

```
class function TAnimal.GetAs<T>(const AName: string): T;
var
  Res: TAnimal;
begin
  Res := Get(AName);
  if Res.InheritsFrom(T) then
    Result := T(Res)
  else
    Result := nil;
end;
```

이제 우리는 인스턴스를 생성하고 as 캐스트 없이도 사용할 수 있다. 하지만, 여전히 타입을 파라미터로 전달해야 한다:

```
var
  ADog: TDog;
begin
  ADog := TDog.GetAs<TDog>( 'Pluto');
  Memo1.Lines.Add(ADog.ToString);
  ADog.Free;
```

클래스가 다른 파생된 오브젝트 반환하기 Returning a Derived Object of a Different Class

클래스가 같은 오브젝트를 반환할 때는, 생성자들을 알맞게 사용해서 이 코드를 교체할 수 있다. 하지만 제네릭을 써서 공변 반환 타입을 얻는 것이 실제로 더 유연하다. 그렇게 하면, 우리는 다른 클래스(즉 클래스 계층)의 오브젝트를 반환할 수 있다:

```
type
  TAnimalShop = class
    class function GetAs<T: TAnimal, constructor>(
      const AName: string): T;
end;
```

참고 하나(또는 파라미터 또는 파라미터들에 따라 하나 이상의) 다른 클래스의 오브젝트를 만드는 클래스를 일반적으로 “클래스 팩토리”라고 부른다.

우리는 이제 (클래스 자체만으로는 불가능했던) 구체적인 클래스 제약을 사용할 수 있다. constructor 제약을 명시하여, 주어진 클래스의 오브젝트만 해당 제네릭 메서드 안에서 생성하도록 하면 된다.

```
class function TAnimalShop.GetAs<T>(const AName: string): T;
var
  Res: TAnimal;
begin
  Res := T.Create;
  Res.Name := AName;
  if Res.InheritsFrom(T) then
    Result := T(Res)
  else
```



```
    Result := nil;  
end;
```

이제는 보다시피, 호출할 때 해당 클래스 타입을 두 번 반복할 필요가 없다:

```
ADog := TAnimalShop.GetAs<TDog>( 'Pluto');
```


15: 익명 메서드 Anonymous Methods

오브젝트 파스칼 언어에는 프로시저 타입들(이 타입들은 포인터를 선언해서 프로시저와 함수를 가리킨다)과 메서드 포인터들(이 타입들은 포인터를 선언해서 메서드를 가리킨다)이 둘 다 있다.

참고 더 많은 정보를 알고 싶은 경우라면, 프로시저 타입은 4장에서 다뤘다. 한편, 이벤트와 메서드 포인터 타입들은 10장에서 설명했다.

여러분이 이것들을 “직접” 사용하는 경우는 거의 없을 것이다. 하지만 개발자 누구나 사용하고 있으며, 오브젝트 파스칼의 핵심 기능이다. 실제로, 메서드 포인터 타입은 컴포넌트들과 비주얼 컨트롤(visual control)들에 있는 이벤트 핸들러들의 기초다: 여러분이 이벤트 핸들러를 선언할 때마다, 심지어 간단한 Button1Click 하나조차도, 여러분은 메서드를 선언하고 그 메서드를 이벤트(이 경우에는 OnClick 이벤트)에 연결하고 있는 것이다. 그 연결에 메서드 포인터가 사용된다.

익명 메서드(anonymous method)는 이 기능을 확장한다. 그래서 여러분이 실제 메서드 코드를 파라미터로 전달할 수 있도록 해준다. 다른 곳에 정의된 메서드의 이름 대신에 말이다. 그런데 다른 점은 이것뿐만이 아니다. 익명 메서드가 다른 기법과 매우 다른 점은 바로 로컬 변수의 수명을 관리하는 방법이다.

위 정의는 (Javascript 등 다른 많은 언어에서) 클로저(closure)라고 부르는 특징과 같다. 실제로 오브젝트 파스칼의 익명 메서드가 클로저라면, 왜 언어에서 그것을 가리키는 용어가 다른 걸까? 이 두 가지 용어 모두 서로 다른 프로그래밍 언어에서 사용되고 있기 때문이다. 그리고 엠바카데로가 만드는 C++ 컴파일러(울킨이 - C++ Builder의 컴파일러)에서 이미 클로저(closure)라는 용어를 사용해 다른 기능 즉 오브젝트 파스칼에서 이벤트 핸들러라 부르는 기능을 가리키고 있기 때문이다. 익명 메서드는 지난 몇 년 간 몇몇 다른 프로그래밍 언어, 서로 다른 형태와 다른 이름으로 사용되어 오던 것이다. 특히

동적 언어 [dynamic language](#)에서 그렇다. 나는 자바스크립트 클로저를 매우 많이 경험했다. 특히 jQuery 라이브러리에서 그렇다. C#에서는 이에 대응하는 기능을 익명 델리게이트 [anonymous delegate](#)로 부른다.

그러나, 여기서 클로저 그리고 관련 기법들을 다양한 프로그래밍 언어에서 비교하고 싶지 않다. 오히려, 오브젝트 파스칼에서 이것이 어떻게 작동하는지 자세히 설명하겠다.

참고 매우 높은 관점에서 보면, 제네릭은 타입을 위해 코드가 파라미터가 될 수 있도록 하는 것이라고 볼 수 있다. 한편, 익명 메서드는 메서드를 위해 코드가 파라미터가 될 수 있도록 하는 것이다.

익명 메서드의 구문과 의미 [Syntax and Semantics of Anonymous Methods](#)

오브젝트 파스칼의 익명 메서드는 *메서드 값 생성을 표현식의 문맥* [expression context](#) 안에서 하는 메커니즘이다. 다소 난해한 정의 같지만, 오히려 정확하게, 메서드 포인터와의 핵심적인 차이 즉, *표현식의 문맥*을 잘 강조하고 있다. 이것을 파고들기 전에, 매우 간단한 코드 예제부터 시작하자 (AnonymFirst 예제 안에는 이 소단원에서 사용되는 코드들이 모두 있다). 익명 메서드 타입 선언은 아래 코드와 같다. 오브젝트 파스칼이 타입에 엄격한 언어이므로 여러분은 이 선언을 작성해야 한다:

```
type
  TIntProc = reference to procedure(N: Integer);
```

아래의 메서드 참조 타입 선언과 다른 점이 있다. 선언에 사용되는 키워드가 다르다:

```
type
  TIntMethod = procedure(N: Integer) of object;
```

익명 메서드 변수 [An Anonymous Method Variable](#)

익명 메서드 타입을 가졌으니 이제 여러분은, 이 간단한 경우에서, 그 타입의 변수를 선언하고, 그 타입에 호환되는 익명 메서드를 대입할 수 있다. 그러면 그 변수를 통해 대입한 메서드를 호출할 수 있다:

```
procedure TFormAnonymFirst.BtnSimpleVarClick(Sender: TObject);
var
  AnIntProc: TIntProc;
begin
  AnIntProc :=
    procedure(N: Integer)
    begin
      Memo1.Lines.Add(IntToStr(N));
    end;
  AnIntProc(22);
end;
```


위 코드에서 실제 프로시저를 대입하는 구문을 잘 보자. 그 자리에서 코드를 작성해 로컬 변수인 `AnIntProc`에 대입하고 있다.

익명 메서드 파라미터 An Anonymous Method Parameter

보다 흥미로운 예제를 보자(구문은 더 놀라울 것이다). 우리는 익명 메서드를 함수에게 파라미터로 전달할 수 있다. 함수가 익명 메서드 파라미터를 받는다고 보자:

```
procedure CallTwice(Value: Integer; AnIntProc: TIntProc);
begin
    AnIntProc(Value);
    Inc(Value);
    AnIntProc(Value);
end;
```

위 함수는 파라미터로 전달된 메서드를 두 번 호출한다. 단, 이어진 정수를 사용한다. 처음에는 파라미터로 전달된 정수를 그 다음에는 그보다 1 증가한 정수를 사용한다. 여러분은 위 함수를 호출할 때 실제 익명 메서드를 전달할 수 있다. 놀랍게도 그 자리에서 직접 코드를 작성할 수 있다. 아래와 같다:

```
procedure TFormAnonymFirst.BtnProcParamClick(Sender: TObject);
begin
    CallTwice(48,
        procedure(N: Integer)
        begin
            Show(IntToHex(N, 4));
        end);
    CallTwice(100,
        procedure(N: Integer)
        begin
            Show(FloatToStr(Sqrt(N)));
        end);
end;
```

구문의 관점에서 주목할 점이 있다. 이 파라미터로 프로시저가 전달되는데, 괄호 안에 들어 있다. 게다가 세미콜론(semicolon)으로 끝나지 않는다. 이 코드의 실제 효과를 보자. 48과 49를 전달해 `IntToHex`를 호출한다. 그리고 100과 101의 제곱근(square root)을 전달해 `FloatToStr`를 호출한다. 그 결과 출력은 다음과 같다:

```
0030
0031
10
10.0498756211209
```

로컬 변수 사용하기 Using Local Variables

똑같은 효과를 우리는 메서드 포인터를 사용해서 얻을 수 있다. 단, 그 구문은 다르고 읽기 힘들다. 오히려, 익명 메서드를 명확하게 차별화하는 점이 있다. 바로 호출하는

메서드에 있는 로컬 변수를 참조하는 방식이다. 다음 코드를 보자:

```
procedure TFormAnonymFirst.BtnLocalValClick(Sender: TObject);
var
  ANumber: Integer;
begin
  ANumber := 0;
  CallTwice(10,
    procedure(N: Integer)
    begin
      Inc(ANumber, N);
    end);
  Show(IntToStr(ANumber));
end;
```

위에서도 여전히 익명 메서드를 CallTwice 프로시저에게 전달한다. 그 익명 메서드 안에서는 로컬 파라미터 N을 사용한다. 그런데, 문맥`context`, 문맥 안에 있는 ANumber 로컬 변수도 사용하고 있다. 어떤 효과가 있을까? 익명 메서드는 두 번 호출되면서 로컬 변수인 ANumber를 변경한다. 처음에는 파라미터로 전달된 10을 더하고 두번째에는 11을 더한다. 그래서 ANumber의 최종 값은 21이 된다.

로컬 변수의 수명 확장하기 Extending the Lifetime of Local Variables

앞 예제에서 흥미로운 효과를 봤다. 그런데, 중첩된 함수 호출이 이어진다는 점에서, 그 로컬 변수를 사용할 수 있다는 사실은 그리 놀랍지 않다. 하지만, 익명 메서드의 힘은 따로 있다. 익명 메서드는 로컬 변수를 사용할 수 있을 뿐만 아니라 그 수명을 필요할 때까지 확장할 수 있다. 긴 설명 대신 하나의 예제를 통해 요점을 보자.

참고 다소 기술적인 세부 요소를 설명하자면, 익명 메서드는 자신이 사용하는 변수와 파라미터를 힙(heap)으로 복사한다. 자신이 생성될 때 그렇게 한다. 그리고 나서 자신이 (익명 메서드) 인스턴스로 살아있는 동안 그것들을 계속 유지한다.

AnonymFirst 예제의 TFormAnonymFirst 폼 클래스에 (클래스 자동 완성을 사용해) 프로퍼티로 익명 메서드 포인터 타입을 하나 추가했다(글쎄, 실제로 이 프로젝트의 모든 코드에서는 똑같이 이 익명 메서드 포인터 타입을 사용한다).

```
private
  FAnonMeth: TIntProc;
procedure SetAnonMeth(const Value: TIntProc);
public
  property AnonMeth: TIntProc
    read FAnonMeth write SetAnonMeth;
```

그리고 프로그램의 폼에 버튼을 두 개를 더 추가했다. 첫번째 버튼은 그 프로퍼티, 즉 익명 메서드를 저장한다. 그런데, 그 익명 메서드는 우리가 앞에서 본 BtnLocalValClick 메서드 안에 있던 것과 같이 로컬 변수를 사용한다.

```
procedure TFormAnonymFirst.BtnStoreClick(Sender: TObject);
```



```

var
  ANumber: Integer;
begin
  ANumber := 3;
  AnonMeth :=
    procedure(N: Integer)
    begin
      Inc(ANumber, N);
      Show(IntToStr(ANumber));
    end;
end;

```

위 메서드를 실행하면, 그 익명 메서드는 실행되지 않는다. 그저 저장만 된다. 로컬 변수인 `ANumber`는 3으로 초기화된다. 그리고 수정되지 않고 로컬 범위를 벗어난다(메서드가 끝나기 때문이다). 그래서 사라진다. 적어도, 이것은 여러분이 표준 오브젝트 파스칼 코드에서 예상한 그대로다.

두 번째 버튼은 `AnonMeth` 프로퍼티에 저장된 익명 메서드를 아래와 같이 호출한다.

```

procedure TFormAnonymFirst.BtnCallClick(Sender: TObject);
begin
  if Assigned(AnonMeth) then
    begin
      CallTwice(2, AnonMeth);
    end;
end;

```

위 코드가 실행되면, 익명 메서드를 호출한다. 그런데, 그 익명 메서드가 사용하는 `ANumber` 로컬 변수는 더 이상 스택에 남아있지 않다. 하지만, 익명 메서드는 자신이 실행되는 문맥(context)을 잡아두고 있기 때문에, 그 변수는 그대로 거기에 있다. 그리고 주어진 익명 메서드 인스턴스(즉, 그 메서드에 대한 참조)가 존재하는 동안 사용된다.

더 분명히 증명하기 위해 다음과 같이 해보자. `Store` 버튼을 한 번 누르고, `Call` 버튼을 두 번 누른다. 획득된 같은 변수가 사용되고 있다는 것을 볼 수 있다:

```

5
8
10
13

```

참고 이 결과가 나온 이유를 보자. 값은 3에서 시작한다. `CallTwice`를 호출할 때마다 익명 메서드는 파라미터를 전달한다. `CallTwice`는 처음에는 파라미터로 받은 값(2이다)을 사용해 익명 메서드를 실행하고, 두 번째 호출은 1을 증가한 후(3이다)를 사용해 다시 실행한다.

이제 `Store`를 한 번 더 누르고 `Call`을 한 번 더 누른다. 무슨 일일까? 왜 로컬 변수의 값이 리셋(reset)되었을까? 새 익명 메서드 인스턴스를 대입함으로써, 이전 익명 메서드가 제거되었다(그 실행 문맥도 함께 제거됨). 그리고 새 실행 문맥이 잡혔다. 그 로컬 변수 안에 들어간 새 인스턴스와 함께 획득된 것이다. 전체 테스트 순서인 `Store - Call - Call - Store - Call`을 진행하면 다음과 같은 결과가 나온다:


```
5
8
10
13
5
8
```

이 동작은 이렇게 구현되어 있다. 다른 언어들에 하는 것과 닮았다. 이 구현 덕분에 익명 메서드는 매우 강력한 언어 요소가 되었다. 여러분은 이전까지 그저 불가능했던 것들을 구현하는 데 사용할 수 있다.

익명 메서드의 이면 Anonymous Methods Behind the Scenes

만약 이런 변수 잡아두기가 익명 메서드에서 가장 쓸모 있는 특징이라면, 봐 둘 만한 기법들이 몇 가지 더 있다. 실제 세상의 몇 가지 예를 보기 전에 먼저 보기로 하자. *여전히, 익명 메서드가 처음이라면, 여러분은 이 고급 단계 소단원을 건너뛰고 나중에 다시 돌아와 읽기로 해도 좋다.*

(잠재적으로) 생략된 괄호 The (Potentially) Missing Parenthesis

위 코드를 잘 보면, `AnonMeth` 심볼을 사용해 익명 메서드를 가리키고 있다. 호출하는 게 아니다. 호출하려면 아래와 같이 쓰면 된다:

```
AnonMeth(2)
```

차이점은 명백하다; 메서드를 호출하려면 적절한 파라미터를 전달해야 한다. 파라미터 없는 익명 메서드에서는 이것이 조금 더 혼란스럽다. 만약 다음과 같이 선언하면:

```
type
  TAnyProc = reference to procedure;
var
  AnyProc: TAnyProc;
```

`AnyProc`를 호출하려면 반드시 그 뒤에 빈 괄호를 붙여야 한다. 그렇게 하지 않으면 컴파일러는 단순히 그 메서드(그 주소)를 가져오려는 것으로 생각한다. 즉 메서드를 호출하지 않는다:

```
AnyProc();
```

유사한 일이 익명 메서드를 반환하는 함수를 호출할 때 생긴다(`AnonymFirst` 예제에서 발췌함).

```
function GetShowMethod: TIntProc;
var
  X: Integer;
begin
  X := Random(100);
```



```

    ShowMessage('New x is ' + IntToStr(X));
    Result :=
        procedure(N: Integer)
        begin
            X := X + N;
            ShowMessage(IntToStr(X));
        end;
    end;

```

이제 질문이 있다: 어떻게 호출할까? 만약 여러분이 아래와 같이 그냥 호출하면:

```
GetShowMethod;
```

컴파일 되고 실행도 된다. 하지만, 이 함수는 익명 메서드를 대입하는 코드를 호출하는 것만 한다. 익명 메서드가 반환 값에 담기긴 하지만 그냥 버려진다.

어떻게 하면 파라미터를 전달하는 그 실제 익명 메서드를 호출할 수 있을까? 한 가지 방법은 익명 메서드 변수를 임시로 사용하는 것이다.

```

var
    IP: TIntProc;
begin
    IP := GetShowMethod(); // 괄호 필요!!!
    IP(3);

```

이 경우 주목할 점이 있다. GetShowMethod 호출 뒤에 괄호가 있다. 만일 생략할 경우 (표준적인 파스칼에서) 다음과 같은 오류를 얻는다:

```
E2010 Incompatible types: 'TIntProc' and 'Procedure'
```

괄호가 없으면, 컴파일러는 여러분이 메서드 포인터인 IP 안에 대입하려는 것이 GetShowMethod 함수 자체라고 판단한다. 그 함수의 결과를 대입하려는 의도라고 생각하지 않는다. 여전히, 임시 변수 사용은 이 경우 최고의 방법이 아닐 수 있다. 코드가 자연스럽지 않고 복잡해지기 때문이다. 아래의 간단한 호출은 어떨까?

```
GetShowMethod(3);
```

이것은 컴파일 되지 않는다. 왜냐하면, 이 메서드는 파라미터를 받지 않기 때문이다. 여러분은 첫 호출에서 빈 괄호를 붙여주어야 한다. 그리고 그 뒤에서 정수 파라미터를 전달해야 한다. 메서드 호출 결과인 익명 메서드에게 파라미터를 전달하기 위해서다. 즉, 여러분은 이렇게 쓸 수 있다:

```
GetShowMethod()(3);
```

익명 메서드 구현 Implementation of Anonymous Methods

익명 메서드의 구현 뒤에서는 무슨 일이 벌어질까? 컴파일러가 익명 메서드를 위해 만들어 내는 실제 코드는 인터페이스를 기반으로 한다. 그리고 Invoke 메서드를 호출

한다 (숨겨져 있다). 또한 주로 참조 카운팅을 지원한다 (익명 메서드와 그것이 잡고 있는 문맥의 수명을 결정하는 데 유용하다).

그 내부 세부 사항을 들여다보는 것은 아마 매우 복잡하고 그만큼의 가치가 안된다. 이 구현은 속도 측면에서 매우 효율적이고, 각 익명 메서드마다 약 500바이트의 추가 메모리를 차지한다는 점만 알아도 충분할 것이다.

다른 말로 하면, 오브젝트 파스칼의 메서드 참조는 *특별한* 단일 메서드 인터페이스로 구현된다. 그리고 구현하는 메서드 참조와 서명이 똑같은 메서드를 컴파일러가 만든다. 그 인터페이스는 참조 카운팅을 활용한다. 그래서 스스로를 자동으로 파기할 수 있다.

참고 실제로 익명 메서드에 쓰이는 인터페이스가 다른 인터페이스처럼 생겼더라도, 컴파일러는 이들 특별한 인터페이스들을 구분한다. 따라서 여러분은 코드에서 그것들을 섞어 쓰지 못한다.

이 숨겨진 인터페이스 외에도, 각 익명 메서드 호출마다 컴파일러는 숨겨진 오브젝트를 만든다. 그 안에는 해당 메서드 구현 그리고 호출 문맥을 잡고 있기 위해 필요한 데이터가 들어 있다. 그래서 그 메서드를 호출할 때마다 여러분이 *획득된* 변수들을 새로 얻게 된다.

바로 사용할 수 있는 참조 타입들 Ready-To-Use Reference Types

익명 메서드를 파라미터로 사용할 때마다 여러분은 대응하는 참조 포인터 데이터 타입을 정의해야 한다. 로컬 타입들이 증폭되는 것을 막기 위해, 오브젝트 파스칼은 미리 사용하도록 준비된 몇 가지 참조 포인터 타입을 System.SysUtils 유닛에 넣어 놓았다.

아래 코드 조각에서 볼 수 있듯이, 이 타입들의 정의는 대부분 파라미터화 된 타입 *parametrized type*을 사용한다. 따라서 여러분은 제네릭한 선언 하나로도, 서로 다른 참조 포인터 타입들을 각 가능한 데이터 타입에 대해 얻을 수 있다:

```
type
  TProc = reference to procedure;
  TProc<T> = reference to procedure(Arg1: T);
  TProc<T1, T2> = reference to procedure(
    Arg1: T1; Arg2: T2);
  TProc<T1, T2, T3> = reference to procedure(
    Arg1: T1; Arg2: T2; Arg3: T3);
  TProc<T1, T2, T3, T4> = reference to procedure(
    Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4);
```

위 선언을 사용해, 여러분은 익명 메서드 파라미터를 받는 프로시저를 정의할 수 있다. 다음과 같다:

```
procedure UseCode(Proc: TProc);
function DoThis(Proc: TProc): string;
function DoThat(ProcInt: TProc<Integer>): string;
```


첫째와 둘째 경우에 여러분은 파라미터 없는 익명 메서드를 전달한다. 세번째 경우에 여러분은 정수 하나를 파라미터로 받는 익명 메서드를 전달한다:

```
UseCode(
    procedure
    begin
        // 약간의 코드
    end);
StrRes := DoThat(
    procedure(I: Integer)
    begin
        // 약간의 코드
    end);
```

위와 마찬가지로, System.SysUtils 유닛에는 제네릭한 반환 값을 가지는 익명 메서드들의 묶음이 정의되어 있다:

```
type
TFunc

```

이 정의들은 매우 광범위하다. 여러분은 셀 수 없는 데이터 타입 조합을 사용할 수 있다. 최대 4개의 파라미터와 하나의 반환 타입을 조합할 수 있다. 맨 마지막 정의는 두 번째 정의와 매우 비슷하다. 하지만, 매우 자주 사용되는 특정 상황, 즉 제네릭한 파라미터를 받고 불리언 타입을 반환하는 함수에 맞춰져 있다.

현실의 익명 메서드 Anonymous Methods in the Real World

처음 볼 때는, 익명 메서드가 가진 힘과 그것을 사용하면 이득이 되는 상황을 완전히 이해하기 어렵다. 그러니 이 언어를 다루는 더 복잡한 예제를 제시하는 대신, 실제로 영향을 미치고 더 깊이 탐색하는데 출발점이 되는 예제 몇 가지에 집중하겠다.

익명 이벤트 핸들러 Anonymous Event Handlers

오브젝트 파스칼의 차별화된 특징 중 하나는 이벤트 핸들러의 구현이다. 이것은 메서드 포인터를 사용한다. 익명 메서드는 이벤트에 새로운 동작들을 붙이는 데 쓸 수 있다. 메서드를 따로 선언하지 않아도 되고, 그 메서드의 실행 문맥을 따로 잡아두지 않아도 되기 때문에 좋다. 그래서 한 메서드에서 다른 메서드로 파라미터를 전달하기 위해,

폼에 필드들을 추가로 더 넣는 방식을 더 이상 사용하지 않아도 된다.

그 예시로 (AnonButton 예제에서), *익명 클릭* 이벤트를 버튼에 추가했다. 그러기 위해, 알맞은 메서드 포인터 타입을 선언하고, 새 이벤트 핸들러를 사용자 지정 버튼 클래스에게 추가했다(인터포저 클래스를 사용해 정의했다)

```
type
  TAnonNotif = reference to procedure(Sender: TObject);

  // 인터포저 클래스
  TButton = class(FMX.StdCtrls.TButton)
  private
    FAnonClick: TAnonNotif;
    procedure SetAnonClick(const Value: TAnonNotif);
  public
    procedure Click; override;
  public
    property AnonClick: TAnonNotif
      read FAnonClick write SetAnonClick;
  end;
```

참고 인터포저 클래스는 그 기반 클래스와 이름이 같은 파생된 클래스다. 이름이 같은 두 클래스를 가지는 것은 가능하다. 그 두 클래스가 서로 다른 유닛에 있다면 말이다. 결국 최종 전체 이름 (*유닛명.클래스명*)은 서로 다르다. 인터포저 클래스를 선언하면 편할 때가 있다. 여러분은 그저 버튼 컨트롤 하나를 폼 위에 올려 놓고 거기에 추가 동작을 붙일 수 있다. 새 컴포넌트를 IDE 안에 설치해서 여러분의 폼 위에 있던 컨트롤들을 그 새 타입으로 교체할 필요가 없다. 여러분이 이 폼수를 위해 기억해야 하는 것은 오직 한 가지다. 인터포저 클래스 정의가 별도의 유닛 안에 있다면(이 예제에서는 폼이 있는 그 유닛 안에 있다), 그 유닛은 `uses` 문 [statement](#) 안에서 반드시 그 기반 클래스를 정의하고 있는 유닛보다 뒤에 나열되어야 한다.

이 클래스의 코드는 매우 단순하다. 세터 메서드는 새로운 포인터를 저장한다. 그리고 Click 메서드는 그것을 먼저 호출하고 나서 표준 처리(즉, OnClick 이벤트 핸들러가 존재한다면 그것을 호출한다)를 수행한다:

```
procedure TButton.SetAnonClick(const Value: TAnonNotif);
begin
  FAnonClick := Value;
end;

procedure TButton.Click;
begin
  if Assigned(FAnonClick) then
    FAnonClick(Self)
  inherited;
end;
```

여러분이 이 새 이벤트 핸들러를 사용하는 방법을 살펴보자. 기본적으로 익명 메서드를 거기에 대입하면 된다:

```
procedure TFormAnonButton.BtnAssignClick(Sender: TObject);
begin
```



```

    BtnInvoke.AnonClick :=
    procedure(Sender: TObject)
    begin
        Show((Sender as TButton).Text);
    end;
end;

```

여기까지 보면, 그저 무의미해 보인다. 여러분은 표준 이벤트 핸들러 메서드로도 같은 효과를 쉽게 얻을 수 있기 때문이다. 하지만, 아래 코드부터 차이가 나기 시작한다. 이 익명 메서드는 그 이벤트 핸들러를 대입하는 컴포넌트를 가리키는 참조를 획득하기 때문이다. 그러기 위해 Sender 파라미터를 참조하고 있다.

아래 코드는 그 참조를 먼저 임시로 로컬 변수에 대입한 후에 수행한다. 익명 메서드의 Sender 파라미터가 BtnKeepRefClick 메서드의 Sender 파라미터를 숨기기 전에 말이다:

```

procedure TFormAnonButton.BtnKeepRefClick(Sender: TObject);
var
    ACompRef: TComponent;
begin
    ACompRef := Sender as TComponent;
    BtnInvoke.AnonClick :=
    procedure(Sender: TObject)
    begin
        Show((Sender as TButton).Text +
            ' assigned by ' + ACompRef.Name);
    end;
end;

```

BtnInvoke 버튼을 눌렀을 때, 여러분은 그 버튼의 캡션caption과 그 익명 메서드 핸들러를 대입한 컴포넌트의 이름을 함께 볼 수 있다.

익명 메서드의 시간 측정하기 Timing Anonymous Methods

개발자들은 속도를 상대적으로 비교하기 위해 시간 측정 코드timing code를 기존 루틴에 추가하는 경우가 자주 있다. 여러분이 코드 조각 두 개의 속도를 비교하기 위해 수백만 번 실행해보고 싶다고 가정해보자. 여러분은 6장의 LargeString 예제에서 본 다음과 같은 코드를 쓸 수 있다:

```

procedure TForm1.Button1Click(Sender: TObject);
var
    Str1, Str2: string;
    I: Integer;
    T1: TStopwatch;
begin
    Str1 := 'Marco ';
    Str2 := 'Cantu ';

    T1 := TStopwatch.StartNew;
    for I := 1 to MaxLoop do
        Str1 := Str1 + Str2;
    T1.Stop;
    Show('Length: ' + Str1.Length.ToString);

```



```
Show('Concatenation: ' + T1.ElapsedMilliseconds.ToString);
end;
```

두번째 메서드는 코드가 비슷하다. 하지만 평범한 문자열 이어 붙이기 [concatenation](#) 대신 `StringBuilder` 클래스를 사용했다.

이제 우리는 익명 메서드를 활용할 수 있다. 시간 측정 골격을 만들고 그것을 특정 코드에게 파라미터로 전달할 수 있다. 그 코드의 업데이트 버전은 `AnonLargeStrings` 예제에 넣어두었다. 시간 측정 코드를 계속해서 반복 작성하기보다, 여러분은 함수를 하나 작성해 그 안에 시간 측정 코드를 넣고, 그 함수가 비교하고 싶은 코드 조각을 불러낼 수 있도록 하면 된다. 그 호출은 파라미터가 없는 익명 메서드를 사용한다:

```
function TimeCode(NLoops: Integer; Proc: TProc): string;
var
  T1: TStopwatch;
  I: Integer;
begin
  T1 := TStopwatch.StartNew;
  for I := 1 to NLoops do
    Proc;
  T1.Stop;
  Result := T1.ElapsedMilliseconds.ToString;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Str1, Str2: string;
begin
  Str1 := 'Marco ';
  Str2 := 'Cantu ';
  Show('Concatenation: ' +
    TimeCode(MaxLoop,
      procedure()
      begin
        Str1 := Str1 + Str2;
      end));
  Show('Length: ' + Str1.Length.ToString);
end;
```

알아 둘 점이 있다. 여러분이 표준 버전과 익명 메서드 기반 버전을 실행한다면, 출력 결과가 약간 다를 것이다. 익명 메서드 버전이 10% 정도 더 나쁘다.

그 이유는, 프로그램이 직접 로컬 코드 [local code](#)를 실행하지 않고, 익명 메서드 구현에 대한 가상 [virtual](#) 메서드를 호출하기 때문이다. 이 차이점이 일관적이므로, 테스트하는 코드는 완벽하게 합리적이다. 그 두 구현으로 얻은 테스트 결과들이 서로 교환되면서 사용되지 않는 한 말이다.

그러나, 여러분의 코드에서 성능을 쥐어짜야 하는 경우, 익명 메서드를 사용하는 것은 함수를 직접 사용하도록 코드를 작성한 것만큼 빠르지 않다. 성능 측면에서는 아마 가상이 아닌 [nonvirtual](#) 메서드 포인터를 사용한다면 이 두 경우 사이 정도가 될 것이다.

쓰레드 동기화 Threads Synchronization

멀티-쓰레드 애플리케이션 안에서 그 사용자 인터페이스를 변경해야 하는 경우, 비주얼 컴포넌트들의 프로퍼티들(즉 메모리 안에 있는 오브젝트들)은 메인 쓰레드에 속하기 때문에 여러분이 동기화(synchronization) 메커니즘 없이는 접근할 수 없다. 델파이의 비주얼 컴포넌트 라이브러리들은 본질적으로 쓰레드-안전(thread-safe)하지 않다(사용자 인터페이스 라이브러리들 대부분이 사실 그렇다). 즉 만일 둘 이상의 쓰레드가 오브젝트 하나에게 동시에 접근하는 경우에 그 오브젝트의 상태가 손상될 수 있다는 뜻이다.

오브젝트 파스칼의 클래식한 해법은 TThread 클래스다. 이것은 Synchronize라는 특별한 메서드를 호출한다. 그 때 파라미터로 또다른 메서드(안전하게 실행되어야 하는 메서드)에 대한 참조를 전달한다. 이 두 번째 메서드는 파라미터를 가질 수 없다. 따라서 혼란 관행이 생겼다. 추가 필드들을 쓰레드 클래스에 넣어서 해당 정보를 한 메서드에서 다른 메서드로 전달하는 방법이다. 실용적인 예제로, 나는 Mastering Delphi 2005라는 책에서 WebFind 애플리케이션을 작성했다(구글 검색을 HTTP를 통해 수행하고 검색 결과인 HTML 페이지에서 링크들을 추출하는 프로그램). 그 때, 아래에 있는 쓰레드 클래스를 사용했다:

```
type
TFindWebThread = class(TThread)
protected
  FAddr, FText, FStatus: string;
  procedure Execute; override;
  procedure AddToList;
  procedure ShowStatus;
  procedure GrabHtml;
  procedure HtmlToList;
  procedure HttpWork(Sender: TObject;
    AWorkMode: TWorkMode; AWorkCount: Int64);
public
  FStrUrl: string;
  FStrRead: string;
end;
```

세 개의 protected^{보호된} 문자열 필드 그리고 몇몇 추가 메서드들은 사용자 인터페이스 동기화를 지원하기 위해 선언되었다. 예를 들어, HttpWork 이벤트 핸들러는 내부 컴포넌트인 THTTPClient의 OnReceiveData 이벤트에게 걸린다. 참고로 THTTPClient는 델파이의 HTTP Client 라이브러리에 들어 있는 컴포넌트다. 어쨌든 HttpWork 이벤트 핸들러는 ShowStatus 메서드를 호출한다. 다음 코드와 같다:

```
procedure TFindWebThread.HttpWork(const Sender: TObject;
  AContentLength, AReadCount: Int64; var AAbort: Boolean);
begin
  FStatus := 'Received ' + IntToStr(AReadCount) + ' for ' + FStrUrl;
  Synchronize(ShowStatus);
end;

procedure TFindWebThread.ShowStatus;
begin
  Form1.StatusBar1.SimpleText := FStatus;
end;
```


오브젝트 파스칼 RTL에 있는 Synchronize 메서드는 서로 다르게 오버로드된 두 개의 정의를 가지고 있다:

```
type
  TThreadMethod = procedure of object;
  TThreadProcedure = reference to procedure;

  TThread = class
    procedure Synchronize(AMethod: TThreadMethod); overload;
    procedure Synchronize(AThreadProc: TThreadProcedure); overload;
```

익명 메서드 파라미터가 있는 버전을 사용해, 우리는 폼 클래스에서 FStatus 텍스트 필드와 ShowStatus 함수를 제거할 수 있다. 또한 HttpWork 이벤트 핸들러는 새 버전인 Synchronize의 익명 메서드 파라미터를 사용하도록 다시 작성할 수 있다:

```
procedure TFindWebThreadAnon.HttpWork(const Sender: TObject;
  AContentLength, AReadCount: Int64; var AAbort: Boolean);
begin
  Synchronize(
    procedure
    begin
      Form1.StatusBar1.SimpleText :=
        'Received ' + IntToStr(AReadCount) +
        ' for ' + FStrUrl;
    end);
end;
```

똑같은 방식을 이 클래스의 코드 전체에 사용한다. 그러면 이 쓰레드 클래스는 다음과 같이 된다(이 두 개의 쓰레드 클래스들은 모두 WebFind 예제 안에 있다):

```
type
  TFindWebThreadAnon = class(TThread)
  protected
    procedure Execute; override;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure HttpWork(const Sender: TObject; AContentLength: Int64;
      AReadCount: Int64; var AAbort: Boolean);
  public
    FStrUrl: string;
    FStrRead: string;
  end;
```

익명 메서드를 쓰면 쓰레드 동기화가 필요한 코드를 간결하게 할 수 있다. 여러분이 임시 필드를 사용하지 않을 수 있기 때문이다.

참고 익명 메서드는 쓰레딩과 관계가 많다. 쓰레드는 코드 실행에 사용되고, 익명 메서드는 코드 표현에 사용되기 때문이다. 그래서 TThread 클래스 안에서는 그것들을 사용할 수 있도록 지원하고 있다. 뿐만 아니라 병렬 프로그래밍 라이브러리 안에도 지원한다 (TParallel.For 안에서 그리고 TTask 정의에서). 익명 메서드를 사용할 수 있는 지원이 있는 이유다. 멀티-쓰레딩 실험은 이 장의 주제를 넘어선다. 따라서 그 방향으로 예제를 더 추가하지 않겠다. 하지만 여전히 또다른 쓰레드 하나를 다음 예제로 사용한다. HTTP 호출을 만드는 데 거의 필수적이기 때문이다.

오브젝트 파스칼에서의AJAX AJAX in Object Pascal

이 소단원의 마지막 예제인, AnonAjax 애플리케이션 예제는, (조금 어렵더라도) 저자가 좋아하는 익명 메서드 예제들 중 하나다. 그 이유는 몇 년 전 내가 AJAX 애플리케이션을 jQuery 라이브러리로 작성하며 자바스크립트 [Javascript](#)로 클로저 [closure](#) (혹은 익명 메서드)를 배웠기 때문이다.

참고 AJAX란 Asynchronous JavaScript XML을 줄인 말이다. 원래는 웹 브라우저가 웹 서비스를 호출하는 데 사용되는 형식 [format](#)이었다. 웹 기술이 점차 대중화되고 널리 퍼지면서, 웹 서비스는 REST 아키텍처와 JSON 형식으로 넘어갔다. 그리고, AJAX란 용어는 사라져 가고 그 자리를 REST가 차지했다. 어쨌든 이 예제에서는 이 오래된 이름을 그냥 유지하기로 했다. 예제 애플리케이션의 목적과 그 뒤에 있는 역사를 설명하기 때문이다. 더 많은 사항은 다음 링크에 있다: [https://en.wikipedia.org/wiki/Ajax_\(programming\)](https://en.wikipedia.org/wiki/Ajax_(programming)).

AjaxCall 전역 함수는 쓰레드를 하나 만들고, 그 쓰레드에게 익명 메서드를 전달한다. 익명 메서드는 그 쓰레드가 끝날 때 실행할 것이다. 이 전역 함수는 쓰레드 생성자를 감싸는 래퍼 [wrapper](#)일 뿐이다:

```
type
  TAjaxCallback = reference to procedure(
    ResponseContent: TStringStream);
procedure AjaxCall(const StrUrl: string;
  AjaxCallback: TAjaxCallback);
begin
  TAjaxThread.Create(StrUrl, AjaxCallback);
end;
```

모든 코드는 TAjaxThread 클래스라는 쓰레드 클래스 안에 있다. 이것은 클라이언트 컴포넌트인 THTTPClient(HTTP Client 라이브러리에 들어 있음)를 가지고 있다. URL에 비동기로 [asynchronously](#) 접근할 때 사용된다.

```
type
  TAjaxThread = class(TThread)
  private
    FHttp: THTTPClient;
    FUrl: string;
    FAjaxCallback: TAjaxCallback;
  protected
    procedure Execute; override;
  public
    constructor Create(const StrUrl: string;
      AjaxCallback: TAjaxCallback);
    destructor Destroy; override;
end;
```

이 생성자는 몇 가지 초기화를 한다. 받은 파라미터들을 복사해 쓰레드 클래스에 있는 대응하는 로컬 필드들에 넣는다. 그리고 FHttp 오브젝트를 생성한다. 이 클래스의 진짜 알맹이는 Execute 메서드 안에 있다. 이것은 HTTP 요청 [request](#)을 수행한다. 그 결과는 스트림 안에 저장한다. 그 스트림은 나중에 재설정되고 콜백 함수(즉 그 익명 메서드)에게 전달된다.


```

procedure TAjaxThread.Execute;
var
  AResponseContent: TStringStream;
begin
  AResponseContent := TStringStream.Create;
  try
    FHttp.Get(FUrl, AResponseContent);
    AResponseContent.Position := 0;
    FAjaxCallback(AResponseContent);
  finally
    AResponseContent.Free;
  end;
end;

```

그 사용 예시로 AnonAjax 예제를 보자. 버튼이 하나 있다. 이것은 웹 페이지의 내용을 복사해서 메모 컨트롤에 넣는다 (요청했던 URL을 추가한 곳 다음에):

```

procedure TFormAnonAjax.BtnReadClick(Sender: TObject);
begin
  AjaxCall(EdUrl.Text,
    procedure (AResponseContent: TStringStream)
    begin
      Memo1.Lines.Text := AResponseContent.DataString;
      Memo1.Lines.Insert(0, 'From URL: ' + EdUrl.Text);
    end);
end;

```

HTTP 요청이 끝난 후, 여러분은 거기에 어떠한 종류의 처리도 할 수 있다. 예를 들어, HTML에서 링크를 추출한다 (앞에서 WebFind 애플리케이션이 했던 것과 비슷하다). 이 함수를 더 유연하게 하기 위해, 파라미터로 익명 메서드를 받는다. 그래서 각 링크마다 실행되도록 한다:

```

type
  TLinkCallback = reference to procedure(const StrLink: string);
procedure ExtractLinks(StrData: string; ProcLink: TLinkCallback);
var
  StrAddr: string;
  NBegin, NEnd: Integer;
begin
  StrData := LowerCase(StrData);
  NBegin := 1;
  repeat
    NBegin := PosEx('href="http', StrData, NBegin);
    if NBegin <> 0 then
      begin
        // HTTP 참조의 끝부분을 찾기
        NBegin := NBegin + 6;
        NEnd := PosEx('"', StrData, NBegin);
        StrAddr := Copy(StrData, NBegin, NEnd - NBegin);
        // 이동
        NBegin := NEnd + 1;
        // 익명 함수 실행
        ProcLink(StrAddr)
      end;
    until NBegin = 0;
end;

```


만약 여러분이 이 함수를 AJAX 호출의 결과에 적용하고 메서드를 하나 더 제공하여 처리하도록 하면, 여러분은 두개의 중첩된 익명 메서드 호출을 보게 된다. AnonAjax 예제의 두 번째 버튼처럼 말이다:

```
procedure TFormAnonAjax.BtnLinksClick(Sender: TObject);
begin
  AjaxCall(EdUrl.Text,
    procedure(AResponseContent: TStringStream)
    begin
      ExtractLinks(AResponseContent.DataString,
        procedure(const AUrl: string)
        begin
          Memo1.Lines.Add(AUrl + ' in ' + EdUrl.Text);
        end);
    end);
end;
```

이 경우 메모 컨트롤은 링크들의 컬렉션을 얻는다. 반환된 페이지의 HTML이 아니다. 위 링크 추출 루틴의 변종으로 이미지 추출 루틴을 만들어 보자. ExtractImages 함수는 반환된 HTML 파일에서 img 태그의 소스(src)를 찾는다. 그리고 TLinkCallback과 호환되는 또 다른 익명 메서드를 호출한다. 이제 여러분은 이런 구상을 할 수 있다. HTML 페이지를 열고 (AjaxCall 함수를 사용한다), 그 안의 이미지 링크들을 추출하고, AjaxCall을 다시 사용해 실제 이미지들을 가져온다. 이것은 삼중으로 [triple-nested](#) 클로저를 사용한다는 뜻이다. 그것도 오브젝트 파스칼 프로그래머에게 매우 흔치 않은 코딩 구조 안에서 말이다. 하지만, 이 코드는 매우 강력하다. 그리고 표현력이 높다:

```
procedure TFormAnonAjax.BtnImagesClick(Sender: TObject);
var
  NHit: Integer;
begin
  NHit := 0;
  AjaxCall(EdUrl.Text,
    procedure(AResponseContent: TStringStream)
    begin
      ExtractImages(AResponseContent.DataString,
        procedure(const AUrl: string)
        begin
          Inc(NHit);
          Memo1.Lines.Add(IntToStr(NHit) + ' ' +
            AUrl + ' in ' + EdUrl.Text);
          if NHit = 1 then // 첫 결과를 적재
          begin
            var RealURL := IfThen(AURL[1]='/',
              EdUrl.Text + AURL, AURL); // URL을 확장
            AjaxCall(RealUrl,
              procedure(AResponseContent: TStringStream)
              begin
                Image1.Picture.Graphic.
                  LoadFromStream(AResponseContent);
              end);
          end;
        end);
    end);
  end;
```

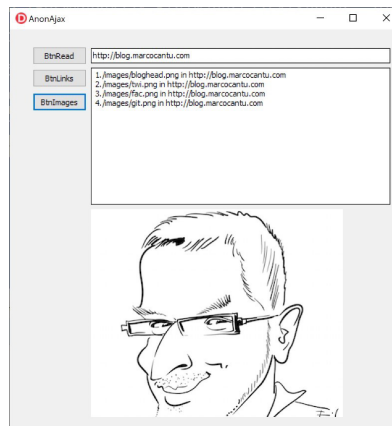

참고 이 코드 조각은 내 블로그 게시글의 주제였다. “Anonymous, Anonymous, Anonymous” 라는 2008년 9월의 글이며, 댓글도 조금 있다: https://blog.marcocantu.com/blog/anonymous_3.html.

그래픽이 Image 컴포넌트 안에 있는 것과 형식 `format`이 같은 파일을 적재할 경우에만 작동하긴 하나, 이 코드와 그 결과는 둘 다 인상적이다. 특히 숫자의 나열을 잘 보자. `NHit` 로컬 변수를 잡아 두고 수행한다. 만일 버튼을 연속으로 두 번 빠르게 누르면 이 순번 나열에 무슨 일이 생길까? 각 익명 메서드는 카운터인 `NHit`에 대해 저마다 서로 다른 복사본을 얻는다. 아마 목록 순서를 벗어나서 표시될 가능성이 많다. 또한 아마 두 번째 쓰레드가 그 출력을 내보내는 것이 첫 쓰레드보다 더 먼저일 것이다.

이렇게 잠재적 문제가 있긴 하다. 어쨌든 마지막 버튼을 사용하면 이미지가 표현된다. 그림 15.1은 그 결과를 보여준다.

그림 15.1:

웹 페이지에서 이미지를 가져오는 삼중
익명 메서드 호출의 출력



16: 리플렉션Reflection과 애트리뷰트Attribute

전통적으로 파스칼과 같이 타입이 엄격하고 정적인 언어의 컴파일러는, 존재하는 타입에 대한 정보를 런타임에 거의 또는 아예 주지 않는다. 데이터 타입에 대한 모든 정보는 컴파일 과정에서만 볼 수 있다.

이 전통을 오브젝트 파스칼의 첫 버전이 깼다. 즉, 프로퍼티들과 기타 클래스 멤버들에 대한 정보를 런타임에 제공했다. 단, `published` 라는 특정 컴파일러 지시어가 표시된 것들에 대해서다. 이 기능을 활성화하려면 해당 클래스들에 `{SM+}`라는 특정 설정을 붙여 컴파일해야 한다. 이 기능은 스트리밍streaming 메커니즘의 기초다. 그리고 VCL의 DFM 파일(그리고 FireMonkey 라이브러리에서는 FMX 파일)의 밑바탕이다. 덕분에 여러분은 폼 그리고 기타 비주얼 디자이너들을 가지고 작업을 할 수 있다.

델파이 1이 처음 만들어졌을 때, 이 기능은 완전히 새로운 아이디어였다. 그 이후에 다른 개발 도구들도 이 기능을 도입하고 확장하여 구축했다.

처음에는, 타입 시스템에 대한 확장들이 있었다(오직 오브젝트 파스칼로만). COM 안에 있는 메서드들을 탐색하고 동적 호출을 하기 위한 것들이었다. 이것은 오브젝트 파스칼에서 디스패치dispatch ID에 의해 여전히 지원된다. 메서드들을 배리언트 등 COM 관련 다른 기능들에 적용할 수 있다. 결국에는, 오브젝트 파스칼의 COM 지원이 이 언어의 고유한 런타임 타입 정보runtime type information, RTTI로 확장되었다. 하지만, 그 주제는 이 책의 범위 밖이다.

Java 그리고 .NET과 같은 관리되는managed 환경이 출현하면서, 매우 광범위한 형태의 런타임 타입 정보가 전면에 나서게 되었다. 상세한 RTTI를 컴파일러가 실행 모듈executable module들에게 묶어서 제공했다. 그래서 그 모듈들을 사용하는 프로그램에서 찾을 수 있도록 했다. 이는 단점도 있다. 프로그램 내부의 일부를 노출한다. 그리고 모듈의 크기가 커진다. 하지만, 이 기능은 새로운 프로그래밍 모델 즉 동적 언어들의 유연성을

견고한 구조와 타입에 엄격한 언어들의 속도와 합친 방식을 같이 가져왔다.

좋은 선택(실제로 이 기능이 도입될 당시 격렬한 논쟁의 주제였다), 오브젝트 파스칼은 천천히 그런 방향으로 변하고 있다. 그리고 RTTI의 광범위한 형태를 채택하는 것은 이 개발 방향에서 매우 중요한 단계다. 이제부터 보겠지만, 우리는 RTTI를 사용하지 않도록 할 수도 있다. 하지만, 하지만, 사용한다면 여러분은 여러분의 애플리케이션 안에서 추가 능력을 활용할 수 있게 된다.

이 주제는 그리 간단하지 않다. 그러므로, 한 단계씩 진행하겠다.

- 처음엔, 우리는 (컴파일러 안에 구축된) 새 확장된 `extended` RTTI 그리고 그 `Rtti` 유닛 안에 있는 새 클래스들(여러분이 사용하고 둘러볼 수 있다)에 집중한다.
- 두번째로, 새 `TValue`의 구조 그리고 동적으로 불러내기에 대해 본다.
- 세번째로, 사용자 지정 애트리뷰트 `custom attribute`를 소개한다. 닷넷에 있는 것과 같은 기능이다. 이를 통해 여러분은 컴파일러가 생성하는 RTTI 정보를 확장할 수 있다.

확장된 RTTI의 뒤편에 있는 이유들을 돌아보고 그 실제 사용 예제를 보는 것은 이 장의 마지막 부분에서만 진행하겠다.

확장된 RTTI `Extended RTTI`

오브젝트 파스칼 컴파일러는 기본적으로 많은 양의 확장된 RTTI 정보를 제공한다. 이 런타임 정보에는 모든 타입들 즉 클래스들, 기타 모든 사용자 정의 타입들뿐만 아니라 컴파일러에서 미리 정의한 핵심 데이터 타입들까지 해당된다. 또한 게시된 `published` 필드들과 공개 `public` 필드들, 심지어 보호된 `protected` 그리고 비공개인 `private` 요소들도 포함한다. 어떤 오브젝트든 그 내부 구조를 들여다볼 수 있으려면 이것이 필요하다.

첫 예제 `A First Example`

컴파일러가 생성하는 정보 그리고 그것에 접근하는 다양한 기법을 보기 전에, 결론으로 넘어가서 여러분이 RTTI로 할 수 있는 것들이 무엇인지 보자. 아래 예시는 매우 짧다. 그리고, 더 오래된 RTTI를 가지고도 작성할 수 있는 내용이다. 하지만, 내가 전하려는 내용을 이해하는데 도움이 될 것이다. (또한 오브젝트 파스칼 개발자라고 해서 모두가 전통적인 RTTI를 명시적으로 사용했던 것은 아니라는 점을 고려한 예문이다).

버튼을 담은 폼이 있다고 가정하자(`RttiIntro` 예제에서 발췌함). 여러분은 사용자가 클릭한 버튼 컨트롤의 `Text` 프로퍼티를 읽을 수 있다. 아래 코드와 같이 작성하면 된다:

```
uses
  Rtti;

procedure TFormRttiIntro.BtnInfoClick(Sender: TObject);
```



```

var
    Context: TRttiContext;
begin
    Show(Context.
        GetType(TButton).
        GetProperty( 'Text' ).
        GetValue(Sender).ToString());
end;

```

위 코드는 TRttiContext 레코드를 사용해서, TButton 타입에 대한 정보를 참조한다. 그 타입 정보부터 그 타입의 한 프로퍼티에 대한 RTTI 데이터까지 가져올 수 있다. 그리고 나서, 그 프로퍼티 데이터를 사용해 원하는 프로퍼티의 실제 값을 참조한다. 그 결과는 문자열로 변환한다.

작동 방식이 궁금하다면 계속 읽기 바란다. 요점은 이렇다. 이제 이 방식을 사용하면 그저 어떤 프로퍼티에 동적으로 접근할 수 있을 뿐만 아니라, 필드의 값을 읽을 수도 있다. 심지어 비공개private 필드까지도 말이다.

또한 여러분은 프로퍼티의 값을 바꿀 수도 있다(RttiIntro 예제의 두 번째 버튼 참조)

```

procedure TFormRttiIntro.BtnChangeClick(Sender: TObject);
var
    Context: TRttiContext;
    AProp: TRttiProperty;
begin
    AProp := Context.GetType(TButton).GetProperty('Text');
    AProp.SetValue(BtnChange, StringOfChar( '*', Random (10) + 1));
end;

```

이 코드는 해당 Text를 *표시가 임의의 개수만큼 나열되는 문자열로 바꿔준다. 위쪽 코드와 다른 점은 임시 로컬 변수를 만들고, 그것으로 해당 프로퍼티의 RTTI 정보를 참조한다는 점이다. 이제 우리가 뭘 할 수 있는지 알았으니, 처음으로 다시 돌아가서 컴파일러가 생성하는 확장된 RTTI 정보를 살펴보자.

컴파일러가 생성하는 정보 Compiler Generated Information

여러분이 해야 할 건 없다. 내버려 두면 컴파일러가 이 추가 정보를 여러분의 실행 프로그램(애플리케이션, 라이브러리, 패키지, 기타 등등)에 추가한다. 그저 프로젝트를 열고 컴파일만 하면 된다. 기본 설정에 따라, 컴파일러는 확장된 RTTI를 생성한다. 여기에는 모든 필드들(비공개 필드까지 포함), 공개public 또는 게시된published 메서드들(과 프로퍼티들)을 위한 정보가 담긴다.

여러분이 비공개private 필드들의 RTTI 정보를 얻게 된다는 사실이 놀라울 수도 있다. 하지만, 동적 연산dynamic operation (이진 오브젝트 직렬화binary object serialization 등)을 수행하려면 그리고 힙 안에 있는 오브젝트를 추적하려면 이 정보가 필요하다.

여러분은 확장된 RTTI 생성을 제어할 수 있다. 이는 설정 행렬표matrix를 따른다. 이

표의 한 축은 가시성^{visibility}이고 다른 축은 멤버의 유형이다. 아래 표는 그 시스템 기본 설정을 묘사하고 있다:

	필드	메서드	프로퍼티
Private	X		
Protected	X		
Public	X	X	X
Published	X	X	X

이 가시성 설정 4 개는 세트^{set} 타입이다. System 유닛에 이렇게 선언되어 있다:

```
type
TVisibilityClasses = set of (vcPrivate,
    vcProtected, vcPublic, vcPublished);
```

이 세트를 사용할 수 있도록 미리 준비된^{ready-to-use} 상수 값들이 있다. RTTI 가시성에 대한 이 기본 설정들은 TObject 그리고 거기서 파생된 다른 모든 클래스들에 적용된다.

```
const
DefaultMethodRttiVisibility = [vcPublic, vcPublished];
DefaultFieldRttiVisibility = [vcPrivate..vcPublished];
DefaultPropertyRttiVisibility = [vcPublic, vcPublished];
```

컴파일러에 의해 만들어지는 정보를 제어할 때는 새 지시어인 \$RTTI가 사용된다. 이 지시어에는 이 설정이 주어진 타입만을 위한 것인지 아니면 그 자손들까지 적용되는 것인지를 가리키는 상태(EXPLICIT나 INHERITED)가 명시된다. 그리고 이어서, 지정자^{specifier} 3 개를 명시한다. 즉, 메서드, 필드, 프로퍼티에 대한 각각의 가시성을 지정한다. System 유닛 안에 적용된 기본 설정은 다음과 같다:

```
{ $RTTI INHERIT
  METHODS(DefaultMethodRttiVisibility)
  FIELDS(DefaultFieldRttiVisibility)
  PROPERTIES(DefaultPropertyRttiVisibility) }
```

여러분이 만든 클래스들의 모든 멤버에 대한 확장 RTTI 생성을 완전히 비활성화하려면 다음과 같은 지시어를 사용한다:

```
{ $RTTI EXPLICIT METHODS([]) FIELDS([]) PROPERTIES([]) }
```

참고 (다른 컴파일러 지시어들은 가능하지만) RTTI 지시어는 유닛 선언 앞에 적을 수 없다. 왜냐하면 RTTI 지시어는 System 유닛 안에 정의된 설정에 의존하기 때문이다. 만약 유닛 선언보다 앞에 적는다면, 그다지 직관적이지 않은 내부 오류 메시지를 받게 된다. 그러니 어떤 경우에도, RTTI 지시어는 unit 문보다 뒤에 두도록 하자.

이 설정을 사용할 때 알아야 할 점이 있다. 이 설정은 여러분의 코드에만 적용된다. 그리고 완전한 제거는 불가능하다. RTL을 위한 그리고 기타 라이브러리 클래스들을 위한 RTTI 정보는 이미 DCU와 패키지 안으로 컴파일 되어 들어가 있기 때문이다.

또 알아 둘 점이 있다. \$RTTI 지시어는 게시된 [published](#) 타입을 위해 생성되는 전통적인 RTTI에 전혀 변화를 주지 않는다: 그 RTTI는 \$RTTI 지시어에 상관없이 항상 생성된다.

참고 RTTI를 처리하는 클래스들은 System.Rtti 유닛 안에 있다. 다음 소단원에서 설명한다. 이것들은 전통적인 RTTI와 PTypeInfo 구조체를 잡아챈다.

이 지시어로 할 수 있는 일은 여러분이 만든 클래스에 대해 확장 RTTI를 생성하지 않도록 하는 것이다. 또는 그와 정반대로, 여러분은 생성되는 RTTI의 양을 더 늘릴 수도 있다. 비공개 [private](#) 그리고 보호된 [protected](#) 메서드(와 프로퍼티)들을 넣어주면 된다. 원한다면 그렇게 할 수 있다 (비록 그다지 합리적이지는 않지만).

확장 RTTI 정보를 실행 파일에 추가하게 되면 확실히 파일의 크기가 커진다 (배포할 파일이 커지는 것이 주요 단점이다. 추가 로딩 시간과 메모리 흔적은 별 관련이 없다). 만일, RTTI를 여러분의 코드에서 사용하지 않기로 결정한다면, 여러분의 프로그램의 유닛에서 RTTI를 빼도 된다. 그러면 긍정적 효과를 얻을 수 있을 것이다. 하지만, RTTI는 상당히 강력한 기능이다. 이 장에서 보겠지만, 대부분의 경우 실행 파일 크기 추가를 감당할 가치가 있다.

약한 타입 링킹과 강한 타입 링킹 [Weak- and Strong-Type Linking](#)

또 어떤 다른 방법으로 프로그램의 크기를 줄일 수 있을까? 그 효과를 뚜렷하지 않아도 알아챌 수 있을 정도로 줄일 수 있는 방법이 몇 가지 있다.

실행 파일이 제공할 RTTI 정보를 평가할 때, 컴파일러가 추가한 것을 링커가 제거할 수 있음 고려하라. 기본으로, 프로그램 안으로 컴파일 되지 않은 클래스와 메서드는 확장된 RTTI를 갖지 않는다(상당히 쓸모 없다). 기본 RTTI도 마찬가지로 갖지 않다.

정반대로, 모든 확장 RTTI를 포함하고 작동하게 하려면, 코드에서 명시적으로 가리키지 않은 클래스와 메서드도 여러분이 링크 [link](#)해야 할 것이다.

두개의 컴파일러 지시어를 사용해 실행 파일에 링크할 정보를 제어할 수 있다. 첫째, \$WeakLinkRTTI 지시어다(도움말에 설명이 충분하다). 프로그램에서 쓰지 않는 타입에 대해 이것을 켜면, 그 타입과 RTTI 정보 모두 최종 실행 파일에서 제거된다.

대안으로, 여러분은 모든 타입과 그 확장된 RTTI를 포함하도록 강제할 수 있다. \$StrongLinkTypes 지시어를 사용하면 된다. 많은 프로그램에서 그 효과는 매우 크다. 거의 두 배로 프로그램 크기가 커진다.

RTTI 유닛 [The RTTI Unit](#)

모든 타입들에 대한 확장 RTTI 생성이 오브젝트 파스칼의 reflection(곧 살펴본다)을

받치는 첫 번째 기둥이라면, 두 번째 기둥은 이 정보를 쉽게 그리고 높은 수준에서 탐색할 수 있는 능력이다. 이는 System.Rtti 유닛 덕분이다. 세 번째 기둥은, 잠시 후 보겠지만, 사용자 지정 애트리뷰트 [custom attributes](#) 지원이다. 하나씩 차근차근 살펴보자.

전통적으로, 오브젝트 파스칼 애플리케이션은 System.TypeInfo 유닛의 함수들을 사용해 published 런타임 타입 정보에 접근할 수 있었다(지금도 그렇다). 이 유닛에는 저-수준 데이터 구조들과 함수들이 (모두가 포인터와 레코드를 기반으로 하여) 정의되어 있다. 이와 더불어 고-수준 루틴들이 몇 개 있어서 더 쉽게 다룰 수 있도록 해준다.

다른 한편, Rtti 유닛은 확장된 RTTI를 매우 쉽게 다룰 수 있도록 해준다. 그에 알맞은 메서드들과 프로퍼티들을 갖춘 클래스들의 세트를 제공한다. 그 다양한 오브젝트들에 접근하려면, TRttiContext 레코드 구조가 그 시작점이다. 그 안에는 사용할 수 있는 타입을 찾을 때 사용할 수 있는 메서드 4 개가 있다:

```
function GetType(ATypeInfo: Pointer): TRttiType; overload;
function GetType(AClass: TClass): TRttiType; overload;
function GetTypes: TArray<TRttiType>;
function FindType(const AQualifiedName: string): TRttiType;
```

위와 같이 여러분이 전달할 수 있는 것은 클래스, 타입에서 획득한 PTypeInfo 포인터, 전체 이름 [qualified name](#) (유닛 이름까지 붙은 타입 이름, 예: “System.TObject”)이다. 또한 타입들의 전체 목록을 뽑아 낼 수 있다. 이것들은 RTTI 타입들을 담은 하나의 배열에 정의되어 있다. 보다 정확하게는 TArray<TRttiType>로 정의되어 있다.

나열을 위해 작성한 코드의 마지막 메서드 호출이다(TypesList 예제의 코드를 줄임):

```
procedure TFormTypesList.BtnTypesListClick(Sender: TObject);
var
    AContext: TRttiContext;
    TheTypes: TArray<TRttiType>;
    AType: TRttiType;
begin
    TheTypes := AContext.GetTypes;
    for AType in TheTypes do
        if AType.IsInstance then
            Show(AType.QualifiedName);
end;
```

GetTypes 메서드는 데이터 타입들의 전체 리스트를 반환한다. 하지만, 이 프로그램은 클래스를 표현하는 타입들만 걸러 낸다. 이 유닛에는 타입을 표현하는 수많은 다른 클래스들이 십여 개 정도 더 있다.

참고 Rtti 유닛은 (TRttiInstanceType로) 클래스 타입을 “인스턴스”와 “인스턴스 타입”들로 가리킨다. 이것은 우리가 보통 실제 오브젝트를 가리킬 때 *인스턴스*라는 용어를 사용하므로 다소 혼란스럽다.

```
for AType in TheTypes do
    if AType is TRttiInstanceType then
        Show(AType.QualifiedName);
```

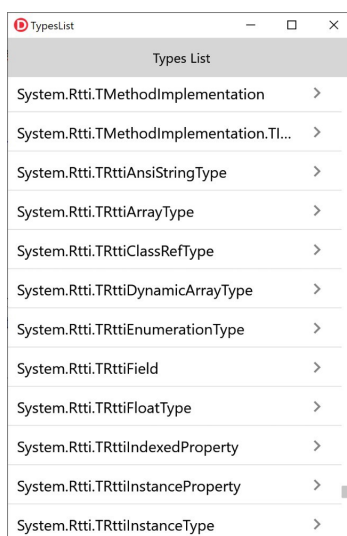

예제의 실제 코드는 조금 더 복잡하다. 먼저 스트링 리스트를 채우고, 리스트 요소를 정렬하며, 최적화를 위해 BeginUpdate와 EndUpdate를 사용해 ListView 컨트롤을 채운다. 단, 컨트롤을 다시 그리는 일은 연산이 끝날 때까지 미룬다(try-finally 블록을 사용해 항상 마무리 동작이 수행되도록 보장한다):

```
var
  AContext: TRttiContext;
  TheTypes: TArray<TRttiType>;
  SList: TStringList;
  AType: TRttiType;
  STypeName: string;
begin
  ListView1.ClearItems;
  SList := TStringList.Create;
  try
    TheTypes := AContext.GetTypes;
    for AType in TheTypes do
      if AType.IsInstance then
        SList.Add(AType.QualifiedName);
    SList.Sort;
    ListView1.BeginUpdate;
    try
      for STypeName in SList do
        (ListView1.Items.Add).Text := STypeName;
      finally
        ListView1.EndUpdate;
    end;
  finally
    SList.Free;
  end;
end;
```

이 코드는 그림 16.1과 같은 수백 개의 데이터 타입으로 된 더 긴 리스트를 만든다. 실제 숫자는 플랫폼과 컴파일러 버전에 따라 다르다. 이 그림에는 다음 소단원에서 다룰 RTTI 유닛의 타입들이 표시되고 있으니 잘 보기 바란다.

그림 16.1:

TypeList 예제의 출력



Rtti 유닛 안의 RTTI 클래스들 The RTTI Classes in the Rtti Unit

다음 목록은 그 전체 상속 그래프다. TRttiObject 추상 클래스 abstract class에서 파생된 것들이면서 System.Rtti 유닛에 정의된 것들이다:

```
TRttiObject // 추상임
  TRttiNamedObject
    TRttiType
      TRttiStructuredType // 추상임
        TRttiRecordType
        TRttiInstanceType
        TRttiInterfaceType
      TRttiOrdinalType
      TRttiEnumerationType
      TRttiInt64Type
      TRttiInvokableType
        TRttiMethodType
        TRttiProcedureType
      TRttiClassRefType
      TRttiEnumerationType
      TRttiSetType
      TRttiStringType
      TRttiAnsiStringType
      TRttiFloatType
      TRttiArrayType
      TRttiDynamicArrayType
      TRttiPointerType
    TRttiMember
      TRttiField
      TRttiProperty
        TRttiInstanceProperty
      TRttiIndexedProperty
      TRttiMethod
      TRttiParameter
      TRttiPackage
    TRttiManagedField
```

이 클래스들은 각자의 이름에 명시된 타입 named type에 대한 고유 정보를 제공한다. 예를 들어, 오직 TRttiInterfaceType만이 인터페이스 GUID에 접근하는 방법을 제공한다.

참고 Rtti 유닛이 처음 구현되었을 때는 (TStringList의 Strings[] 같은) 인덱스 되는 프로퍼티에 접근하는 RTTI 오브젝트가 없었다. 그 후에 추가되었으며 지금은 쓸 수 있다. 덕분에 런타임 타입 정보가 정말 완벽해졌다.

RTTI 오브젝트 수명 관리 RTTI Objects Lifetime Management와 TRttiContext 레코드

여러분이 앞에서 있었던 BtnTypesListClick 메서드의 소스 코드를 본다면, 뭔가 꽤 잘못된 것처럼 보이는 부분이 있을 것이다. GetTypes의 호출이 타입의 배열을 반환한다. 그런데, 그 코드에서는 이 내부 오브젝트들을 해제 free하지 않는다.

그 이유가 있다. TRttiContext 레코드 구조는 생성되는 모든 RTTI 오브젝트의 실질적인

소유자가 되기 때문이다. 그 레코드가 파기되는 시점(즉 범위를 벗어나면 *out of scope*)에는, 내부 인터페이스가 비워지면서 자신의 소멸자를 호출한다. 그것은 자신을 통해 생성된 모든 RTTI 오브젝트들을 해제 *free*한다.

사실 TRttiContext 레코드는 역할이 두 가지다. 한 측면으로, 이것은 RTTI 오브젝트들의 수명 *lifetime*을 제어한다(위에서 설명했다). 다른 하나는 RTTI 정보를 캐시 *cache*한다. 검색해서 다시 만드는 비용이 꽤 크기 때문이다. 그래서 여러분은 아마 TRttiContext 레코드에 대한 참조를 상당한 기간동안 살아있도록 유지하고 싶을 것이다. 그러면 여러분은 그것이 소유하고 있는 RTTI 오브젝트들을 다시 생성하지 않으면서 계속 접근할 수 있기 때문이다. 재생성은 시간을 잡아먹는 부분이기 때문이다.

내부에서 TRttiContext 레코드는 TRttiPool 타입의 전역 풀 *pool,보관소*을 사용한다. 이것은 크리티컬 섹션 *critical section*을 사용하여 서로 다른 스레드 간의 코드 실행을 동기화 *synchronize*한다. 그래서 그 접근은 스레드 안전 *thread safe*하다.

참고 RTTI 풀링 *pooling* 메커니즘의 스레드 안전에는 예외가 있다. 자세한 사항은 Rtti 유닛 그 자체 안에 있는 주석에 적혀있다.

즉, 좀 더 정확하게 하면, RTTI 풀은 TRttiContext 레코드들 사이에 공유된다. 따라서, 풀 안의 RTTI 오브젝트들은 TRttiContext 레코드가 최소 하나 이상 메모리에 있으면 계속 유지된다. 유닛의 주석을 인용하자면:

{... 최소 하나 이상의 컨텍스트가 살아있지 않은 RTTI 오브젝트에 작업하는 것은 오류를 일으킨다. 최소 하나의 컨텍스트를 살려두어서 이 풀의 변수를 유효하게 유지해야 한다.}

바꿔 말하면, 여러분은 RTTI 컨텍스트 *context*를 해제한 이후에도 RTTI 오브젝트를 캐시하거나 유지하는 일을 피해야 한다. 아래 예제는 메모리 접근 위반을 일으킨다 (역시 *TypesList* 예제의 일부다):

```
function GetThisType(AClass: TClass): TRttiType;
var
  AContext: TRttiContext;
begin
  Result := AContext.GetType(AClass);
end;

procedure TFormTypesList.Button1Click(Sender: TObject);
var
  AType: TRttiType;
begin
  AType := GetThisType(TForm);
  Show(AType.QualifiedName);
end;
```

요약하면, RTTI 오브젝트들은 해당 컨텍스트에 의해 관리된다. 여러분이 직접 그것들을 유지하려고 해서는 안 된다. 컨텍스트는 결국 레코드다. 따라서 자동으로 파기된다. TRttiContext를 다음과 같은 방식으로 사용하는 코드를 여러분이 보게 될 수도 있다:


```

AContext := TRttiContext.Create;
try
    // 컨텍스트 사용
finally
    AContext.Free;
end;

```

이 유사-생성자와 유사-소멸자는 내부 인터페이스(뒤편에서 사용되는 실제 데이터 구조들을 관리한다)를 nil로 설정해 폴링 메커니즘을 정리한다. 하지만, 이 동작은 로컬 타입에 대해서(예를 들어 레코드인 경우) 자동이다. 그러니 이렇게 할 필요가 없다. 여러분이 포인터를 사용해 이 컨텍스트 레코드를 다른 어느 곳에서 참조하고 있지 않다면 말이다.

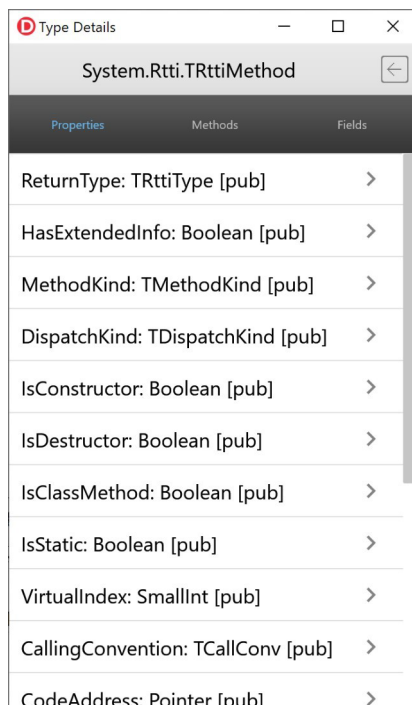
클래스 정보 표시하기 Displaying Class Information

런타임에 여러분이 검사하고 싶은 가장 관련 있는 타입은 이른바 구조화된 타입들, 즉 인스턴스, 인터페이스, 레코드다. 인스턴스에 초점을 맞춰보자. 우리는 클래스 간 관계를 참조할 수 있다. 인스턴스 타입에서 얻을 수 있는 BaseType 정보를 따라가면 된다.

타입 접근은 확실히 흥미로운 출발점이다. 하지만 관련성이 있고 특히 새로운 것은 해당 멤버 등 이 유형들의 추가 세부 정보를 알 수 있는 능력이다. 이 프로그램에서 여러분이 타입 하나를 선택해 클릭하면, 그 타입의 프로퍼티들, 메서드들, 필드들의 목록이 탭 컨트롤의 페이지 세 개에 표시된다. 그림 16.2와 같다.

그림 16.2:

TypeList 예제에서 보인
TRttiMethod 클래스를
위한 세부적인 타입 정보



이 보조 폼 유닛을 확장하거나 잘 적용하면, 다른 애플리케이션 안에서 제네릭 타입 탐색기로 사용할 수도 있을 것이다. 어쨌든, 이 유닛에는 ShowTypeInfo이라는 메서드가 있다. 이 메서드는 주어진 타입 안에 있는 프로퍼티, 메서드, 필드 각각을 살펴면서, 리스트 박스 세 개 중 해당되는 곳에 추가한다. 그러면서, 각 이름 옆에는 해당 가시성을 함께 보여준다(비공개 *pri*, 보호된 *pro*, 공개 *pub*, 게시된 *pbl*이라고 표시함). 그것들은 VisibilityToken 함수 안에 있는 간단한 case 문이 반환해준다:

```
procedure TFormTypeInfo.ShowTypeDetails(TypeName: string);
var
  AContext: TRttiContext;
  AType: TRttiType;
  AProperty: TRttiProperty;
  AMethod: TRttiMethod;
  AField: TRttiField;
begin
  AType := AContext.FindType(TypeName);
  if not Assigned(AType) then
    Exit;

  LabelType.Text := AType.QualifiedName;
  for AProperty in AType.GetProperties do
    FormTypeInfo.LVProperties.Items.Add.Text := AProperty.Name +
      ': ' + AProperty.PropertyType.Name + ' ' +
      VisibilityToken(AProperty.Visibility);
  for AMethod in AType.GetMethods do
    LVMethods.Items.Add.Text := AMethod.Name + ' ' +
      VisibilityToken(AMethod.Visibility);
  for AField in AType.GetFields do
    LVFields.Items.Add.Text := AField.Name + ': ' +
      AField.FieldType.Name + ' ' +
      VisibilityToken(AField.Visibility);
end;
```

여러분은 이 프로퍼티들의 타입으로부터 추가 정보를 추출하기, 메서드의 파라미터 목록을 가져오기, 반환 타입을 확인하기 등등 더 많은 작업을 수행할 수 있다. 완전한 RTTI 브라우저를 여기에서 구축하고 싶지는 않다. 그저 무엇을 해낼 수 있는지에 대한 느낌만 전달하고자 한다.

패키지를 위한 RTTI RTTI for Packages

여러분이 타입 또는 타입들의 목록에 접근하는 데 사용할 수 있는 메서드들 외에, TRttiContext 레코드에는 또 하나의 매우 흥미로운 메서드 즉 GetPackages가 있다. 이것은 현재 애플리케이션에서 사용하고 있는 런타임 패키지들의 목록을 반환한다. 이 메서드를 런타임 패키지가 없이 컴파일 된 애플리케이션에서 실행할 경우, 오직 실행 파일 자체만 얻게 된다. 하지만, 런타임 패키지와 함께 컴파일 된 애플리케이션에서 실행한다면, 그 패키지들의 목록을 얻을 수 있다. 이 지점에서부터, 여러분은 각 패키지가 만들어 제공하는 타입들을 파고 들어갈 수 있다. 주목할 점이 있다. 이 경우 그 타입들의 목록은 훨씬 더 길다. 왜냐하면 애플리케이션에서 사용되지 않는 RTL과

시각적 라이브러리 타입들을 스마트 링커가 제거하지 않았기 때문이다.

여러분이 런타임 패키지를 사용한다면, 패키지(및 실행 파일 자체) 각각에 대한 타입들의 목록을 추출할 수도 있다. 다음과 같은 코드를 사용하면 된다:

```
var
  AContext: TRttiContext;
  APackage: TRttiPackage;
  AType: TRttiType;
begin
  for APackage in AContext.GetPackages do
  begin
    ListBox1.Items.Add( 'PACKAGE ' + APackage.Name);
    for AType in APackage.GetTypes do
    if AType.IsInstance then
    begin
      // show type name with spaces for indentation
      ListBox1.Items.Add( ' - ' + AType.QualifiedName);
    end;
  end;
end;
```

참고 오브젝트 파스칼의 패키지는 컴포넌트를 개발 환경에 추가하는 용도로 쓸 수 있다 (11장에서 봤다). 그런데, 패키지는 런타임에도 사용될 수 있다. 즉 매우 큰 단독 실행 파일 대신 런타임 패키지들 몇 개와 주 실행파일을 함께 배포할 수도 있다. 윈도우 개발에 익숙하다면 패키지는 DLL과 비슷한 역할을 한다(기술적으로는 DLL이다). 더 정확하게는 .NET 어셈블리들과 유사한 역할이다. 패키지는 윈도우에서 매우 중요한 역할을 한다. 하지만 모바일 플랫폼에는 현재 지원되지 않는다(이 역시 iOS 등 운영체제 애플리케이션 배포 제한으로 인해 그렇다).

TValue의 구조 The TValue Structure

새로운 확장 RTTI를 사용하면 프로그램의 내부 구조를 탐색할 수 있을 뿐만 아니라 프로퍼티 및 필드의 값 등 특정 정보도 얻을 수 있다. TypeInfo 유닛은 GetPropValue 함수를 제공해서 일반 프로퍼티에 접근하고 그 값을 담은 배리언트 타입을 찾아준다. 그런데 이 새 Rtti 유닛은 다른 구조 [structure](#)를 사용해 타입 없는 요소를 담는다. 바로 TValue 레코드다.

TValue 레코드는 오브젝트 파스칼에서 가능한 거의 모든 데이터 타입을 저장할 수 있다. 그리고 원래의 데이터 표현을 추적해서 수행한다. 그러기 위해 그 데이터와 그 데이터의 타입을 모두 가지고 있다. 이 레코드로 할 수 있는 일은 주어진 형식 [format](#)으로 데이터를 읽고 쓰는 것이다. TValue에 정수를 적어 넣으면, 거기에서 오직 정수만 읽을 수 있다. 만약 문자열을 적어 넣으면, 다시 읽을 수 있는 것도 문자열이다.

할 수 없는 일도 있다. 한 형식에서 다른 형식으로 변환하지 못한다. 그래서 TValue에 비록 AsString와 AsInteger메서드가 있지만, 전자는 오직 그 데이터가 실제로 문자열을 나타낼 때에만 사용할 수 있다. 후자는 정수가 대입되어 있을 때만 쓸 수 있다. 예를

들어, 아래와 같이 여러분은 AsInteger 메서드를 사용할 수 있다. 그리고 IsOrdinal 메서드 호출은 True 값을 반환한다:

```
var
  V1: TValue;
begin
  V1 := 100;
  if V1.IsOrdinal then
    Log(IntToStr(V1.AsInteger));
```

하지만, 여러분은 AsString 메서드를 쓸 수 없다. 그러면, *invalid typecast* 예외가 생긴다:

```
var
  V1: TValue;
begin
  V1 := 100;
  Log(V1.AsString);
```

만약 문자열 표현이 필요하다면, ToString 메서드를 사용하면 된다. 이 메서드 안에는 많은 case 문들이 있다. 그래서 매우 많은 데이터 타입들을 다룰 수 있다:

```
var
  V1: TValue;
begin
  V1 := 100;
  Log(V1.ToString);
```

과거 엠바카데로 R&D 팀 멤버로 RTTI를 작업했던 Barry Kelly의 말을 읽어보면, 더 잘 이해할 수 있을 것이다:

TValue는 메서드들을 RTTI-기반으로 호출할 때 주고받은 값들을 마샬링 marshal (원활하게 이동), 그리고 필드와 프로퍼티에 대한 읽기와 쓰기를 하는 데 사용되는 타입입니다. Variant와 다소 유사하지만 Object Pascal 타입 시스템에 훨씬 더 적합합니다. 예를 들어 인스턴스가 직접 담길 수 있습니다. 세트, 클래스 참조 등등도 그렇습니다. 또한 더 타입에 엄격합니다. 따라서 (예를 들어) 문자열을 숫자로 조용히 변환하지 않습니다.

이것의 역할을 더 잘 알게 되었을 것이다. 이제, TValue 레코드의 실제 능력을 보자. 이것은 더 고-수준 메서드들의 묶음을 가고 있다. 실제 값을 대입하고 추출하는 것들이다. 또한, 저-수준인 포인터에 기반한 메서드들도 있다. 그 중 첫 번째 그룹에 하겠다. 값을 대입할 수 있도록, TValue는 여러 Implicit 연산자들을 정의한다. 그래서 여러분은 직접 대입을 수행할 수 있다. 위 코드 조각에서도 그렇게 했다:

```
class operator Implicit(const Value: string): TValue;
class operator Implicit(Value: Integer): TValue;
class operator Implicit(Value: Extended): TValue;
class operator Implicit(Value: Int64): TValue;
class operator Implicit(Value: TObject): TValue;
class operator Implicit(Value: TClass): TValue;
class operator Implicit(Value: Boolean): TValue;
```


위의 모든 연산자들이 하는 일은 제네릭 클래스 메서드인 `From`을 호출하는 것이다:

```
class function From<T>(const Value: T): TValue; static;
```

이 클래스 함수를 호출할 때는, 데이터 타입을 명시하고, 이어서 그 타입에 해당하는 값을 전달하면 된다. 아래 코드를 사용하면, 앞에서 정수 100 값을 대입하던 코드를 대체할 수 있다:

```
V1 := TValue.From<Integer>(100);
```

이것은 `TValue`에 어떤 데이터 타입이든지 대입하게 하는 보편적인 기법이다. 데이터가 대입되면, 여러분은 여러 메서드들을 사용해 그 데이터의 타입을 테스트할 수 있다:

```
property Kind: TTypeKind read GetTypeKind;
function IsObject: Boolean;
function IsClass: Boolean;
function IsOrdinal: Boolean;
function IsType<T>: Boolean; overload;
function IsArray: Boolean;
```

위 제네릭 `IsType`은 거의 모든 데이터 타입을 대상으로 사용할 수 있음을 알아 두자.

아래와 같이, 해당 타입에 맞게 데이터를 추출하는 메서드들이 있다. 다시 말하지만, 이 메서드들은 `TValue` 안에 실제로 저장된 데이터와 타입 호환이 가능한 경우에만 쓸 수 있다. 변환이 전혀 수행되지 않기 때문이다:

```
function AsObject: TObject;
function AsClass: TClass;
function AsOrdinal: Int64;
function AsType<T>: T;
function AsInteger: Integer;
function AsBoolean: Boolean;
function AsExtended: Extended;
function AsInt64: Int64;
function AsInterface: IInterface;
function AsString: string;
function AsVariant: Variant;
function AsCurrency: Currency;
```

위 메서드들 중에는 두 가지 버전을 가진 것들도 있다. 즉 추가로 `Try` 버전을 가지고 있다. 그 목적은 데이터 타입이 호환되지 않을 경우 예외를 발생시키는 대신 `False`를 반환하기 위해서다. 일부 제한적인 변환^{conversion} 메서드들도 있다. 그 중에서 가장 많이 쓰이는 것은 제네릭 `Cast`와 `ToString` 함수다. `ToString`은 이미 코드에서 쓴 적이 있다:

```
function Cast<T>: TValue; overload;
function ToString: string;
```

TValue를 사용해 프로퍼티를 읽기 Reading a Property with TValue

`TValue`의 중요한 이유가 있다. `TValue`는 확장된 RTTI와 RTTI 유닛을 사용해 프로퍼티

와 필드의 값에 접근하는 데 사용되는 구조다. TValue를 사용하는 실제 예제를 보자. 우리는 이 레코드 타입을 TButton 오브젝트의 게시된 [published](#) 프로퍼티와 비공개 [private](#) 필드에 접근하는 데 사용할 수 있다. 다음 코드와 같다 (RttiAccess 예제의 일부임):

```
var
  Context: TRttiContext;
  AType: TRttiType;
  AProperty: TRttiProperty;
  AValue: TValue;
  AField: TRttiField;
begin
  AType := Context.GetType(TButton);
  AProperty := AType.GetProperty( 'Text' );
  AValue := AProperty.GetValue(Sender);
  Show(AValue.AsString);

  AField := AType.GetField( 'FDesignInfo' );
  AValue := AField.GetValue(Sender);
  Show(AValue.AsInteger.ToString);
end;
```

메서드 불러내기 [Invoking Methods](#)

새로운 확장된 RTTI를 이용하면, 값과 필드에 대한 접근뿐만 아니라, 메서드 호출 방식도 더 간단하게 할 수 있다. 이 경우 여러분은 메서드의 각 파라미터마다 TValue 요소를 정의해야 한다. Invoke라는 전역 함수가 있다. 이것은 여러분이 메서드를 실행하기 위해 호출할 수 있다:

```
function Invoke(CodeAddress: Pointer; const Args: TArray<TValue>;
  CallingConvention: TCallConv; AResultType: PTypeInfo): TValue;
```

더 나은 대안으로, 오버로드 되어 더 간략해진 Invoke 메서드가 TRttiMethod 클래스 안에 있다:

```
function Invoke(Instance: TObject;
  const Args: array of TValue): TValue; overload;
```

이 두 번째 간략한 형태를 사용해 메서드를 호출하는 두 예제를 보자 (하나는 값을 반환하고 그 다음 예제는 파라미터를 요구한다). 이 코드는 RttiAccess 예제의 일부다:

```
var
  Context: TRttiContext;
  AType: TRttiType;
  AMethod: TRttiMethod;
  TheValues: array of TValue;
  AValue: TValue;
begin
  AType := context.GetType(TButton);
  AMethod := AType.GetMethod( 'ToString' );
  TheValues := [];
  AValue := AMethod.Invoke(Sender, TheValues);
  Show(AValue.AsString);
```



```

AType := Context.GetType(TForm1);
AMethod := AType.GetMethod( 'Show' );
SetLength(TheValues, 1);
TheValues[0] := AValue;
AMethod.Invoke(Self, TheValues);
end;

```

애트리뷰트 사용하기 Using Attributes

이 장의 맨 앞에서, (오브젝트 파스칼 컴파일러가 생성하는) 확장된 RTTI에 대해 그리고 (새 Rtti 유닛에서 도입된) RTTI의 접근 능력에 대해 이해했을 것이다. 이제 이 장의 두 번째 부분에서 이 전체 아키텍처가 도입된 핵심적인 이유를 보자. 즉, 사용자 지정 애트리뷰트 [attribute](#)를 정의하여, 컴파일러에 의해 생성되는 RTTI를 특정한 방법으로 확장할 수 있다는 점을 살펴보자. 우리는 이 기술에 대해 먼저 다소 추상적인 관점으로 살펴본다. 그리고 나서 이것이 오브젝트 파스칼을 한 단계 발전시키는데 왜 중요한지 그 이유를 집중 조명한다. 실제 예시들을 살펴보면서 진행하겠다.

애트리뷰트란? What is an Attribute?

오브젝트 파스칼과 C#의 애트리뷰트(자바 용어로는 어노테이션 [annotation](#))는 일종의 주석 또는 표시이다. 여러분은 이것을 여러분의 소스 코드에 추가할 수 있으며, 타입, 필드, 메서드, 프로퍼티에 적용하도록 할 수 있다. 컴파일러는 이것을 프로그램 안에 넣는다. 일반적으로 대괄호로 표기한다. 다음과 같다:

```

type
  [MyAttribute]
  TMyClass = class
    ...

```

이 정보는 디자인 시점에 개발 도구 안에서 읽거나, 실행 시점에 최종 애플리케이션에서 읽을 수 있으며, 읽은 값에 따라 프로그램은 동작을 바꿀 수 있다.

애트리뷰트는 오브젝트의 클래스가 가진 실제 핵심 기능 변경을 위해서는 사용되지 않는 것이 일반적이다. 오히려, 이러한 클래스들이 추가 메커니즘에 참여할 수 있도록 하는 데 사용된다. 클래스를 *직렬화 가능* [serializable](#)이라고 선언하는 것은 코드에 아무런 영향을 주지 않는다. 하지만, 직렬화 코드는 (그 애트리뷰트를 보고) 그 클래스에서 작동할 수 있다는 것을 알게 된다. 또한 어떻게 하는지 알 수도 있다 (그 애트리뷰트와 함께 제공되는 추가 정보, 즉 그 클래스의 필드 또는 프로퍼티에 추가 애트리뷰트들이 표기되어 있다면 말이다).

이것은 오브젝트 파스칼에서 원래의 RTTI 즉 제한된 RTTI가 사용되는 방식과 같다. 제한된 RTTI는 [published](#)으로 표시된 프로퍼티들은 오브젝트 인스펙터 안에 나타

나고, 스트림되어 [streamed](#) DFM 파일에 들어가고, 실행 시점에 접근하는 것도 가능하다. 애트리뷰트는 이 메커니즘을 더욱 유연하고 강력하게 만든다. 또한 훨씬 더 복잡하게 사용할 수도 있다. 그리고 잘못 사용하기도 쉽다. 다른 강력한 언어 기능들이 그렇듯이 말이다. 그저 이 기능이 있다는 이유만으로, 여러분이 이미 알고 있는 객체 지향 프로그래밍에 대한 모든 좋은 것들을 내다 버리고 이 새 모델을 포용하라는 뜻이 아니다. 오히려 서로를 보완해 주기 위해 있다.

예시를 들어, 직원 클래스는 여전히 계층 구조 상 사람 클래스에서 파생된 클래스로 표현될 수 있다(is-a). 직원 오브젝트는 여전히 그 배지에 ID를 가진다(has-a). 그런데, 여러분은 직원 클래스에 “표시 [mark](#)” 또는 “표기 [annotate](#)”를 할 수 있다. 그래서 그 클래스가 데이터베이스에 매핑되도록 또는 특정 런타임 품에서 표현되도록 할 수 있다. 즉, 우리는 상속(is-a), 소유(has-a), 표기(marked-as)라는 세 가지로 구분되는 메커니즘을 사용할 수 있으므로, 애플리케이션을 설계할 때 더 유연하다 또한 분리식 설계 [decoupled design](#)를 할 수 있다.

오브젝트 파스칼에서 사용자 지정 애트리뷰트를 지원하는 컴파일러 기능들을 보고 몇 가지 실제 예제를 본 다음에는 방금 언급한 추상적인 아이디어를 더 쉽게 이해하게 될 것이다. 그러기를 바란다!

애트리뷰트 클래스와 애트리뷰트 선언 [Attribute Classes and Attribute Declarations](#)

새로운 애트리뷰트 클래스 (또는 애트리뷰트 카테고리)를 어떻게 정의할까? 여러분은 System 유닛에 있는 새로 들어간 클래스인 TCustomAttribute를 상속받아야 한다.

```
type
  SimpleAttribute = class(TCustomAttribute)
end;
```

여러분이 애트리뷰트 클래스에 붙인 클래스 이름은 심볼이 되므로 소스 코드 안에서 사용될 수 있다. 접미사인 *Attribute*를 빼고 사용할 수도 있다. 즉, 클래스 이름을 SimpleAttribute라고 지었다면, 코드에서 여러분이 그 애트리뷰트를 사용할 때는 Simple 또는 SimpleAttribute라고 적으면 된다. 이런 이유 때문에, 오브젝트 파스칼 클래스에서 사용되는 고전적인 접두자 T는 애트리뷰트의 경우에는 대체로 사용되지 않는다. 단, 예외 있다. 바로 System.TCustomAttribute이다.

```
type
  [Simple]
  TMyClass = class(TObject)
public
  [Simple]
  procedure One;
```

위 코드는, Simple 애트리뷰트를 클래스 전체와 메서드 하나에 적용했다. 애트리뷰트는 이름 외에 하나 또는 그 이상의 파라미터를 지원한다. 애트리뷰트에 전달하는 파라미터 들은(만약 있다면) 그 애트리뷰트 클래스의 생성자에 표기된 사항들과 일치해야 한다.


```

type
  ValueAttribute = class(TCustomAttribute)
  private
    FValue: Integer;
  public
    constructor Create(N: Integer);
    property Value: Integer read FValue;
end;

```

파라미터 하나를 가지는 애트리뷰트를 적용하는 방법은 아래와 같다:

```

type
  [Value(22)]
  TMyClass = class(TObject)
  public
    [Value(0)]
    procedure Two;

```

애트리뷰트 값들은 생성자에게 전달된다. 따라서 반드시 상수 `constant` 표현식이어야 한다. 컴파일 시간에 해석되기 `resolve` 때문이다. 이런 이유로 여러분은 제한된 몇 가지 데이터 타입만 사용할 수 있다: 순서 `ordinal` 값, 문자열, 세트 `set`, 클래스 참조다. 긍정적 측면으로, 여러분은 서로 다른 파라미터를 가지는 오버로드된 생성자 여러 개를 가질 수 있다. 같은 심볼에 여러 애트리뷰트들을 적용할 수 있다는 점도 알아두자. (RttiAttrib 예제에서 발췌. 소단원의 코드 조각들을 요약하는 예제다)

```

type
  [Simple][Value(22)]
  TMyClass = class(TObject)
  public
    [Simple]
    procedure One;
    [Value(0)]
    procedure Two;
end;

```

만약 (아마 `uses` 문을 빠뜨리는 바람에) 정의되지 않은 애트리뷰트를 사용하려고 하면 어떻게 될까? 불행히도 매우 오해하기 쉬운 경고 메시지를 얻는다:

```

[DCC Warning] RttiAttribMainForm.pas(44): W1025
  Unsupported language feature: 'custom attribute'

```

이것이 경고(Warning)라는 사실에 유의하자. 그 애트리뷰트가 무시된다는 의미이다. 따라서 여러분은 이런 경고들을 잘 살펴야 한다. 다른 모든 경고들도 그렇기는 하다. 하지만, 특히 "Unsupported language feature" 경고는 오류처럼 대하는 것이 좋다 (프로젝트 옵션 대화 상자의 Hints and Warnings 페이지에서 설정해 놓을 수도 있다).

```

[DCC Error] RttiAttribMainForm.pas(38):
  E1025 Unsupported language feature: 'custom attribute'

```

마지막으로 같은 개념을 구현한 다른 것들과 비교하면, 애트리뷰트의 범위를 제한하는 방법이 현재 없다. 즉, 애트리뷰트를 타입에만 적용하고 메서드에는 적용하지 않도록

할 수 없다. 그 대신, 편집기에서 이름 바꾸기 리팩토링^{rename refactoring}이 애트리뷰트를 지원한다. 애트리뷰트 클래스의 이름을 바꿀 줄 뿐만 아니라, 그 애트리뷰트를 사용하는 곳에서 전체 이름으로 사용하거나 이름 뒷부분의 “attribute”이 없거나 모두 찾아낸다.

참고 애트리뷰트 리팩토링은 Malcolm Groves의 블로그에서 처음 언급되었다.
<http://www.malcolmgroves.com/blog/?p=554>

애트리뷰트 탐색하기 Browsing Attributes

이제 이 코드는 어떤 애트리뷰트가 정의되었는지 찾을 방법이 없으면 쓸모 없으며, 이들 애트리뷰트 때문에 오브젝트에 다른 동작을 삽입할 가능성이 있다. 첫번째 부분을 보면서 시작하자. Rtti 유닛의 클래스들은 어떤 심볼이 연관된 애트리뷰트가 있는지 알 수 있게 한다.

이 코드는 현재 클래스의 애트리뷰트 목록을 보여준다 (RttiAttrib 예제에서 발췌함):

```
procedure TMyClass.One;
var
  Context: TRttiContext;
  Attributes: TArray<TCustomAttribute>;
  Attrib: TCustomAttribute;
begin
  Attributes := Context.GetType(ClassType).GetAttributes;
  for Attrib in Attributes do
    Form39.Log(Attrib.ClassName);
```

이 코드를 실행하면 다음과 같이 출력된다:

```
SimpleAttribute
ValueAttribute
```

여러분은 다음 코드를 for-in 루프에 추가하여 주어진 애트리뷰트 타입의 특정 값을 추출하도록 확장할 수 있다:

```
if Attrib is ValueAttribute then
  Form39.Show(' - ' + IntToStr(ValueAttribute(Attrib).Value));
```

주어진 애트리뷰트, 혹은 아무 애트리뷰트가 붙어 있는 메서드를 가져오는 건 어떨까? 여러분은 미리 메서드를 걸러 낼 수 없다. 각 메서드를 살펴 그 애트리뷰트 값을 보고 여러분과 관련이 있는 애트리뷰트인지 확인해야 한다. 그 과정에 도움이 되도록, 어떤 메서드가 주어진 애트리뷰트를 지원하는지 확인하는 함수를 작성했다:

```
type
  TCustomAttributeClass = class of TCustomAttribute;

function HasAttribute(AMethod: TRttiMethod;
  AttribClass: TCustomAttributeClass): Boolean;
var
  Attributes: TArray<TCustomAttribute>;
  Attrib: TCustomAttribute;
```



```

begin
  Result := False;
  Attributes := AMethod.GetAttributes;
  for Attrib in Attributes do
    if Attrib.InheritsFrom(AttribClass) then
      Exit(True);
end;

```

HasAttribute 함수는 RttiAttrib 프로그램에서 호출한다. 그래서 특정 Simple 애트리뷰트가 있는지 확인한다. 또한 발견한 다른 모든 애트리뷰트들도 나열한다:

```

var
  Context: TRttiContext;
  AType: TRttiType;
  AMethod: TRttiMethod;
begin
  AType := Context.GetType(TMyClass);

  Log('Methods marked with [Simple] attribute');
  for AMethod in AType.GetMethods do
    if HasAttribute(AMethod, SimpleAttribute) then
      Show(AMethod.Name);

  Log('');
  Log('Methods marked with any attribute');
  for AMethod in AType.GetMethods do
    if HasAttribute(AMethod, TCustomAttribute) then
      Show(AMethod.Name);

```

그 효과로, 주어진 애트리뷰트, 혹은 아무 애트리뷰트가 붙어 있는 메서드를 나열한다:

```

Methods marked with [Simple] attribute
One

Methods marked with any attribute
One
Two

```

이런 단순한 애트리뷰트 설명이 아니라, 여러분이 일반적으로 수행하는 작업은 몇 가지 독립적인 동작들을 추가하고, 그 동작에 대한 결정이 (실제 코드에 의해서가 아니라) 클래스 애트리뷰트에 의해서 되도록 하는 것일 것이다. 그 예시로, 이전 코드에 특정 동작을 삽입해보자: 그 목표는 클래스의 메서드를 중에서 주어진 애트리뷰트 표시가 붙은 것들을 모두 호출하는 것이다. 파라미터가 없는 메서드들이라고 가정하고 해보자:

```

procedure TForm39.BtnInvokeIfZeroClick(Sender: TObject);
var
  Context: TRttiContext;
  AType: TRttiType;
  AMethod: TRttiMethod;
  ATarget: TMyClass;
  ZeroParams: array of TValue;
begin
  ATarget := TMyClass.Create;
  try
    AType := Context.GetType(ATarget.ClassType);

```



```

    for AMethod in AType.GetMethods do
        if HasAttribute(AMethod, SimpleAttribute) then
            AMethod.Invoke(ATarget, ZeroParams);
        finally
            ATarget.Free;
        end;
    end;
end;

```

이 코드 조각이 수행하는 작업은 다음과 같다. 오브젝트를 생성하고, 그것의 타입을 파악하고, 지정된 애트리뷰트를 확인하고, Simple 애트리뷰트가 붙어 있는 각 메서드를 호출한다. 여러분이 기반 클래스에서 상속하거나, 인터페이스를 구현하거나, 요청을 수행하기 위한 특정 코드를 작성하는 대신, 이와 같이 주어진 애트리뷰트를 사용해 여러 메서드 중 하나에 표시하기만 하면 새로운 동작을 얻을 수 있다. 이 예제만이 애트리뷰트 사용을 매우 확실하게 보여주는 것은 아니다. 애트리뷰트를 사용하는 몇 가지 흔한 패턴들과 실제 경우들에 대한 탐구를 여러분은 이 장의 마지막 부분에서 참고할 수 있다.

가상 메서드 인터셉터 Virtual Method Interceptors

이 소단원은 매우 고급 단계의 오브젝트 파스칼 기능을 다루며 여러분은 처음 델파이 언어를 배운다면 이 부분을 건너뛸 수 있다. 여기는 더 전문적인 독자를 위한 부분이다.

확장된 RTTI가 도입된 후에 추가된 또 다른 관련 기능이 있다. 기존 클래스의 가상 메서드 실행을 가로챌 수 있는 *intercept* 기능이다. 기존 오브젝트를 위한 프록시 클래스 *proxy class*를 생성하여 가로채기를 한다. 다른 말로 하면, 여러분은 기존 오브젝트를 가져온 다음 그 가상 메서드를 변경할 수 있다 (메서드 하나만 또는 한꺼번에 모두를 바꿀 수 있다).

그렇게 하는 이유가 뭘까? 표준 오브젝트 파스칼 애플리케이션에서, 여러분은 아마도 이 기능을 쓰지 않을 것이다. 오브젝트에 다른 동작이 필요하다면, 그것을 바꾸거나 하위 클래스를 생성하면 된다. 그런데, 라이브러리에서는 상황이 다르다. 라이브러리는 매우 일반적인 방법으로 작성되어야 하기 때문이다. 라이브러리들은 자신들이 다루게 될 오브젝트에 대해 아는 것이 거의 없다. 그리고 그 오브젝트들 자체에는 가능한 적게 부담을 줘야 한다. 이와 같은 상황을 다루기 위해 가상 메서드 인터셉터가 오브젝트 파스칼에서 추가되었다.

참고 Barry Kelly의 가상 메서드 인터셉터에 대한 자세한 블로그 글이 여기 있다 (내가 많이 인용했다). <http://blog.barrkel.com/2010/09/virtual-method-interception.html>

가능한 상황들에 집중하기 전에, 이 기술 자체를 살펴보자. 가상 메서드를 최소한 하나 이상 가지고 있는 기존 클래스를 가정하자. 다음과 같다:

type


```

TPerson = class
...
public
  property Name: string read FName write SetName;
  property Birthdate: TDate read FBirthdate write SetBirthdate;
  function Age: Integer; virtual;
  function ToString: string; override;
end;

function TPerson.Age: Integer;
begin
  Result := YearsBetween(Date, FBirthdate);
end;

function TPerson.ToString: string;
begin
  Result := FName + ' is ' + IntToStr(Age) + ' years old';
end;

```

이 클래스의 오브젝트인 FPerson1을 여러분이 가지고 있다고 가정하자. 이제 여러분은 TVirtualMethodInterceptor(RTTI 유닛 안에 정의된 새 클래스임)의 오브젝트를 만들고, 여러분이 원하는 하위 클래스(TPerson)에 묶을 수 있다. 그러면, FPerson1 오브젝트의 정적 클래스는 이제 이 새 동적 클래스로 바뀐다:

```

var
  FVmi: TVirtualMethodInterceptor;
begin
  FVmi := TVirtualMethodInterceptor.Create(TPerson);
  FVmi.Proxify(FPerson1);

```

FVmi 오브젝트가 있으니, 이제 여러분은 그 이벤트 (OnBefore, OnAfter, OnException)에 대한 특별한 핸들러를 익명 메서드로 장착^{install}할 수 있다. 이것들은 발동하는 시점은 모든 가상 메서드 호출 전, 모든 가상 메서드 호출 후, 가상 메서드에 예외가 발생시이다. 아래에서 이 세 익명 메서드 타입의 서명^{signature}을 볼 수 있다:

```

TInterceptBeforeNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out DoInvoke: Boolean;
  out Result: TValue);
TInterceptAfterNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; var Result: TValue);
TInterceptExceptionNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out RaiseException: Boolean;
  TheException: Exception; out Result: TValue);

```

각 이벤트마다 여러분은 그 오브젝트, 그 메서드 참조, 그 파라미터, 그 결과 (이미 설정되었거나 아닐 것이다)를 얻는다. OnBefore 이벤트 안에서는 여러분이 DoInvoke 파라미터를 설정해서 해당 표준 실행을 비활성화 할 수도 있다. 한편 OnExcept 이벤트에서는 예외에 대한 정보를 얻을 수 있다.

InterceptBaseClass 예제는 위 TPerson 클래스를 사용한다. 거기에서 나는 그 클래스 가상 메서드를 가로채서 아래의 로그 기록 [logging](#) 코드를 반영했다:

```
procedure TFormIntercept.BtnInterceptClick(Sender: TObject);
begin
    FVmi := TVirtualMethodInterceptor.Create(TPerson);
    FVmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
        const Args: TArray<TValue>; out DoInvoke: Boolean;
        out Result: TValue)
        begin
            Show( 'Before calling ' + Method.Name);
        end;
    FVmi.OnAfter := procedure(Instance: TObject; Method: TRttiMethod;
        const Args: TArray<TValue>; var Result: TValue)
        begin
            Show( 'After calling ' + Method.Name);
        end;
    FVmi.Proxify(FPerson1);
end;
```

주의할 점이 있다. FVmi 오브젝트는 적어도 FPerson1 오브젝트가 사용될 때까지 유지되어야 한다. 그렇지 않으면 여러분은 더 이상 사용할 수 없는 동적 클래스를 사용하는 것이 된다. 따라서, 이미 해제된 익명 메서드를 호출하는 것이 된다. 이 예제에서는, 그것을 품의 필드(FVmi)로 저장했다. 그것이 가리키는 오브젝트(FPerson1)도 그랬다.

프로그램은 그 오브젝트를 사용한다. 즉 메서드들을 호출하고 기반 클래스의 이름을 확인한다:

```
Show( 'Age: ' + IntToStr(FPerson1.Age));
Show( 'Person: ' + FPerson1.ToString);
Show( 'Class: ' + FPerson1.ClassName);
Show( 'Base Class: ' + FPerson1.ClassParent.ClassName);
```

인터셉터를 장착하기 전에는, 그 출력이 다음과 같다:

```
Age: 26
Person: Mark is 26 years old
Class: TPerson
Base Class: TObject
```

인터셉터를 장착한 후에는, 그 출력이 다음과 같이 된다:

```
Before calling Age
After calling Age
Age: 26
Before calling ToString
Before calling Age
After calling Age
After calling ToString
Person: Mark is 26 years old
Class: TPerson
Base Class: TPerson
```


이 클래스의 이름이 자신의 기반 클래스의 이름과 같다는 것을 눈여겨보자. 이름은 같지만 사실 다른 클래스다. 이 동적 클래스는 가상 메서드 인터셉터가 만들어낸 것이다.

생성된 오브젝트의 클래스를 원래 클래스로 되돌릴 방법은 공식적으로 없다. 하지만, 그 원래 클래스는 FVmi 오브젝트 안에 여전히 있다. 즉 그 오브젝트의 기반 클래스로 남아있다. 따라서, 여전히 여러분은 *강제*로 그 오브젝트의 원래 클래스 데이터(맨 앞에 있는 4 바이트) 안에, 올바른 클래스 참조를 대입할 수도 있다.

```
PPointer(FPerson1)^ := FVmi.OriginalClass;
```

더 나아가, OnBefore 코드를 수정했다. 그래서 여러분이 Age를 호출하면, 주어진 값을 반환하고 실제 메서드 실행을 건너뛴다:

```
FVmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out DoInvoke: Boolean;
  out Result: TValue)
begin
  Show('Before calling ' + Method.Name);
  if Method.Name = 'Age' then
    begin
      Result := 33;
      DoInvoke := False;
    end;
  end;
```

출력은 다음과 같이 변한다(Age가 호출되는 것과 그와 관련된 OnAfter 이벤트를 건너뛰는 것을 주목하라):

```
Before calling Age
Age: 33
Before calling ToString
Before calling Age
After calling ToString
Person: Mark is 33 years old
Class: TPerson
Base Class: TPerson
```

이제 가상 메서드 인터셉터 뒤편의 기술적 세부 사항을 봤다. 이제 우리는 다시 돌아가서 어떤 상황에서 이 기능을 사용하고 싶을지 알아보자.

다시 말하지만, 기본적으로 표준 애플리케이션에서는 이 기능을 쓸 이유가 없다. 주 대상은 고급 라이브러리를 개발하고, 테스트 혹은 오브젝트 처리를 위해 사용자 지정 동작을 구현하려는 사람들이다.

예를 들어, 유닛 테스트 라이브러리를 만드는 데 도움이 된다. 비록 가상 메서드들만 테스트할 수 있겠지만 말이다. 또한 여러분은 이것을 사용자 지정 애트리뷰트와 함께 사용해서 관점 지향([aspect-oriented](#)) 프로그래밍과 유사한 코딩 스타일을 구현할 수도 있다.

RTTI 사례 연구 RTTI Case Studies

이제 RTTI의 기초와 애트리뷰트의 사용을 다루었다. 그러므로 이 기법이 유용하다는 것을 증명할 몇 가지 실제 세계 상황을 볼 가치가 있다. 더 유연한 RTTI와 애트리뷰트를 통해 사용자가 지정할 수 있는 능력이 관련된 많은 상황이 있으나, 그 상황의 긴 목록을 나열할 지면이 없다. 그 대신 수행할 수 있는 일로 두 개의 간단하지만 중요한 예제를 단계별로 개발하는 방법을 안내하겠다.

첫 예제 프로그램은 클래스 내 특정 정보를 식별하기 위해 애트리뷰트를 사용하는 방법을 보인다. 특히 우리는 아키텍처^{architecture},설계^{설계}의 일부라고 선언하고 오브젝트 자체를 참조하는 고유 ID와 설명을 갖고 있는 클래스의 오브젝트를 검사할 수 있기를 원한다. 이것은 (제네릭이든 전통적인 것이든) 컬렉션 내의 오브젝트를 설명할 때처럼 여러 상황에서 유용할 수 있다.

두 번째 예제는 스트리밍의 예제로, 구체적으로 클래스를 XML 파일로 스트리밍하는 것이다. published로 게시된 RTTI을 사용하는 고전적인 접근법에서 시작하여 새로운 확장된 RTTI로 이동할 것이며, 마지막으로 어떻게 애트리뷰트를 통해 코드를 변경하고 더 유연하게 만드는지 보일 것이다.

ID와 설명을 위한 애트리뷰트들 Attributes for ID and Description

만약 많은 오브젝트 사이에 공유된 여러 메서드를 가지고 싶다면, 고전적인 방법은 가상 메서드가 있는 기반 클래스를 정의하고 거기서 다양한 오브젝트가 상속을 받아 가상 메서드를 오버라이드 해야 했다. 이것은 좋은 방법이지만, 고정된 기반 클래스를 가질 때 그 아키텍처^{architecture}에 들어갈 수 있는 클래스들의 관점에서 많은 제한을 부과한다.

이 상황을 극복할 수 있는 표준 기법은 공통 기반 클래스 대신 인터페이스를 사용하는 것이다. (공통 조상 클래스는 없지만) 인터페이스를 구현하는 여러 클래스들은 가상 메서드와 매우 비슷하게 작동하는 인터페이스 메서드의 구현을 제공한다.

(장점과 단점 모두) 완전히 다른 스타일은 들어갈 수 있는 클래스와 주어진 클래스(혹은 프로퍼티)를 표시하는 애트리뷰트를 사용하는 것이다. 이것은 더 많은 유연함을 주며 인터페이스가 관여하지 않지만, 컴파일 시간의 해결^{resolution}보다 비교적 느리고 오류가 발생하기 쉬운 런타임 정보 조회^{look-up} 프로세스를 기반으로 한다. 즉, 인터페이스보다 이 코딩 스타일을 더 나은 접근 방식으로 옹호하지 않으며, 다만 평가할 가치가 있고 사용하기에 흥미로울 수 있는 몇몇 상황에서만 사용할 것을 권한다.

설명 애트리뷰트의 클래스 The Description Attribute Class

이 예시를 위해, 적용될 요소를 가리키는 설정^{setting}의 애트리뷰트를 정의하였다. 세 개

의 다른 애트리뷰트를 사용할 수도 있지만, 저자는 그보다 애트리뷰트 네임스페이스 namespace를 오염시키는 일을 피하고자 한다. 이것이 애트리뷰트 클래스의 정의다:

```
type
  TDescriptionAttrKind = (dakClass, dakDescription, dakId);

  DescriptionAttribute = class(TCustomAttribute)

private
  FDak: TDescriptionAttrKind;
public
  constructor Create(ADak: TDescriptionAttrKind = dakClass);
  property Kind: TDescriptionAttrKind read FDak;
end;
```

애트리뷰트를 파라미터 없이 쓰기 위해 유일한 파라미터에 기본값이 있는 생성자를 사용함에 유의하자.

예시 클래스들 [The Sample Classes](#)

그 다음 애트리뷰트를 사용하는 두 개의 예시 클래스들을 작성했다. 각 클래스는 애트리뷰트가 표시되었으며 두 개의 메서드는 같은 애트리뷰트가 다른 종류로 지정되도록 표시되었다.

첫번째 클래스(TPerson)에는 Description 애트리뷰트가 있고, 이것은 GetName 함수에 매핑되어 있다. 그리고 그것의 TObject.GetHashCode 메서드를 사용하여 임시 ID를 제공하여, 애트리뷰트를 적용하도록 하는 메서드를 재선언 [redeclare](#)한다(그 메서드의 코드는 그저 inherited 버전을 호출한다):

```
type
  [Description]
  TPerson = class
private
  FBirthdate: TDate;
  FName: string;
  FCountry: string;
  procedure SetBirthdate(const Value: TDate);
  procedure SetCountry(const Value: string);
  procedure SetName(const Value: string);
public
  [Description(dakDescription)]
  function GetName: string;
  [Description(dakID)]
  function GetStringCode: Integer;
published
  property Name: string read GetName write SetName;
  property Birthdate: TDate read FBirthdate write SetBirthdate;
  property Country: string read FCountry write SetCountry;
end;
```

두번째 클래스(TCompany)는 더 간단하며 ID를 위한 스스로의 값과 설명이 있다:


```

type
[Description]
TCompany = class
private
  FName: string;
  FCountry: string;
  FID: string;
  procedure SetName(const Value: string);
  procedure SetID(const Value: string);
public
  [Description(dakDescription)]
  function GetName: string;
  [Description(dakID)]
  function GetID: string;
published
  property Name: string read GetName write SetName;
  property Country: string read FCountry write FCountry;
  property ID: string read FID write SetID;
end;

```

두 클래스 사이에 비슷한 점이 있기는 하지만, 계층 구조, 공통 인터페이스 등의 측면에서는 전혀 관련이 없다. 이 두 클래스가 유일하게 공유하는 점은 Description이라는 같은 애트리뷰트를 사용하고 있다는 것뿐이다.

예시 프로젝트와 애트리뷰트 찾아보기 [The Sample Project and Attributes Navigation](#)

공유된 애트리뷰트는 다음과 같이 프로그램의 메인 폼에 선언된 리스트에 추가된 오브젝트의 정보를 표시하기 위해 쓴다.

```

FObjectsList: TObjectList<TObject>;

```

이 리스트는 프로그램이 시작할 때 생성되고 초기화된다:

```

procedure TFormDescrAttr.FormCreate(Sender: TObject);
var
  APerson: TPerson;
  ACompany: TCompany;
begin
  FObjectsList := TObjectList<TObject>.Create;

  // person 추가
  APerson := TPerson.Create;
  APerson.Name := 'Wiley';
  APerson.Country := 'Desert';
  APerson.BirthDate := Date - 1000;
  FObjectsList.Add(APerson);

  // company 추가
  ACompany := TCompany.Create;
  ACompany.Name := 'ACME Inc.';
  ACompany.ID := IntToStr(GetTickCount);
  ACompany.Country := 'Worldwide';
  FObjectsList.Add(ACompany);

```



```
// 관련 없는 오브젝트를 추가
FObjectsList.Add(TStringList.Create);
```

오브젝트에 대한 정보(즉, 사용 가능한 ID와 설명)를 표시하려면 프로그램은 RTTI를 통해 애트리뷰트 검색을 사용한다. 먼저, 클래스가 특정 애트리뷰트로 표시되었는지 결정하는 헬퍼 함수를 사용한다:

```
function TypeHasDescription(AType: TRttiType): Boolean;
var
  Attr: TCustomAttribute;
begin
  for Attr in AType.GetAttributes do
  begin
    if Attr is DescriptionAttribute then
      Exit(True);
    end;
    Result := False;
  end;
```

참고 이 경우에는 애트리뷰트를 적용할 때 사용하는 심볼인 “Description”이 아닌 완전한 클래스 이름, DescriptionAttribute를 확인할 필요가 있는데, 컴파일러가 대괄호 안에 쓰였을 때만 줄인 이름을 인식하기 때문이다.

이것이 그러한 경우일 때, 프로그램은 중첩된 루프(nested loop)로 각 메서드에 대한 각 애트리뷰트를 가져오며, 이것이 우리가 찾는 그 애트리뷰트인지 확인한다:

```
if TypeHasDescription(AType) then
begin
  for AMethod in AType.GetMethods do
    for Attr in AMethod.GetAttributes do
      if Attr is DescriptionAttribute then
        ...
```

이 루프의 중심에서는, 애트리뷰트가 표시된 메서드들이 호출(invoked)된다. 그 메서드들의 결과들은 임시 문자열 두 개에 넣는다(나중에 사용자 인터페이스에 추가될 것이다):

```
if Attr is DescriptionAttribute then
  case DescriptionAttribute(Attr).Kind of
    dakClass: ; // 무시
    dakDescription:
      trDescr := AMethod.Invoke(AnObject, []).ToString;
    dakId:
      strID := AMethod.Invoke(AnObject, []).ToString;
```

프로그램이 실패하는 것은 애트리뷰트가 복제될 경우(즉, 여러 메서드가 같은 애트리뷰트로 표시된 경우 예외를 발생하게 하려는 상황)를 확인하는 일이다. 이전 페이지의 모든 코드 조각을 합쳐서, 이것이 UpdateList 메서드의 완전한 코드다:

```
procedure TFormDescrAttr.UpdateList;
var
  AnObject: TObject;
  Context: TRttiContext;
```



```

AType: TRttiType;
Attrib: TCustomAttribute;
AMethod: TRttiMethod;
StrDescr, StrID: string;
begin
  for AnObject in FObjectsList do
    begin
      AType := Context.GetType(AnObject.ClassInfo);
      if TypeHasDescription(AType) then
        begin
          for AMethod in AType.GetMethods do
            for Attrib in AMethod.GetAttributes do
              if Attrib is DescriptionAttribute then
                case DescriptionAttribute(Attrib).Kind of
                  dakClass: ; // 무시
                  dakDescription:
                    // 애트리뷰트가 복제되었는지 확인
                    StrDescr := aMethod.Invoke(
                      AnObject, []).ToString;
                    dakId:
                      StrID := AMethod.Invoke(
                        AnObject, []).ToString;
                end;
                // 애트리뷰트 탐색 종료
                // 무언가 찾았는지 확인
                with ListView1.Items.Add do
                  begin
                    Text := STypeName;
                    Detail := StrDescr;
                  end;
                end;
            end;
          end;
        // 그 외에는 오브젝트를 무시하며, 예외가 발생할 수 있음
        end;
    end;
  end;
end;

```

이 프로그램의 결과가 다소 흥미롭지 않다면, 그 수행 방법과 관련이 있다. 몇몇 클래스들과 그들의 두 메서드에 애트리뷰트를 표시했기 때문에, 이들 클래스들을 외부 알고리즘으로 처리할 수 있었다.

다른 말로 하면, 이 클래스들 자체는 이제 구체적인 기반 클래스나, 인터페이스 구현, 혹은 그 아키텍처^{architecture}의 일부가 되는 어떠한 내부 코드도 필요하지 않다. 그저 애트리뷰트를 사용해서, 참여하겠다는 의사 선언^{declare}하기만 하면 된다. 그 클래스에 대한 모든 관리 책임은 ‘외부에 있는’ 몇몇 코드 안에 있다.

XML 스트리밍^{XML Streaming}

RTTI를 사용하는 한 가지 흥미롭고 매우 유용한 경우가 있다. 오브젝트의 상태를 파일에 저장하거나 유선으로 다른 애플리케이션에 보내기 위해 오브젝트의 이식 가능한 “external” 표현을 만드는 것이다. 전통적으로 이 문제에 대한 오브젝트 파스칼의 접근 방식은 DFM 파일을 생성할 때 사용된 것과 동일한 접근 방식으로 오브젝트의

published^{게시된} 프로퍼티를 스트리밍하는 것이었다. 다른 선택 가능한 접근법으로 사용자가 직접 스트리밍 프레임워크^{streaming framework}를 작성하여 오브젝트를 스트림으로 직렬화^{serialize}하거나 스트림에서 오브젝트로 비직렬화^{deserialize}를 하는 것도 있다.

이제 RTTI는 오브젝트의 실제 데이터와 그 필드를 외부 인터페이스가 아닌 방법으로 저장할 수 있게 한다. 이것은 비록 추가적인 복잡함을 이끌어내지만 더 강력하다. 사례를 들면 내부 오브젝트 데이터의 관리가 있다. 다시 말하지만, 다음 예시는 기법의 간단한 쇼케이스^{showcase}, 시범 사례이며 그것이 함축한 전체를 설명하지 않는다.

이 예제들은 간단함을 위해 하나의 프로젝트에서 컴파일 된 3개의 예제다. 첫째는 published 프로퍼티에 기반한 전통적인 오브젝트 파스칼 접근법이며, 두번째는 확장된 RTTI와 필드를 사용하고, 세번째는 데이터 매핑을 변경하기 위해 애트리뷰트를 사용한다.

사소한 XML 쓰기 클래스 The Trivial XML Writer Class

XML 생성을 할 때 도움이 되도록, XmlPersist 예제를 만들었다. 이 예제의 기반은 TTrivialXmlWriter 클래스를 확장한 버전이다. 그 클래스는 내가 TTextWriter 클래스 사용법을 보여주기 위해 Delphi 2009 Handbook에서 만들었던 것이다. 그 내용을 여기서 다루기에는 너무 길다. 지금은 그저 이 클래스가 할 수 있는 일들만 간단히 알아도 충분할 것이다. 이 클래스는 자신이 연 XML 노드들을 추적할 수 있다 (문자열들의 스택 덕분이다). 그리고 그 노드들을 LIFO(후입선출) 순서로 닫을 수 있다.

참고 Delphi 2009 Handbook에 있는, TTrivialXmlWriter 클래스의 소스 코드는, 다음 링크에 있다:
<http://github.com/MarcoDelphiBooks/Delphi2009Handbook/tree/master/07/ReaderWriter>

원래 클래스에 내가 몇 가지 제한된 형식화^{formatting} 코드 그리고 오브젝트를 저장하는 세 개의 메서드를 추가했다. 이 소단원에서 다룰 세 가지 방식을 기반으로 하는 것들이다. 전체 클래스 선언은 다음과 같다:

```
type
  TTrivialXmlWriter = class
  private
    FWriter: TTextWriter;
    FNodes: TStack<string>;
    FOwnsTextWriter: Boolean;
  public
    constructor Create(AWriter: TTextWriter); overload;
    constructor Create(AStream: TStream); overload;
    destructor Destroy; override;
    procedure WriteStartElement(const SName: string);
    procedure WriteEndElement(Indent: Boolean = False);
    procedure WriteString(const SValue: string);
    procedure WriteObjectPublished(AnObj: TObject);
    procedure WriteObjectRtti(AnObj: TObject);
    procedure WriteObjectAttrib(AnObj: TObject);
    function Indentation: string;
  end;
```


위 코드의 의도를 알려면, 아래 WriteStartElement 메서드를 보자. 이것은 위 코드의 Indentation 함수를 써서 내부 스택에 있는 현재 노드 수의 두 배만큼 공백을 추가한다:

```
procedure TTrivialXmlWriter.WriteStartElement(const SName: string);
begin
    FWriter.Write(Indentation + '<' + SName + '>');
    FNodes.Push(SName);
end;
```

이 클래스에 대한 전체 코드는 이 프로젝트의 소스 코드 안에 있다.

고전 RTTI 기반 스트리밍 Classic RTTI-Based Streaming

소개와 해당 지원 클래스 설명을 마쳤으니, 맨 처음부터 시작해보자. 즉, 오브젝트를 XML-기반 형식으로 저장한다. 먼저 고전적인 RTTI를 사용해 본다. 이것은 게시된 [published](#) 프로퍼티들을 활용한다.

WriteObjectPublished 메서드는 코드가 상당히 복잡하다. 그래서 설명을 조금 하겠다. 이것은 TypeInfo 유닛이 기반이다. 그리고 오래된 RTTI의 저-수준 버전을 사용한다. 그래서 주어진 오브젝트(AnObj 파라미터)의 게시된 [published](#) 프로퍼티들을 나열한다. 그 코드는 아래에 있다:

```
NProps := GetTypeData(AnObj.ClassInfo)^.PropCount;
GetMem(PropList, NProps * SizeOf(Pointer));
GetPropInfos(AnObj.ClassInfo, PropList);
for I := 0 to NProps - 1 do
    ...
```

위 코드가 하는 일을 보자. 프로퍼티의 개수를 얻어내고, 알맞은 크기의 데이터 구조를 할당하고, 게시된 프로퍼티들에 대한 정보를 가져와 그 데이터 구조를 채운다. 이 저-수준 코드를 여러분이 작성할 수 있을지 궁금한가? 글썄, 이것이 바로 새 RTTI가 도입된 좋은 이유다: 오래된 RTTI를 단순화하고 그 복잡함을 숨긴다.

계속 해서 아래 코드를 보자. 각 프로퍼티마다, 프로그램은 숫자 프로퍼티와 문자열 프로퍼티에서 값을 추출한다. 한편 하위 오브젝트르 추출하고 재귀적으로 동작한다:

```
StrPropName := UTF8ToString(PropList[i].Name);
case PropList[i].PropType^.Kind of
    tkInteger, tkEnumeration, tkString, tkUString, ...:
        begin
            WriteStartElement(StrPropName);
            WriteString(GetPropValue(AnObj, StrPropName));
            WriteEndElement;
        end;
    tkClass:
        begin
            InternalObject := GetObjectProp(AnObj, StrPropName);
            // 하위 클래스에 대해 재귀 동작함
            WriteStartElement(StrPropName);
            WriteObjectPublished(InternalObject as TPersistent);
            WriteEndElement(True);
        end;
end;
```


몇 가지 추가로 복잡한 것들이 있다. 하지만, 이 예제의 목적 그리고 전통적인 방식을 보여주는 정도면 충분할 것이다.

프로그램의 효과를 보여주기 위해 두 개의 클래스(TCompany와 TPerson)를 작성했다. 이전 예제를 가져와 바꾼 것인데, 이번에는 company가 person을 가질 수 있다. 그래서 Boss라는 프로퍼티를 추가로 넣었다. 실제 세상에서는 더 복잡하겠지만, 이 예제로는 합리적인 가정이다. 두 클래스의 게시된 [published](#) 프로퍼티들은 아래와 같다:

```
type
  TPerson = class(TPersistent)
  ..
  published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
  end;

  TCompany = class(TPersistent)
  ..
  published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
    property ID: string read FID write FID;
    property Boss: TPerson read FPerson write FPerson;
  end;
```

프로그램의 메인 폼에는 버튼이 있다. 이 버튼은 이 두 클래스로 오브젝트를 생성하고 서로를 연결한다. 그리고 그것들을 XML 스트림에 저장해서 이후에 표시하도록 한다. 스트리밍 부분의 코드는 다음과 같다:

```
SS := TStringStream.Create;
XmlWri := TTrivialXmlWriter.Create(SS);
XmlWri.WriteStartElement('Company');
XmlWri.WriteObjectPublished(ACompany);
XmlWri.WriteEndElement;
```

그 결과는 XML 파일이다. 다음과 같다:

```
<Company>
  <Name>ACME Inc.</Name>
  <Country>Worldwide</Country>
  <ID>29088851</ID>
  <Boss>
    <Name>Wiley</Name>
    <Country>Desert</Country>
  </Boss>
</Company>
```

확장된 RTTI로 필드를 스트리밍하기 [Streaming Fields With Extended RTTI](#)

고-수준의 RTTI를 오브젝트 파스칼에서 사용할 수 있으므로, 이 낡은 프로그램에서 게시된 [published](#) 프로퍼티들을 접근하는 부분을 확장된 RTTI를 사용해 바꿀 수도 있었을 것이다. 그 대신, 여기서는 그 오브젝트의 내부 표현, 즉 비공개 [private](#) 데이터 필드들을

저장하는 데에 사용한다. 뭔가 더 하드코어^{hard-core}하게 작업할 뿐만 아니라, 더 고-수준 코드로 작업한다. WriteObjectRtti 메서드의 전체 코드는 다음과 같다:

```
procedure TTrivialXmlWriter.WriteObjectRtti(AnObj: TObject);
var
  AContext: TRttiContext;
  AType: TRttiType;
  AField: TRttiField;
begin
  AType := AContext.GetType(AnObj.ClassType);
  for AField in AType.GetFields do
    begin
      if AField.FieldType.IsInstance then
        begin
          WriteStartElement(AField.Name);
          WriteObjectRtti(AField.GetValue(AnObj).AsObject);
          WriteEndElement(True);
        end
      else
        begin
          WriteStartElement(AField.Name);
          WriteString(AField.GetValue(AnObj).ToString);
          WriteEndElement;
        end;
      end;
    end;
end;
```

XML 결과는 이전과 비슷하다. 하지만, 덜 깔끔하다. 필드 이름들이 프로퍼티 이름보다 일반적으로 읽기가 더 어렵기 때문이다:

```
<Company>
  <FName>ACME Inc.</FName>
  <FCountry>Worldwide</FCountry>
  <FID>29470148</FID>
  <FPerson>
    <FName>Wiley</FName>
    <FCountry>Desert</FCountry>
  </FPerson>
</Company>
```

그런데, 큰 차이가 하나 더 있다. 이렇게 하기 위해, 클래스들은 TPersistent 클래스에서 상속받지 않아도 된다. 또한 어떠한 특별한 옵션으로 컴파일할 필요도 없다.

애트리뷰트의 사용해 스트리밍을 사용자 지정하기 Using Attributes to Customize Streaming

XML 태그 이름의 문제와 별개로, 아직 언급하지 않은 또 다른 문제가 있다. XML 태그 이름들, 즉 컴파일 된 실제 심볼들,을 사용하는 것은 그다지 좋은 생각이 아니다. 또한, 이 코드에서는 XML-기반 스트리밍에서 몇몇 프로퍼티나 필드를 제외할 방법이 없다.

참고 오브젝트 파스칼에서 프로퍼티 스트리밍은 지시어를 사용해 제어할 수 있다. 그러면 TypeInfo 유닛을 사용해 읽을 수 있기 때문이다. 하지만 이 해법은 여전히 단순함 또는 깔끔함과 거리가 멀다. 비록 DFM 스트리밍 메커니즘에서 이 해법을 효과적으로 사용하고 있지만 말이다.

그 문제들은 애트리뷰트를 사용해 해결할 수 있다. 하지만 이 해결책은 단점도 있다. 우리의 클래스 선언 안에서 그것들을 상당히 많이 사용해야 한다. 내가 그다지 좋아하지 않는 코딩 스타일이다. 이 코드의 새 버전에서는, 애트리뷰트 생성자를 하나 만들고 파라미터를 받아서 선택할 수 있도록 했다:

```
type
  XmlAttribute = class(TCustomAttribute)
private
  FTag: string;
public
  constructor Create(StrTag: string = '');
  property TagName: string read FTag;
end;
```

아래 애트리뷰트-기반 스트리밍 코드는 확장된 RTTI에 기반하는 이전 버전의 변종이다. 차이점은 하나다. 이제 프로그램은 CheckXmlAttr이라는 헬퍼 함수를 호출한다. 그래서 그 필드가 애트리뷰트를 가지고 있는지 확인한다. 또한 태그 이름을 장식하는 옵션도 있다:

```
procedure TTrivialXmlWriter.WriteObjectAttrib(AnObj: TObject);
var
  AContext: TRttiContext;
  AType: TRttiType;
  AField: TRttiField;
  StrTagName: string;
begin
  AType := AContext.GetType(AnObj.ClassType);
  for AField in AType.GetFields do
    begin
      if CheckXmlAttr(AField, StrTagName) then
        begin
          if AField.FieldType.IsInstance then
            begin
              WriteStartElement(StrTagName);
              WriteObjectAttrib(AField.GetValue(AnObj).AsObject);
              WriteEndElement(True);
            end
          else
            begin
              WriteStartElement(StrTagName);
              WriteString(AField.GetValue(AnObj).ToString);
              WriteEndElement;
            end;
          end;
        end;
      end;
    end;
  end;
```

가장 쓸모 있는 코드는 CheckXmlAttr 헬퍼 함수 안에 들어 있다:

```
function CheckXmlAttr(AField: TRttiField;
  var StrTag: string): Boolean;
var
  Attrib: TCustomAttribute;
begin
  Result := False;
```



```

for Attrib in AField.GetAttributes do
  if Attrib is XmlAttribute then
    begin
      StrTag := XmlAttribute(Attrib).TagName;
      if StrTag = '' then // 기본값
        StrTag := AField.Name;
        Exit(True);
      end;
    end;
end;

```

XML 애트리뷰트가 없는 필드는 무시된다. 또한 XML 출력에 사용되는 태그를 사용자가 지정할 수 있다. 시연을 위해, 이 프로그램에는 아래 클래스들이 있다 (이번에는 코드에 나열되어 있는 게시됨 [published](#) 프로퍼티들을 생략했다. 쓸모가 없기 때문이다):

```

type
  TAttrPerson = class
    private
      [xml( 'Name' )]
      FName: string;
      [xml]
      FCountry: string;
      ...

  TAttrCompany = class
    private
      [xml( 'CompanyName' )]
      FName: string;
      [xml( 'Country' )]
      FCountry: string;
      FID: string; // 생략될 것임
      [xml( 'TheBoss' )]
      FPerson: TAttrPerson;
      ...

```

위 선언은 아래와 같은 XML로 출력된다 (자세히 보자. ID 태그가 생략되었다. 그리고, FCountry 필드의 (보기 좋지 않던) 기본 이름이 다르게 표시되었다):

```

<Company>
  <CompanyName>ACME Inc.</CompanyName>
  <Country>Worldwide</Country>
  <TheBoss>
    <Name>Wiley</Name>
    <FCountry>Desert</FCountry>
  </TheBoss>
</Company>

```

바로 앞의 버전과 차이점이 있다. 우리는 이제 매우 유연하게 다룰 수 있게 되었다. 무슨 필드를 포함할지 그리고 XML 안에서 어떤 이름으로 표현할지를 결정할 수 있다. 앞 버전에서는 불가능했던 것들이다.

이 예제는 그저 뼈대 수준의 구현에 불과하다. 하지만, 여러분에게 이 마지막 버전이 만들어지는 과정을 보여주는 기회가 되었다고 생각한다. 고전 RTTI에서 시작하여 한 단계씩 진행했으므로 이 다양한 기법들 사이의 차이를 여러분이 느낄 수 있었을 것이다.

명심해야 할 중요한 것이 있다. 애트리뷰트를 사용하는 것이 항상 좋은 해답이라고 받아들이지는 말자! 오히려, 명확히 이해하자. RTTI와 애트리뷰트는 런타임의 알려지지 않은 오브젝트의 구조를 살펴봐야 하는 상황에서 엄청난 힘과 유연성을 더해 준다.

다른 RTTI 기반 라이브러리들 Other RTTI-Based Libraries

이 장을 마무리하면서, 집고 넘어가야 할 한 가지 사실이 있다. 몇몇 라이브러리들이 확장된 RTTI를 활용하기 시작했다. 제품의 일부에서도 또한 써드-파티에서도 그렇다. 한 가지 예는 비주얼 라이브 바인딩 [Visual Live Bindings](#)에 있는 바인딩 표현식 메커니즘이다. 여러분은 바인딩 표현을 생성할 수 있고, 거기에 표현식 [expression](#)(이어붙이기, 추가하기 등의 연산이 달려있는 텍스트로 된 문자열)을 대입할 수 있고, 그 표현식이 외부 오브젝트와 그 필드들을 가리키도록 할 수 있다.

이 주제를 깊게 다루고 싶지는 않다. 이것은 특정한 라이브러리에 대한 사용법일 뿐 언어나 핵심 시스템의 일부가 아니기 때문이다. 하지만, 짧게 나열하겠다. 이 정도로도 여러분이 아이디어를 얻을 수 있을 것이라고 생각하기 때문이다:

```
var
  BindExpr: TBindingExpression;
  Pers: TPerson;
begin
  Pers := TPerson.Create;
  try
    Pers.Name := 'John';
    Pers.City := 'San Francisco';

    BindExpr := TBindingExpressionDefault.Create;
    try
      BindExpr.Source := 'person.name + " lives in " + person.city';
      BindExpr.Compile([
        TBindingAssociation.Create(Pers, 'person')]);
      Show(BindExpr.Evaluate.GetValue.ToString);
    finally
      BindExpr.Free;
    end;
  finally
    Pers.Free;
  end;
end;
```

위 방식에서 주목할 점이 있다. 그 장점은 여러분이 런타임에 표현식을 바꿀 수 있다는 사실에서 나온다는 점이다 (비록 위 코드 조각에서는 상수 문자열이지만 말이다). 이 표현식은 입력 상자를 통해서 받을 수도 있다. 또는 여러 가지 표현식들 중에서 동적으로 선택할 수도 있다. 이것은 먼저 `TBindingExpression` 오브젝트에 대입된다. 그리고 나서 분석되고, 런타임에 `Compile` 호출과 함께 *컴파일* 된다 (심볼릭 형태로 변환다, 어셈블리 [assembly](#) 코드가 아니다). 그러면, 그 후 실행될 때, RTTI를 사용하여 `TPerson` 오브젝트에 접근할 수 있게 된다.

단점이 있다. 이 방식으로 표현식을 평가하면 상당히 느리다. 오브젝트 파스칼 소스 코드 안에 컴파일 된 것을 실행하는 것에 비교해서 말이다. 즉, 여러분은 성능 저하와 유연성 향상 사이에서 균형을 찾아야 한다. 예를 들어, 비주얼 라이브 바인딩 모델은 이 원리를 기반으로 구축되었다. 그 결과 매우 멋지고 쉬운 개발자 경험을 제공한다.

17: TObject와 System 유닛

어떤 오브젝트 파스칼 애플리케이션이든, 그 핵심은 클래스 계층구조 [hierarchy](#)다. 시스템의 모든 클래스는 궁극적으로 TObject 클래스의 하위 클래스 [subclass](#)이며, 모든 계층 구조는 하나의 뿌리 [root](#)를 가진다. 따라서 여러분은 TObject 데이터 타입을 사용하여 시스템의 어떤 클래스 타입이든 대신할 수 있다.

TObject 클래스는 RTL의 핵심 유닛 즉 System이라는 유닛 안에 정의되어 있다. 이 유닛은 컴파일할 때마다 항상 자동으로 포함된다. 이처럼 중요한 역할을 하기 때문이다. 여기서 System 유닛 안에 있는 클래스들과 유닛에 있는 함수들을 모두 다루지는 않는다. 하지만, 깊이 살펴볼 가치가 있는 것들이 몇 가지 있다. TObject는 그 중에서도 가장 중요한 클래스다.

참고 TObject 같은 핵심 시스템 클래스가 언어의 일부인지 아니면 런타임 라이브러리(RTL)의 일부 인지를 논쟁하려면 길다. 이 점은 System 유닛의 다른 기능들 역시 마찬가지다. System 유닛은 다른 모든 유닛이 컴파일 될 때 항상 자동으로 포함되는 이유는 그 정도로 중요하기 때문이다. (사실 System 유닛은 `uses 절` [clause](#)에 수작업으로 추가하는 것은 허용되지 않는다.) 어쨌든 이런 논쟁은 쓸데없으므로 다음 기회로 미루겠다.

TObject 클래스 [The TObject Class](#)

방금 말했듯이, TObject 클래스는 매우 특별하다. 다른 모든 클래스들이 결국 TObject로부터 상속을 받기 때문이다. 사실, 여러분이 새 클래스를 선언할 때, 기반 클래스를 명시하지 않으면 그 클래스는 자동으로 TObject를 상속한다. 프로그래밍 언어 관점에서 이런 식의 상황은 *단일 루트 클래스 계층 구조* [singly-rooted class hierarchy](#)라고 한다. 이 특징은 오브젝트 파스칼이 C#, 자바, 그리고 다른 몇몇 현대 프로그래밍 언어들과 공유하는

특징이다. 언급할 만한 예외는 C++다. C++는 단일 기반 클래스 개념이 없다. 따라서 개발자는 완전히 별개인 클래스 계층들 여러 개를 정의할 수 있다.

TObject 기반 클래스는 그 자체로 인스턴스를 생성하여 직접 사용하는 클래스가 아니다. 그러나, 모든 종류의 오브젝트를 담는 클래스로 선언할 변수가 필요한 모든 경우에 여러분이 사용하는 클래스다. 좋은 사용법 예제는 컴포넌트 라이브러리 안에 있다. 그 안에 있는 이벤트 핸들러들은 TObject를 첫 파라미터로 사용한다. 그 이름은 일반적으로 Sender라고 불린다. 즉, 즉 어느 클래스의 어느 오브젝트도 Sender가 될 수 있다. 많은 제네릭 컬렉션들 역시 오브젝트들의 컬렉션이다. 그래서, TObject 타입이 직접 사용되는 경우들이 꽤 있다.

다음 소단원들에서는 TObject 클래스의 몇 가지 특징들을 본다. 이 특징들은 델파이로 작성된 모든 클래스들 그리고 그 클래스를 사용하는 여러분의 모든 코드에 적용된다.

생성과 소멸 Construction and Destruction

TObject를 직접 생성하는 것은 타당하지 않다. 하지만, 이 클래스의 생성자와 소멸자는 중요하다. 그것들은 다른 모든 클래스들에게 자동으로 상속되기 때문이다. 여러분이 클래스를 정의하면서 생성자를 만들어 넣지 않아도, 여러분은 여전히 그 클래스에서 Create를 호출할 수 있다. 그러면, 여러분은 TObject의 생성자를 불러 내는 것이다. 그 생성자는 그저 텅 빈 메서드다 (그 최상위 기반 클래스 안에서는 초기화할 것이 없기 때문이다). 이 Create 생성자는 가상이 아니다. 그리고 여러분은 여러분이 만든 클래스 안에서 이것을 완전히 교체할 수 있다. 이 아무것도 하지 않는 생성자가 맞았지 않다면 말이다. 모든 하위 클래스는 그 기반 클래스의 생성자를 호출하는 것이 좋은 습관이다. TObject.Create를 직접 호출하는 것이 특별히 쓸모가 있지 않아도 그렇다.

참고 이 생성자가 가상이 아님을 강조하는 이유가 있다. 이와 달리, 또다른 핵심 라이브러리 클래스인 TComponent는 가상 생성자를 정의하기 때문이다. TComponent 클래스의 가상 생성자는 스트리밍 시스템에서 핵심 역할을 한다. 다음 장에서 다룰 것이다.

오브젝트를 소멸하기 위해, TObject 클래스에는 Free 메서드가 있다. 이것은 결국 Destroy 소멸자를 호출한다. 자세한 내용은 13장에서 정확한 메모리 관리에 대한 많은 제안들과 함께 다뤘다. 그래서 여기에서는 다시 반복하지 않는다.

오브젝트에 대해 알아보기 Knowing About an Object

TObject 클래스에는 흥미로운 메서드들의 그룹이 있다. 바로 그 타입에 대한 정보를 반환하는 것들이다. 가장 흔하게 쓰이는 것으로는 ClassType과 ClassName 메서드다. ClassName 메서드는 클래스의 이름을 담은 문자열을 반환한다. 이것은 클래스 메서드다. 따라서, (TObject의 수많은 클래스 메서드들이 그렇듯이) 여러분은 이것을 오브젝트와 클래스에 모두 적용할 수 있다. 만약 여러분이 TButton 클래스를 정의했고 그 클래스의

오브젝트로 `Button1`을 정의했다면, 다음 문장은 둘 다 효과가 똑같다:

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

물론, 여러분은 이 메서드를 일반적인 `TObject`에도 적용할 수 있다. 하지만, 여러분은 `TObject`의 정보를 얻지 못할 것이다. 그 대신 그 변수에 대입된 현재의 오브젝트의 해당 클래스 정보를 얻게 될 것이다. 예를 들어, 버튼의 `OnClick` 이벤트 핸들러에서:

```
Text := Sender.ClassName;
```

라고 호출하면, 앞에서 본 호출들과 똑같이 문자열 `'TButton'`을 반환한다. 그 이유는 클래스 이름이 런타임에 (특정 오브젝트 자체에 의해) 결정되기 때문이다. 컴파일러에 의해 결정되는 것이 아니다(컴파일러는 이것이 `TObject` 오브젝트라는 생각만 한다). 물론, 그 참조 [reference](#)가 대입되어 있지 않으면, 즉 그 변수가 `nil`이면, 클래스 메서드를 호출하려는 모든 시도는 예외를 일으킨다.

클래스 이름을 얻는 것은 디버깅 [debugging](#), 로깅 [logging](#), 그리고 클래스 정보를 일반적으로 보여주기 등에서 편리하다. 그런데, 종종 더 중요한 것은 그 클래스의 클래스 참조에 접근하는 것이다. 예를 들어, 클래스 참조 두개를 비교하는 것이 클래스 이름을 담은 문자열 두 개를 비교하는 것보다 더 좋다. 클래스 참조는 단순한 숫자 주소다. 하지만, 문자열은 데이터 구조라서 평가하는데 더 많은 CPU 사이클이 사용된다. 클래스 참조를 얻으려면 `ClassType` 메서드를 사용한다. 한편, `ClassParent` 메서드는 현재 클래스의 기반 클래스에 대한 클래스 참조를 반환한다. 그래서 상위 클래스들, 즉 기반 클래스들의 목록을 탐색할 수 있다. 이 메서드는 `TObject`에 대해서 `nil`을 반환한다. 최상위 클래스이므로 부모 클래스가 없기 때문이다. 여러분이 클래스 참조를 가지고 있다면, 여러분은 그것을 사용해서 어떤 클래스 메서드라도 호출할 수 있다. `ClassName` 메서드도 그 중 하나다.

클래스에 대한 정보를 반환하는 매우 흥미로운 메서드로 `InstanceSize`가 있다. 이것은 오브젝트의 런타임 크기 [runtime size](#)을 반환한다. 즉 그 오브젝트의 필드들을 위해 필요한 메모리의 양을 알려준다. 이 기능은 시스템이 클래스의 새 인스턴스를 할당해야 할 때 내부적으로 사용한다.

참고 여러분은 `SizeOf` 전역 함수가 그런 정보를 제공하지 않는다고 생각할 수 있겠지만, 실제로 그 함수는 오브젝트 참조의 크기를 반환한다. 오브젝트 자체의 크기가 아니다. 그리고, 오브젝트 참조는 포인터이므로 크기가 정해져 있다. 타겟 플랫폼에 따라 4 또는 8바이트다. 한편, `InstanceSize`는 필드들의 크기를 반환한다. 하지만, 오브젝트가 사용하는 실제 메모리 양은 아니다. 그 필드들 중에는 문자열 그리고 기타 메모리에 있는 오브젝트를 참조할 수도 있는데, 그것들은 추가 공간이 필요하기 때문이다.

TObject 클래스의 더 많은 메서드들 More Methods of the TObject Class

TObject 클래스에는 다른 메서드들도 있다. 이것들 역시 여러분은 모든 오브젝트에서 적용할 수 있다. 물론 모든 클래스 또는 클래스 참조에서도 적용할 수 있다. 그것들이 클래스 메서드이기 때문이다. 아래에는 그 중 일부와 해당 짧은 설명을 나열했다:

- **ClassName:** 클래스 이름을 담은 문자열을 반환한다. 주로 화면 표시용으로 사용된다.
- **ClassNameIs:** 클래스 이름이 주어진 값과 일치하는지 확인한다.
- **ClassParent:** 현재 클래스 또는 오브젝트의 클래스에 대한 부모 클래스를 가리키는 클래스 참조를 반환한다. 여러분은 ClassParent의 ClassParent로 계속 탐색해 갈 수 있다. 그렇게 TObject 클래스까지 도달할 수 있다 (그러면 이 메서드는 nil을 반환한다).
- **ClassInfo:** 클래스 내부의, 저수준 런타임 타입 정보(RTTI)의 포인터를 반환한다. TypInfo 유닛의 초창기에 사용되었지만, 지금은 RTTI 유닛에 있는 기능들로 대체되었다. 16장에서 다뤘다. 시스템 내부에서는 여전히 이것을 통해 클래스에 대한 RTTI를 받아낸다.
- **ClassType:** 오브젝트의 클래스를 가리키는 참조를 반환한다(이것은 클래스 메서드가 아니다. 따라서 클래스에 직접 적용하지 못한다). 이것이 오브젝트의 클래스에 대한 참조다. 따라서 이름이 동일하지만 다른 유닛 안에 선언된 두 클래스는 서로 일치하지 않는다. 델파이는 타입에 엄격한 *strongly typed* 언어이기 때문에 이 결과는 옳다.
- **InheritsFrom:** 그 클래스가 주어진 기반 클래스로부터 (직접적 혹은 간접적으로) 상속을 받는지 테스트한다 (이것은 is 연산자와 매우 비슷하며, 실제로 is 연산자 구현에서는 결국 이것을 사용한다).
- **InstanceSize:** 오브젝트 데이터의 크기를 바이트 단위로 반환한다. 이것은 필드들의 크기를 모두 합친 값과, 거기에 더해 특별히 예약된 약간의 추가 바이트가 합쳐진 값이다(추가 바이트에는 예를 들어, 해당 클래스 참조가 포함된다. 하지만, 문자열들과 내부 오브젝트들이 사용하는 실제 메모리는 포함되지 않는다). 하나 더 알아둘 것이 있다. 이것은 인스턴스의 크기다. 한편, 인스턴스 참조의 크기는 포인터의 크기와 같다(4 혹은 8바이트이며, 플랫폼에 따라 다르다).
- **UnitName:** 그 클래스가 정의되어 있는 유닛의 이름을 반환한다. 클래스를 설명하는데 유용할 수 있다. 사실 시스템 안에서 클래스 이름은 유일하지 않다. 이전 장에서 봤듯이, 애플리케이션 안에서, (유닛 이름과 클래스 이름이 점으로 구분된) 이름 전체가 적힌 *qualified* 클래스 이름만이 유일하다.
- **QualifiedClassName:** 유닛 이름과 클래스의 이름 조합을 반환한다. 작동하고 있는 시스템 안에서 그 값이야 말로 진정으로 유일하다.

이 TObject 메서드들은 (그 클래스가 무엇이든) 모든 오브젝트들에서 사용할 수 있다. TObject 가 모든 델파이 클래스들의 공통 조상이기 때문이다.

우리가 어떻게 이들 메서드로 클래스 정보에 접근하는지 그 방법은 아래와 같다:


```

procedure TSenderForm.ShowSender(Sender: TObject);
begin
    Memo1.Lines.Add( 'Class Name: ' + Sender.ClassName);

    if Sender.ClassParent <> nil then
        Memo1.Lines.Add( 'Parent Class: ' + Sender.ClassParent.ClassName);

    Memo1.Lines.Add( 'Instance Size: ' + IntToStr(Sender.InstanceSize));

```

이 코드는 ClassParent가 nil인지 확인한다. 여러분이 실제로 TObject의 인스턴스를 사용하고 있는 상황을 대비하기 위해서다(TObject는 기반 타입이 없다). 여러분은 이 테스트 부분에서 다른 메서드들을 사용해 볼 수도 있다. 예를 들어 Sender 오브젝트가 특정한 타입인지를 확인할 수 있다. 그 코드는 아래와 같다:

```

if Sender.ClassType = TButton then ...

```

또한, Sender 파라미터가 주어진 오브젝트에 대응하는지 확인할 수 있다. 아래와 같다:

```

if Sender = Button1 then ...

```

특정 클래스나 오브젝트를 테스트하는 대신, 일반적으로 여러분은, 주어진 오브젝트에 대해 오브젝트의 타입 호환성을 테스트해야 하는 경우가 많을 것이다; 그 오브젝트의 클래스가 특정 클래스나 그 하위 클래스 중 하나에 해당되는지 확인하면, 해당 클래스의 메서드를 그 오브젝트에서 사용할 수 있는지를 알 수 있기 때문이다. 그 테스트는 InheritsFrom 메서드를 사용해도 되고, is 연산자를 사용해도 된다. 그 둘 사이에 차이점은 하나다. is 연산자는 nil을 다룰 수 있다. 아래 두 코드는 동일하다:

```

if Sender.InheritsFrom(TButton) then ...
if Sender is TButton then ...

```

클래스 정보 보여주기 [Showing Class Information](#)

여러분이 클래스 참조를 가지고 있으면, 여러분은 그 클래스에 대한 설명을 서술할 때 그것의 모든 기반 클래스들을 나열할 수 있다. 다음 코드 조각을 보자. MyClass의 기반 클래스들이 ListBox 컨트롤에 추가된다:

```

ListParent.Items.Clear;
while MyClass.ClassParent <> nil do
begin
    MyClass := MyClass.ClassParent;
    ListParent.Items.Add(MyClass.ClassName);
end;

```

우리가 클래스 참조를 while 루프의 중심으로 사용한다는 점을 여러분은 알 수 있을 것이다. 부모 클래스가 존재하지 않는지(즉, 현재 클래스가 TObject인지)를 테스트한다. 그 대안으로, while 문을 다음 두 가지 방법으로 작성할 수 있다:

```

while not MyClass.ClassNameIs( 'TObject') do... // 느리고, 오류가 나기 쉬움
while MyClass <> TObject do... // 빠르고, 읽기 쉬움

```


TObject의 가상 메서드들 TObject's Virtual Methods

TObject 클래스의 구조는 꽤 안정적이다. 오브젝트 파스칼 언어의 초창기부터 그랬다. 그리고 어느 시점부터, 매우 유용한 가상 메서드 세 개가 추가되었다. 이 메서드들은 모든 오브젝트에서 호출할 수 있다. TObject의 다른 모든 메서드들일 그렇듯이 말이다. 하지만, 이 세 메서드들을 쓰려면 여러분이 여러분의 클래스에 안에서 재정의해야 한다.

참고 만약 여러분이 .NET 프레임워크를 써본 적이 있다면 C#의 기반 클래스 라이브러리의 System.Object 클래스 안에 이 메서드들이 들어 있다는 것을 바로 알아챌 것이다. 이와 비슷한 메서드들은 자바에 있는 기반 클래스들에도 사용된다. 또한 자바스크립트 등 다른 언어들에서도 흔하게 사용된다. 이 메서드들의 근원은, 예를 들어 toString 메서드 같은 것은, Smalltalk로 거슬러 올라간다. 이 언어는 최초의 OOP 언어로 여겨진다.

ToString 메서드

ToString 가상 함수는 주어진 오브젝트의 텍스트 표현(설명 description 또는 심지어 직렬화 serialization)을 반환하는 자리 표시자 placeholder다. TObject 클래스에 있는 이 메서드의 기본 구현은 클래스 이름을 반환한다:

```
function TObject.ToString: string;
begin
    Result := ClassName;
end;
```

물론, 이것은 쓸모와는 거리가 멀다. 이론적으로, 각 클래스는 자기 자신을 설명하는 방법을 사용자에게 제공해야 한다. 예를 들어 오브젝트가 시각적 리스트에 추가될 때 등이다. 런타임 라이브러리 안에 있는 몇몇 클래스들은 ToString 가상 함수를 오버라이드 한다. 예를 들어, TStringBuilder, TStringWriter, Exception 클래스 등이 그렇게 한다. 그래서 예외 목록 안에 해당 메시지를 반환한다(9장의 "내부 예외 메커니즘 inner Exception Mechanism" 소단원에서 설명했다).

표준 방법을 통해 무슨 오브젝트든 해당 문자열 표현을 반환한다는 것은 매우 흥미로운 아이디어다. 여러분도 TObject 클래스의 이 핵심 기능을 언어 기능처럼 취급해 활용하기를 권장한다.

참고 ToString 메서드가 “구문 분석 토큰 parse token 문자열”이나 Classes 유닛에 정의된 toString 심볼을 “의미에서 오버로드” semantically overload함에 주목하라. 그런 이유로 그 심볼은 Classes.toString을 가리킨다.

Equals 메서드

Equals 가상 함수는 두 오브젝트가 동일한 논리 값을 가지는지 확인하는 자리 표시자 placeholder다. 이는 두 변수가 메모리 안의 동일한 오브젝트를 참조하고 있는지를 확인하는 동작(즉, = 연산자를 사용할 때 수행되는 동작)과는 다르다. 그런데, 정말 헛갈린다.

TObject의 기본 구현에서는 실제로 이 두 동작이 똑같기 때문이다. TObject에서는 그 이상의 비교 방법이 없기도 하다:

```
function TObject.Equals(Obj: TObject): Boolean;
begin
    Result := Obj = Self;
end;
```

이 메서드를 사용하는 (그리고 적절하게 오버라이드 하는) 예제로 TStrings 클래스를 보자. Equals 메서드는 TStrings 리스트 안에 있는 문자열들의 개수를 비교한다. 만일 같을 경우, 실제 문자열의 내용을 하나씩 비교한다. 이 동작은 짝이 맞지 않거나 또는 리스트의 끝에 도달할 때까지 진행된다 - 끝까지 도달한다면 두 리스트는 같다.

라이브러리의 부분 중에, 이 기법이 중요하게 사용되는 곳은 제네릭 지원 부분이다. 특히 Generics.Default와 Generics.Collections 유닛이다. 라이브러리나 프레임워크는 오브젝트의 “값 일치” 개념을 정의하는 것이 중요하다. 그 오브젝트 자체를 비교하는 것과는 별도로 필요하다. 표준 메커니즘을 통해 오브젝트를 “값으로” 비교하는 것은 매우 큰 이점이다.

GetHashCode 메서드

GetHashCode 가상 함수는 .NET 프레임워크에서 가져온 또다른 자리 표시자다. 이것은 각 클래스가 자신의 오브젝트들의 해시 코드를 계산할 수 있도록 한다. 기본 코드는 아래와 같다. 이 함수가 반환하는 것은 언뜻 보기에 무작위 값인데, 그 오브젝트 자체의 주소다:

```
function TObject.GetHashCode: Integer;
begin
    Result := Integer(Self);
end;
```

참고 생성되는 오브젝트들의 주소는 일반적으로 제한된 힙 영역들의 집합에서 가져온다. 그 숫자의 분포는 고르지 않다. 따라서 해싱 알고리즘에 오히려 부정적인 영향을 미칠 수 있다. 이 메서드가 좋은 해시 분포를 가진 논리 값들에 기반하여 해시를 생성하도록 여러분이 커스터마이징 할 것을 강력히 권한다. 그러면 디셔너리를 그리고 해시 값을 사용하는 데이터 구조의 성능이 더 좋아질 것이다.

GetHashCode 가상 함수는 해시 테이블을 지원하는 몇몇 컬렉션 클래스들이 사용한다. 또한 TDictionary<T> 같은 클래스들에서 일부 코드를 최적화하는 방법이기도 하다.

TObject 가상 메서드 사용하기 [Using TObject Virtual Methods](#)

여기 TObject의 가상 메서드 몇 가지에 기반한 예제가 있다. 이 예제에 있는 클래스는 앞에서 본 메서드들 중 두 개를 오버라이드 한다:

```
type
    TAnyObject = class
```



```

private
  FValue: Integer;
  FName: string;
public
  constructor Create(AName: string; AValue: Integer);
  function Equals(Obj: TObject): Boolean; override;
  function ToString: string; override;
end;

```

이 메서드 세 개의 구현에서는 그저 GetType에 대한 호출을 ClassType 호출로 바꿨다:

```

constructor TAnyObject.Create(AName: string; AValue: Integer);
begin
  inherited Create;
  FName := AName;
  FValue := AValue;
end;

function TAnyObject.Equals(Obj: TObject): Boolean;
begin
  Result := (Obj.ClassType = Self.ClassType) and
    ((Obj as TAnyObject).Value = Self.Value);
end;

function TAnyObject.ToString: string;
begin
  Result := Name;
end;

```

위 코드를 잘 보자. 오브젝트가 동일한 클래스 타입이고, 그 값이 일치하면, 동일한 것으로 간주된다. 한편, 그 오브젝트의 문자열 표현에는 FName 필드만 들어간다.

이 프로그램은 시작할 때 위 클래스의 오브젝트를 몇 개 만든다:

```

procedure TFormSystemObject.FormCreate(Sender: TObject);
begin
  Ao1 := TAnyObject.Create('Ao1', 10);
  Ao2 := TAnyObject.Create('Ao2 or Ao3', 20);
  Ao3 := Ao2;
  Ao4 := TAnyObject.Create('Ao4', 20);
  ...

```

위 코드를 잘 보자. 두 참조(Ao2 와 Ao3)는 메모리 안의 같은 오브젝트를 가리킨다. 마지막 오브젝트(Ao4)는 그것들과 똑같은 숫자 값을 가진다.

이 프로그램의 사용자 인터페이스에서 사용자는 아무 항목이든 2 개를 선택해 그 오브젝트들을 비교한다. Equals 사용하기와 직접적인 참조 비교하기를 모두 테스트한다. 몇 가지 결과를 보면 아래와 같다:

```

Comparing Ao1 and Ao4
Equals: False
Reference = False

Comparing Ao2 and Ao3
Equals: True
Reference = True

```



```
Comparing Ao3 and Ao4
Equals: True
Reference = False
```

이 프로그램에는 또다른 버튼이 있는데, 그 버튼 자체를 대상으로 테스트한다:

```
var
  Btn2: TButton;
begin
  Btn2 := BtnTest;
  Log( 'Equals: ' +
    BoolToStr(BtnTest.Equals(Btn2), True));
  Log( 'Reference = ' +
    BoolToStr(BtnTest = Btn2, True));
  Log( 'GetHashCode: ' +
    IntToStr(BtnTest.GetHashCode));
  Log( 'ToString: ' + BtnTest.ToString);
end;
```

출력 결과는 (실행 시 해시 값은 변한다) 다음과 같다:

```
Equals: True
Reference = True
GetHashCode: 28253904
ToString: TButton
```

TObject 클래스 요약 TObject Class Summary

요약해서, TObject 클래스 전체 인터페이스를 최신 버전의 컴파일러에서 다음과 같다 (IFDEF, 저-수준 오버로드, 비공개 `private` 구역, 보호된 `protected` 구역 등은 생략했다):

```
type
  TObject = class
  public
    constructor Create;
    procedure Free;
    procedure DisposeOf;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;

    function ClassType: TClass; inline;
    class function ClassName: string;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer; inline;
    class function InstanceSize: Integer; inline;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: string): Pointer;
    class function MethodName(Address: Pointer): string;
    class function QualifiedClassName: string;
    function FieldAddress(const Name: string): Pointer;
    function GetInterface(const IID: TGUID; out Obj): Boolean;

    class function GetInterfaceEntry(
      const IID: TGUID): PInterfaceEntry;
```



```

class function GetInterfaceTable: PInterfaceTable;
class function UnitName: string;
class function UnitScope: string;

function Equals(Obj: TObject): Boolean; virtual;
function GetHashCode: Integer; virtual;
function ToString: string; virtual;
function SafeCallException(ExceptObj: TObject;
    ExceptAddr: Pointer): HResult; virtual;
procedure AfterConstruction; virtual;
procedure BeforeDestruction; virtual;
procedure Dispatch(var Message); virtual;
procedure DefaultHandler(var Message); virtual;
class function NewInstance: TObject; virtual;
procedure FreeInstance; virtual;
destructor Destroy; virtual;

public
  property Disposed: Boolean read GetDisposed;
end;

```

유니코드와 클래스 이름 Unicode and Class Names

오버로드된 메서드들, 즉 MethodAddress나 FieldAddress 같은 것들은 파라미터로 UnicodeString(주로 UTF-16이다) 또는 (UTF-8 문자열로 다룰 수 있는) ShortString을 받을 수 있다. 사실, 일반 유니코드 문자열을 받는 버전은 UTF8EncodeToShortString 함수를 호출해 변환한다:

```

function TObject.FieldAddress(const Name: string): Pointer;
begin
  Result := FieldAddress(UTF8EncodeToShortString(Name));
end;

```

유니코드 지원이 이 언어에 도입된 후에는, 오브젝트 파스칼의 클래스 이름은 내부적으로 ShortString 표현을 사용한다(1-바이트 문자들의 배열임). 그런데, UTF-8 인코딩을 사용하고 있다. ShortString 타입의 전통적인 ANSI 인코딩 encoding이 아니다. 이는 TObject 수준과 RTTI 수준에서 둘 다에서 그렇게 된다.

예를 들어, ClassName 메서드는 매우 저수준 low-level의 코드로 구현되어 있다 다음과 같다:

```

class function TObject.ClassName: string;
begin
  Result := UTF8ToString(
    PShortString(PPointer(
      Integer(Self) + vmtClassName)^)^);
end;

```

이와 유사하게 TypInfo 유닛 안에서는, 클래스 이름에 접근하는 모든 함수들은 내부에서 사용하고 있는 UTF-8 ShortString 표현을 UnicodeString으로 변환한다. 프로퍼티 이름에서도 비슷한 일이 일어난다.

System 유닛 The System Unit

TObject 클래스는 분명히 언어에서 기반 역할을 한다. 하지만, 이것이 언어의 일부인지 런타임 라이브러리의 일부인지 말하기 매우 어렵다. System 유닛 안에는 컴파일러 지원에 대한 기본적이고 통합된 부분을 구성하는 다른 저수준의 클래스들이 있다.

이 유닛의 대부분의 내용은 저수준 데이터 구조, 단순 레코드 구조, 함수와 프로시저, 몇 개의 클래스로 구성된다.

여기서 대부분 클래스에 초점을 맞추겠지만, System 유닛의 다른 많은 기능들도 언어의 핵심임을 부정할 수 없다. 예를 들어, system 유닛은 이른바 “내장된 *intrinsic*” 함수들을 정의한다. 실제 코드는 없지만 컴파일러에 의해 직접 해소 *resolve*되는 것들이다. 그 예는 sizeof다. 컴파일러는 이 호출을 파라미터로 전달된 데이터 구조의 실제 크기로 직접적으로 교체한다.

System 유닛의 시작 부분에 추가된 설명을 읽으면 여러분은 이 유닛의 특별한 역할에 대한 아이디어를 얻을 수 있다(시스템 심볼 *symbol*를 탐색하면, 여러분이 찾고 있는 그 심볼이 아니라, 왜 이 유닛으로 연결되는지 그 이유를 주로 설명한다. :

```
{ 미리 정의된 상수, 타입, 프로시저, }
{ 그리고 함수들 (예를 들어 True, Integer, 혹은 }
{ WriteLn) 은 실제 선언이 없습니다.}
{ 대신 이들은 컴파일러 안에 탑재되어 있습니다 }
{ 그리고 마치 System 유닛의 맨 앞에 }
{ 선언이 된 것처럼 취급됩니다. }
```

이 유닛의 소스 코드를 읽는 것은 다소 지루할 수 있다. 또한 여기에는 전체 런타임 라이브러리의 저수준 코드 일부가 들어 있기 때문에, 그 내용 중 극히 일부만 설명하겠다.

선택된 시스템 타입들 Selected System Types

위에서 말했던 대로, System 유닛은 핵심 데이터 타입들을 정의한다. 그리고 (여러 가지 숫자 타입, 기타 서수 타입, 문자열들에 대한) 많은 타입 별칭 *alias*들도 정의한다. 또한 다른 핵심 데이터 타입들도 여기에 있다. (열거형, 레코드, 강력한 타입 별칭을 이루는) 시스템의 저-수준에서 사용되는 것들이다. 살펴볼 가치가 있는 것들이다.

- TVisibilityClasses: 열거형이다. RTTI 가시성 설정에 사용한다 (자세한 내용은 16장에 있다)
- TGUID: 레코드다. Windows의 GUID를 표현한다. 다른 모든 플랫폼에서도 지원된다
- TMethod: 핵심 레코드다. 이벤트 핸들러를 위해 사용되는 구조를 표현한다. 포인터 하나는 메서드 주소를 그리고 다른 하나는 현재 오브젝트를 가리킨다 (10장에서 간단히 언급했다)

- TMonitor: 레코드다. 쓰레드 동기화 메커니즘(“모니터”[monitor](#)라고 부른다)을 구현한다. C.A.R Hoare와 Per Brinch Hansen이 발명했다. 위키백과[Wikipedia](#)의 “모니터 (동기화)” 문서에 자세한 내용이 있다. 이것은 언어 자체의 핵심 쓰레딩 [threading](#) 지원 기능이다. TMonitor 정보는 시스템의 모든 오브젝트에 부착되기 때문이다.
- TDateTime: Double 타입에 대한 강한 타입 별칭이다. Double 타입은 날짜 정보(값의 정수 부분)와 시간 정보(소수 부분)을 저장할 때 사용된다. 다른 별칭으로 TDate와 TTime 타입도 있다. 이 타입들은 2장에서 다뤘다.
- THandle: 숫자 타입의 별칭이다. 운영체제 오브젝트에 대한 참조를 표현하는데 사용된다. 운영체제 오브젝트는 흔히 (윈도우 API 식으로 말해) “핸들”[handle](#)이라 부른다.
- TMemoryManagerEx: 레코드다. 핵심 메모리 동작들을 가지고 있다. 시스템 메모리 매니저를 사용자 지정 매니저로 바꿀 때 필요한 동작들이다. (이것은 TMemoryManager의 새 버전이다. TMemoryManager 역시 여전히 사용할 수 있다. 이전 버전과의 호환성을 위해서 남겨두었다).
- THeapStatus는 레코드다. 힙 메모리의 상태에 대한 정보를 가진다. 13장에서 짧게 언급했다.
- TTextLineBreakStyle: 열거형 [enumeration](#)이다. 주어진 운영체제에 있는 텍스트 파일을 위한 줄바꿈 스타일을 알려준다. 이 타입의 DefaultTextLineBreakStyle 전역 변수는 현재 정보를 가지고 있으며, 많은 시스템 라이브러리들이 사용한다. 이와 똑같은 정보들은 sLineBreak 상수(이상하게도 라이브러리 코드에 소문자 표기로 되어 있다)는 한 소문자를 가진)가 문자열 값으로 표현한다.

System 유닛의 인터페이스들 [Interfaces in the System Unit](#)

몇 가지 인터페이스 타입 (그리고 핵심 수준에서 인터페이스들을 구현하는 몇 가지 클래스)들이 System 유닛 안에 있다. 살펴볼 가치가 있는 것들이다. System 유닛에서 가장 쓸모가 많은 인터페이스 관련 타입들을 여기에 나열했다.

IInterface는 모든 다른 인터페이스들이 상속을 받는 기본 인터페이스 타입이다. TObject가 다른 클래스들을 위해 하는 것과 똑같은 필수적인 역할을 한다.

- IInvokable와 IDispatch: 동적 호출 [dynamic invocation](#) 형태를 지원하는 인터페이스다 (부분적으로 윈도우 COM 구현에 묶여 있다)
- 열거자 [enumerator](#) 지원과 비교 [comparison](#) 연산들은 다음 인터페이스들에 정의되어 있다: IEnumerator, IEnumerable, IEnumerator<T>, IEnumerable<T>, IComparable, IComparable<T>, IEquatable<T>.

몇 개의 핵심 클래스들도 있다. 이것들은 인터페이스의 기본 구현을 제공한다. 여러분은 아래의 기반 클래스들을 상속받아서 인터페이스를 구현할 수 있다. 11장에서 다뤘다.

- TInterfacedObject: 참조 카운팅과 인터페이스 ID 확인에 대한 기본 구현이 되어 있는 클래스다.

- TAggregatedObject와 TContainedObject: 이 두 클래스에는 결집된 `aggregated` 오브젝트 그리고 `implements` 구문 `syntax`을 위한 특별한 구현을 되어 있다.

선택된 시스템 루틴들 Selected System Routines

System 유닛 안에는 들어 있는 내장된 `intrinsic` 그리고 표준화된 프로시저와 함수는 상당히 많다. 하지만, 흔하게 사용되는 것들은 많지 않다. 아래에는 핵심 함수와 프로시저를 선별하여 놓았다. 모든 오브젝트 파스칼 개발자들이 알아야 하는 것들이다:

- Move: 시스템에 있는 핵심 메모리 복사 동작이다. 주어진 개수의 바이트들을 메모리의 한 위치에서 다른 위치로 그대로 복사한다(매우 강력하고 빠르다. 하지만 약간 위험하다).
- ParamCount 함수와 ParamStr 함수: 애플리케이션의 커맨드-라인 `command-line` 파라미터를 처리하는데 사용될 수 있다 (실제로 Windows와 Mac과 같은 GUI 시스템에서도 작동한다)
- Random과 Randomize: 두 개의 클래식한 함수다. (BASIC의 스템밍 `stemming`처럼) 무작위 값을 제공한다. (유사-무작위 `pseudo-random`다. 여러분이 Randomize를 사용해야 한다는 것을 잊지 않아야만 한다. 그렇지 않으면, 여러분이 프로그램을 실행할 때 마다 항상 같은 수열을 얻게 될 것이다).
- 핵심적인 수학 함수들: 상당히 많다. 여기서는 모두 생략했다.
- 문자열 처리 및 문자열 변환 함수들: 많이 있다. UTF-16 유니코드, UTF-8, ANSI 및 기타 문자열 형식들을 다룰 수 있다. 몇 가지는 특정 플랫폼용이다.

참고 이 함수들 중 몇몇은 간접 정의를 가진다. 다른 말로, 사실 이 함수는 실제 함수에 대한 포인터다. 따라서 그 본연의 시스템 동작이 런타임에 코드에서 동적으로 교체될 수 있다. (물론 자신이 무엇을 하는지 안다면, 여러분의 애플리케이션의 메모리를 폐기하는 좋은 방법이 될 수 있다).

미리 정의된 RTTI 애트리뷰트들 Predefined RTTI Attributes

이 장에서 볼 마지막 데이터 타입들이다. 애트리뷰트 `attribute`들 즉 여러분이 이 언어의 어떤 심볼에든 붙일 수 있는 추가 RTTI 정보들이다. 이 주제는 이미 16장에서 다뤘다. 하지만, 시스템에 미리 정의되어 있는 `predefined` 애트리뷰트들을 언급하지는 않았다.

System 유닛 안에 정의되어 있는 애트리뷰트 클래스들 중 몇 가지다:

- TCustomAttribute: 모든 사용자 지정 애트리뷰트들을 위한 기반 클래스다. 여러분은 여기에서 애트리뷰트를 상속받아야 한다 (컴파일러가 클래스를 애트리뷰트로 식별하도록 하는 유일한 방법이다. 이와 관련된 선언 구문이 따로 없기 때문이다).
- WeakAttribute: 인터페이스 참조가 약한 `weak` 참조임을 표시할 때 사용한다 (13장을 보라)
- UnsafeAttribute: 인터페이스 참조의 참조 카운팅을 비활성화할 때 사용한다 (역시 13장에서 다뤘다)

- RefAttribute: 상수 `const` 함수 파라미터가 되도록 할 때 사용한다
- VolatileAttribute: 휘발성 `volatile` 변수임을 나타낸다. 즉 외부에서 수정할 수 있으며 컴파일러에 의해 최적화되면 안 되는 변수를 나타낸다.
- StoredAttribute: 프로퍼티의 `stored` 플래그를 표현하는 대안이다
- HPPGENAttribute: C++ 인터페이스 파일(HPP 파일) 생성을 제어한다
- HFAAttribute: ARM 64-bit CPU의 파라미터 전달 메커니즘을 세밀히 조정할 때 사용된다. 동종 부동 소수점 집계(Homogeneous Floating-point Aggregates, HFA)를 제어한다.

System 유닛 안에는 더 많은 것들이 있다. 하지만, 전문가들을 위한 내용이므로 생략한다. 이제 이 책의 마지막 장으로 넘어가서, Classes 유닛과 런타임 라이브러리 능력의 일부를 보자.

18: 다른 핵심core RTL 클래스들

TObject 클래스와 System 유닛이 이 언어의 구조적 일부로써 효력을 가진다고 본다면, 즉 애플리케이션을 구축하기 위해 컴파일러 자체에 반드시 필요한 것이라고 본다면, 런타임 라이브러리(RTL)의 다른 모든 항목들은 그 핵심 시스템에 대한 선택적^{optional} 확장들이라고 볼 수 있다.

RTL은 시스템 기능들을 담은 방대한 모음이다. 가장 흔한 표준 작업들을 담고 있다. 그 일부는 터보 파스칼 시대까지 거슬러 올라간다. 이는 오브젝트 파스칼 언어보다도 앞선다. RTL의 많은 유닛들은 함수와 루틴들의 모음이다. 핵심 유틸리티(SysUtils), 수학 함수(Math), 문자열 연산(StringUtils), 날짜와 시간 처리(DateUtils) 등이 있다.

이 책에서는 RTL의 보다 전통적인 부분까지 깊이 파고들고 싶지 않다. 오히려 핵심 클래스들에 집중한다. 즉 오브젝트 파스칼(VCL과 파이어몽키) 및 기타 하위 시스템들 안에서 사용되는 시각적 컴포넌트^{visual component} 라이브러리들의 기반들을 주로 살펴본다. 예를 들어, TComponent 클래스는 “컴포넌트-기반”^{component-based} 아키텍처 개념을 정의하고 있다. 또한 이것은 메모리 관리 및 다른 기반 기능들의 바탕이기도 하다. TPersistent 클래스는 컴포넌트 표현을 스트리밍 하기 위한 핵심이다.

우리가 보면 좋은 다른 클래스들도 많이 있다. RTL은 엄청나게 방대하기 때문이다. 파일 시스템, 핵심 쓰레드 처리^{threading} 지원, 병렬 프로그래밍 라이브러리, 문자열 구축, 유형이 서로 다른 많은 컬렉션들, 컨테이너 클래스들, 핵심 기하학적^{geometrical} 구조들 (예: 점^{point}, 사각형^{rectangle}), 수학적 구조들(벡터^{vector}, 행렬^{matrices}), 기타 등등 너무나 많은 것들을 RTL이 아우르고 있다.

이 책의 초점은 오브젝트 파스칼 언어다. 이 라이브러리들에 대한 가이드가 아니다. 여기서는 오직 몇 가지 클래스만 선택하여 집중하겠다. 핵심 역할을 하는 것들 또는 최근 몇 년 사이 도입되었기 때문에 개발자들 대부분이 알지 못하는 것들이다.

Classes 유닛 The Classes Unit

System.Classes 유닛은 오브젝트 파스칼 RTL 클래스 라이브러리(와 가상 라이브러리들)의 기반이다. 이 유닛은 다양한 클래스들의 거대한 모음이다. 특정 분야에 편중되지 않았다. 먼저 중요한 것들을 간단히 보고 난 후에 가장 중요한 것들을 깊이 분석하는 것이 좋겠다.

Classes 유닛 안의 클래스들 The Classes in the Classes Unit

Classes 유닛 안에 실제로 정의된 전체 클래스의 절반 정도를 간단한 나열해 놓았다.

- TList: 포인터들을 담는 핵심 리스트다. 정해진 타입이 없는 리스트로 종종 활용이 된다. 권장 사항은 TList<T>를 대신 사용하는 것이다. 14장에서 설명했다.
- TInterfaceList: 인터페이스들을 담는 쓰레드-안전한 리스트다. IInterfaceList를 구현한다. 다시 한번 살펴볼 가치가 있다 (하지만, 여기서는 다루지 않는다).
- Tbits: 매우 간단한 클래스다. 숫자나 기타 값의 안에 있는 개별 비트들을 조작한다. 시프트 *shift*, 이진 *or* *binary or*, 이진 *and* *binary and* 연산자 등을 사용하는 비트 조작에 비해 훨씬 더 고수준이다.
- TPersistent: 기초가 되는 클래스(TComponent의 조상 클래스)다. 다음 소단원에서 자세히 다룬다.
- TCollectionItem과 TCollection: 컬렉션 프로퍼티들을 정의하는 데 사용되는 클래스들이다. 즉 값들의 배열을 가지는 프로퍼티들을 위한 것이다. 이것들은 컴포넌트 개발자에게(또한 컴포넌트를 사용할 때 간접적으로) 중요하다. 일반 애플리케이션 개발자에게는 그다지 중요하지 않다.
- TStringList: 문자열을 담는 추상 리스트다. 이와 달리, TStringList는 기반 클래스인 TStringList 클래스의 실제 구현이며, 실제 문자열들을 위한 저장소를 제공한다. 각 항목에는 오브젝트를 첨부할 수 있다. 그리고 스트링 리스트를 이름/값 문자열 쌍을 위해 사용하는 표준 방법이다. 이 클래스에 대한 몇 가지 추가 정보는 이 장 끝에 있는 “스트링 리스트 사용하기” 소단원에 있다.
- TStream: 추상 클래스다. 순차적으로 접근하는 모든 바이트 나열을 표현한다. 다양한 저장 옵션이 있다 (메모리, 파일, 문자열, 소켓, BLOB 필드, 기타 등등). Classes 유닛에는 많은 특정 스트림 클래스들이 정의되어 있다 (THandleStream, TFileStream, TCustomMemoryStream, TMemoryStream, TBytesStream, TStringStream, TResourceStream 등). 다른 특정 스트림들은 다른 RTL 유닛들에 선언되어 있다. 스트림에 대한 소개는 이 장의 “스트림에 대한 소개” 소단원에서 읽을 수 있다.
- 스트리밍을 위한 저수준 컴포넌트 클래스들(TFile, TReader, TWriter, TParser 등): 주로 컴포넌트 작성자들이 사용한다. 심지어 자주 사용하지도 않는다.
- TThread 클래스: 플랫폼에 독립적인 *platform-independent* 멀티-쓰레딩 애플리케이션을 지원한다. 또한 비동기 연산을 위한 클래스인 TBaseAsyncResult도 있다.
- 옵저버 패턴 *observer pattern* 구현을 위한 클래스들: (예를 들어 시각적 라이브 바인딩 *live*

bindings에 사용되는 것으로) TObservers, TLinkObservers, TObserverMapping 등 있다.

- 사용자 지정 애트리뷰트들 용 클래스들: DefaultAttribute, NoDefaultAttribute, StoredAttribute, ObservableMemberAttribute 등.
- 기초가 되는 TComponent 클래스: 이 장의 뒷부분에서 다룬다. VCL과 FireMonkey에서 모든 시각적, 비시각적 컴포넌트들의 기반 클래스다.
- 액션과 액션 리스트 지원을 위한 클래스들: (액션은 UI 요소나 내부적으로 발행된 “명령”command의 추상화이다) TBasicAction, TBasicActionLink 등.
- TDataModule 클래스는 비시각적 컴포넌트 컨테이너를 나타낸다.
- 파일과 스트림 동작을 위한 고수준 인터페이스들: TTextReader와 TTextWriter, TBinaryReader와 TBinaryWriter, TStringReader와 TStringWriter, TStreamReader와 TStreamWriter 등. 이 클래스들도 이 장에서 다룬다.

TPersistent 클래스 The TPersistent Class

TObject 클래스는 매우 중요한 하위 클래스subclass를 하나 가지고 있다. TPersistent라고 불리며, 전체 라이브러리의 기초 중 하나다. 만약 여러분이 클래스의 메서드를 본다면, 그 중요성에 비해서 하는 일이 거의 없어 놀랄 것이다. TPersistent 클래스에는 핵심 요소가 있다. 이 클래스는 옵션 {M+}이 붙어서 정의되어 있다 이 옵션은 published 키워드를 활성화하는 역할을 한다. 10장에서 살펴봤다.

published 키워드는 프로퍼티 스트리밍에 필수적인 역할을 한다. 그래서 이 클래스의 이름이 TPersistent다. TPersistent를 상속받은 클래스들만이 게시된published 프로퍼티의 데이터 타입으로 사용될 수 있다. 오브젝트 파스칼 컴파일러의 최근 버전에 있는 RTTI의 확장이 다른 모델을 허용한다는 점이 사실이지만, VCL과 FMX 라이브러리 컴포넌트 스트리밍은 published 키워드와 {\$M+} 컴파일러 옵션의 역할에 여전히 기반한다.

참고 TPersistent에서 상속받지 않았으며 {\$M+} 컴파일러 플래그flag를 갖지 않은 클래스에 여러분이 만약 published 키워드를 추가한다면, 현재의 컴파일러를 사용하는 시스템은 적절한 지원을 어떤 식으로든 추가한다. 그리고 그것을 경고로 알려준다.

이 계층 구조에서 TPersistent 클래스의 구체적인 역할은 뭘까? 첫째, TComponent의 기반 클래스로 쓰인다. TComponent는 다음 소단원에 소개한다. 둘째, 프로퍼티 값으로 사용되는 데이터 타입들의 기반 클래스로 쓰인다. 그래서 그 프로퍼티들과 그 내부 구조가 올바르게 스트리밍 되도록streamed 한다. 예를 들면, 문자열, 비트맵, 폰트 및 기타 오브젝트들의 리스트를 표현하는 클래스들이다.

만약 TPersistent 클래스와 관련하여 가장 많이 활용되는 기능이 하위 클래스의 published 구역을 다루도록 “활성화activation”하는 일이라면, 이 클래스가 가진 몇 가지 흥미로운 메서드들을 깊이 볼 필요가 있다. 첫 번째는 Assign 메서드다. 한 인스턴스에

서 다른 인스턴스로 오브젝트 데이터의 복사본을 만드는 데 쓰인다(깊은 복사^{deep copy}다. 참조를 복사하는 것이 아니다). 지속성 있는^{persistent} 클래스를 프로퍼티 값으로 사용하려면 이 기능을 수동으로 구현해야 한다(왜냐하면 이 언어에는 자동 깊은 복사 연산이 없다). 두 번째는 그 반대 연산^{operation}인 AssignTo다. 이것은 보호된다. 이 두 메서드 그리고 이 클래스 안에 있는 몇 가지 다른 것들은 애플리케이션 개발자보다 컴포넌트 작성자가 주로 사용한다.

TComponent 클래스 The TComponent Class

TComponent 클래스는 컴포넌트 라이브러리들의 초석^{cornerstone}이다. 오브젝트 파스칼 컴파일러와 함께 쓰이는 경우가 가장 많은 것이 바로 컴포넌트 라이브러리다. 컴포넌트라는 개념은 기본적으로 디자인 시점에 몇 가지 추가 동작을 하는 클래스라는 개념이다. 고유한 스트리밍 능력이 있고(그래서 디자인 시점의 설정이 저장될 수 있다. 그리고 애플리케이션이 작동할 때 다시 복구될 수 있다). 그리고 PME(property-method-event) 모델을 가진다. PME에 대해서는 10장에서 설명했다.

이 클래스는 상당히 많은 표준 동작들과 기능들을 정의하고 있다. 그리고 자신만의 메모리 모델을 도입하고 있다. 그 모델의 기반은, 오브젝트 소유권^{ownership}이라는 개념, 컴포넌트 간 알림^{cross components notification} 등등 매우 많다. 전체 프로퍼티들과 메서드들을 완벽하게 분석하지는 못하지만, TComponent 클래스의 주요 기능들 몇 가지를 집중해서 볼 가치가 확실히 있다. RTL 안에서 중심 역할을 하기 때문이다.

TComponent 클래스의 또다른 중요한 기능이 있다. 가상 Create 생성자를 도입한다는 사실이다. 클래스의 고유한 생성자 코드를 여전히 호출하면서도 클래스 참조로부터 오브젝트를 생성하는 능력을 갖기 위해 매우 중요하다. 우리는 12장에서 이것을 다뤘다. 하지만, 이것은 오브젝트 파스칼 언어의 독특한 기능이다. 따라서 이해할 가치가 있다.

컴포넌트 소유권 Components Ownership

소유권^{Ownership} 메커니즘은 TComponent 클래스의 핵심 요소다. 만약 컴포넌트가 생성될 때 소유자 컴포넌트가 지정되면(가상 생성자에 파라미터로 전달), 그 소유자 컴포넌트는 자신이 가지고 있는 컴포넌트들을 소멸하는 책임을 진다. 요약하자면, 각 컴포넌트는 자신의 소유자에 대한 참조(Owner 프로퍼티)를 가진다. 뿐만 아니라, 자신이 소유하고 있는 컴포넌트들의 목록(Components 배열 프로퍼티)과 그 개수(ComponentCount 프로퍼티)도 가진다.

기본 설정으로, 여러분이 컴포넌트를 디자이너(폼, 프레임 혹은 데이터 모듈) 안에서 내려 놓는^{drop} 경우, 그것이 그 컴포넌트의 소유자로 간주된다. 여러분이 컴포넌트를 코드 안에서 생성하는 경우, 소유자를 명시하는 건 여러분에게 달렸다. 명시하지 않는 경우 nil이 전달된다. 그런 경우에는, 컴포넌트를 메모리에서 해제할 책임을 여러분 스스로가 지게 된다.

여러분은 Components와 ComponentCount 프로퍼티들을 사용해 그 컴포넌트(이 경우 AComp)가 소유하고 있는 컴포넌트들을 나열할 수 있다. 다음 코드와 같다:

```
var
  I: Integer;
begin
  for I := 0 to AComp.ComponentCount - 1 do
    AComp.Components[I].DoSomething;
```

혹은 네이티브 열거형 지원을 사용해서 코드를 작성할 수도 있다. 다음과 같다:

```
var
  ChildComp: TComponent;
begin
  for ChildComp in AComp do
    ChildComp.DoSomething;
```

컴포넌트가 소멸하면 (소유자가 있는 경우에) 소유자의 리스트에서 자신을 제거한다. 그리고 자신이 소유한 모든 컴포넌트를 소멸한다. 이 메커니즘은 오브젝트 파스칼의 메모리 관리에서 중요하다. 가비지 컬렉션 [garbage collection/쓰레기 수집](#)이 없기 때문에, 소유권이 메모리 관리 문제 대부분을 해결할 수 있다. 13장에서 부분적으로 살펴보았다.

언급한 대로, 일반적으로 폼이나 데이터 모듈의 모든 컴포넌트들은 그 폼과 데이터 모듈을 소유자로 가진다. 폼이나 데이터 모듈을 해제할 때, 그들이 가진 컴포넌트들도 같이 소멸한다. 이것이 컴포넌트가 스트림으로부터 생성될 때 일어나는 일이다.

컴포넌트 프로퍼티 [Component Properties](#)

(알림 [notification](#) 등 여기서 다루지 않은 다른 기능들을 포함한) 핵심적인 소유권 메커니즘 이외에도 모든 컴포넌트에는 게시된 [published](#) 프로퍼티 두 가지가 있다:

- Name: 문자열이며, 컴포넌트 이름이다. 컴포넌트를 동적으로 검색할 때 (소유자의 FindComponent 메서드를 호출함)와 폼의 필드가 참조하고 있는 컴포넌트를 연결할 때 사용한다. 소유자가 같은 모든 컴포넌트들은 유일한 이름(대소문자 구별 없이)을 가져야 한다. 하지만 그 이름이 비어 있을 수 있다. 두 가지 짧은 규칙이 있다: 알맞은 컴포넌트 이름을 붙여라. 그러면 여러분의 코드의 가독성을 높아진다. 또한 런타임에 컴포넌트의 이름을 바꾸면 안 된다(정말 끔찍할 부작용이 발생할 수도 있다는 것을 알고 있지 않다면 말이다).
- Tag: NativeInt 값이다(예전에는 Integer였다). 라이브러리에는 사용하지 않는다. 하지만, 여러분은 이것을 추가 정보를 컴포넌트에 연결하는데 사용할 수 있다. 이 타입은 포인터와 그리고 오브젝트 참조와 크기가 호환된다 [size-compatible](#). 그래서 그것들은 컴포넌트의 Tag에 종종 저장된다.

컴포넌트 스트리밍 [Component Streaming](#)

스트리밍 메커니즘은 FireMonkey와 VCL에서 모두에서 사용된다. FMX 또는 DFM

파일을 생성하기 위해 사용된다. 그 바탕은 TComponent 클래스다. 델파이의 스트리밍 메커니즘은 컴포넌트와 그 하위 컴포넌트들이 가지고 있는 게시된 `published` 프로퍼티와 이벤트들을 저장한다. DFM이나 FMX 파일 안에서 그 표현들을 얻을 수 있다. 또한 여러분이 디자이너에서 컴포넌트를 복사해서 텍스트 에디터에 붙여 넣으면 해당 표현을 얻을 수 있다.

그와 똑같은 정보를 런타임에 얻을 수 있게 해주는 메서드들이 있다. TStream 클래스의 WriteComponent와 ReadComponent, 같은 클래스의 ReadComponentRes와 WriteComponentRes, Treader와 TWriter의 ReadRootComponent와 WriteRootComponent가 있다. Treader와 TWriter는 컴포넌트 스트리밍을 다루는 것을 돕는 특별한 클래스들이다. 이들 연산은 일반적으로 폼 스트림의 이진 표현 `binary representation`을 사용한다. 여러분은 전역 프로시저 ObjectResourceToText를 사용해 폼 바이너리 표현을 텍스트 표현으로 변환하거나, ObjectTextToResource로 역변환을 할 수 있다.

한 가지 핵심 요소가 있다. 스트리밍은 컴포넌트의 게시된 `published` 프로퍼티들 전체의 묶음이 아니다. 스트리밍에 포함하는 것들은 다음과 같다:

- 컴포넌트의 게시된 프로퍼티들 중 기본 `default` 값이 아닌 값을 가진 것들 (다시 말해, 기본값은 저장하지 않는다. 크기를 줄이기 위해서다).
- 오직 `stored`로 표시(기본 설정임)되어 있는 게시된 프로퍼티들. `stored`가 `False`인 프로퍼티(혹은 `False`를 반환하는 함수는) 저장되지 않는다.
- 컴포넌트 프로퍼티들에 대응하지 않는 추가 엔트리들이 런타임에 추가될 수 있다 (DefineProperties 메서드를 오버라이딩 `overriding`해 추가함).

스트림 파일로부터 컴포넌트가 생성될 때, 다음 순서대로 진행된다:

- 컴포넌트의 가상 Create 생성자가 호출된다 (알맞은 초기화 코드가 실행된다)
- 프로퍼티들과 이벤트들이 스트림으로부터 적재된다(이벤트의 경우, 메서드 이름을 메모리의 실제 메모리 주소로 다시 매핑된다)
- Loaded 가상 메서드가 호출되어 적재를 마무리한다. (그리고 컴포넌트에 사용자가 추가로 지정한 것들이 처리된다. 이 시점에는 이미 스트림으로부터 적재된 프로퍼티 값들을 사용한다)

현대적인 파일 접근 Modern File Access

오브젝트 파스칼에는 조상인 파스칼 언어에서 전해져 온 파일 처리 키워드들과 핵심 언어 메커니즘이 여전히 있다. 이것들은 기본적으로 오브젝트 파스칼이 도입되면서 더 이상 사용되지 않는다. 따라서 이 책에서는 건드리지 않겠다. 대신 이 소단원에서 다룰 것은 파일을 처리하는 몇 가지 현대적인 기법들이다. IOUtils 유닛, 스트림 클래스들, 읽기 `reader` 클래스와 쓰기 `writer` 클래스를 소개한다.

입력/출력 유틸리티 유닛 The Input/Output Utilities Unit

System.IOUtils 유닛이 런타임 라이브러리에 추가된 것은 비교적 나중이다. 여기에는 레코드 세 개 즉 TDirectory, TPath, TFile이 정의되어 있다. 이 레코드들 안에는 대부분 클래스 메서드들이 정의되어 있다: TDirectory는 폴더를 탐색한다. 그리고 그 안에 있는 파일들과 하위 폴더들을 찾을 때 쓴다. 그러나, TPath와 TFile은 그 차이를 명확하게 알지 못할 수 있다. 먼저, TPath는 파일 이름과 디렉터리 이름을 조작할 때 사용한다. 그 메서드들은 드라이브 추출하기, 경로 [path](#)나 확장자를 빼고 파일 이름 추출하기 등과 같은 일을 한다. 뿐만 아니라, UNC 경로를 조작하는 데도 사용된다. 다음으로, TFile 레코드는, 여러분이 파일의 타임스탬프 [timestamp](#)와 속성 [attribute](#)을 확인할 수 있게 해준다. 뿐만 아니라, 파일에 쓰거나 파일을 복사하는 등의 조작을 할 수도 있다. 예제를 보자. IoFilesInFolder 예제는 주어진 폴더의 모든 하위 폴더를 추출한다. 그리고 그 폴더 아래 있는 파일들 중에서 주어진 확장자에 해당하는 것들을 모두 가져온다.

하위 폴더 추출하기 Extracting Sub-folders

이 프로그램은 사용자가 지정한 폴더의 하위 폴더 [sub-folder](#)들로 리스트 박스를 채운다. 그러기 위해 TDirectory 레코드의 GetDirectories 메서드를 사용한다. 이 메서드는 파라미터로 TSearchOption.soAllDirectories 값을 받는다. 그리고 그 결과는 여러분이 열거할 수 있는 문자열 배열이다:

```
procedure TFormIoFiles.BtnSubfoldersClick(Sender: TObject);
var
    PathList: TStringDynArray;
    StrPath: string;
begin
    if TDirectory.Exists(EdBaseFolder.Text) then
        begin
            ListBox1.Items.Clear;
            PathList := TDirectory.GetDirectories(EdBaseFolder.Text,
                TSearchOption.soAllDirectories, nil);
            for StrPath in PathList do
                ListBox1.Items.Add(StrPath);
            end;
        end;
```

파일 탐색하기 Searching Files

프로그램의 두번째 버튼은 그 폴더들 안에 있는 모든 PAS 파일을 나열한다. 각 폴더를 훑기 위해서, 주어진 마스크 [mask](#)를 반영하는 GetFiles를 호출한다. 여러분은 더 복잡한 필터링을 할 수도 있다. TFilterPredicate 타입의 익명 메서드를 GetFiles의 오버로드된 버전에 전달하면 된다.

이 예제는 마스크를 반영하는 더 간단한 필터링을 사용한다. 그리고 내부의 스트링 리스트를 채운다. 그리고 이 스트링 리스트의 요소 [element](#)들에서 전체 경로를 제거하여 파일 이름만 남긴 후 사용자 인터페이스로 복사된다.

GetDirectories 메서드를 호출하면 하위 폴더들만 얻는다. 현재 폴더는 얻을 수 없다.

그래서 이 프로그램은 현재 폴더 안을 먼저 검색하고 이어서 각 하위 폴더를 검색한다:

```
procedure TFormIoFiles.BtnPasFilesClick(Sender: TObject);
var
  PathList, FilesList: TStringDynArray;
  StrPath, StrFile: string;
begin
  if TDirectory.Exists(EdBaseFolder.Text) then
    begin
      // 정리
      ListBox1.Items.Clear;

      // 주어진 폴더에서 검색
      FilesList := TDirectory.GetFiles(EdBaseFolder.Text, '*.pas');
      for StrFile in FilesList do
        SFilesList.Add(StrFile);

      // 모든 하위 폴더에서 검색
      PathList := TDirectory.GetDirectories(EdBaseFolder.Text,
        TSearchOption.soAllDirectories, nil);
      for StrPath in PathList do
        begin
          FilesList := TDirectory.GetFiles(StrPath, '*.pas');
          for StrFile in FilesList do
            SFilesList.Add(StrFile);
          end;

      // 이제 (경로 없는) 파일 이름을 리스트박스에 복사
      for StrFile in SFilesList do
        ListBox1.Items.Add(TPath.GetFileName(StrFile));
    end;
  end;
```

마지막 줄에서는, TPath의 GetFileName 함수를 사용한다. 파일의 전체 경로에서 파일 이름을 추출한다. TPath 레코드에는 다른 흥미로운 메서드들도 있다. GetTempFileName, GetRandomFileName, 경로를 병합-merge하는 메서드, 유효valid하거나 잘못된illegal 문자가 들어있는지 확인하는 것들 등등이 있다.

스트림에 대한 소개 Introducing Streams

IOutils 유닛이 파일을 찾고 조작하기 위해 있다면, 여러분이 파일 (혹은 다른 유사한 순차 접근하는sequentially accessed 데이터 구조들)을 읽고 쓸 때는 TStream 클래스와 그 많은 자손 클래스들을 사용할 수 있다. TStream 추상 클래스는 모든 스트림 클래스가 공유하는 기본 인터페이스로 주요 Read와 Write 메서드와 함께 몇 개의 프로퍼티(Size와 Position)들을 가진다. 이 클래스로 표현된 개념은 순차 접근sequential access이다. 여러분이 바이트 수만큼 읽고 쓸 때마다, 현재 위치가 그 숫자만큼 올라간다. 대부분의 스트림에서, 이 위치를 뒤로 옮길 수 있으나, 단방향unidirectional 스트림도 있을 수 있다.

자주 쓰이는 스트림 클래스 Common Stream Classes

이미 언급한 대로, Classes 클래스는 몇 가지 구체적인 스트림 클래스들을 정의한다. 아래에 있는 것들도 여기에 해당된다:

- THandleStream: 파일 핸들을 통해 참조되는 디스크 파일 스트림을 정의한다.
- TFileStream: 파일 이름에 의해 참조되는 디스크 파일 스트림을 정의한다.
- TBufferedFileStream: 최적화된 디스크 파일 스트림이다. 메모리 버퍼를 사용해서 추가 성능을 낸다. 이 스트림 클래스는 Delphi 10.1에서 도입되었다.
- TMemoryStream: 메모리 안 데이터의 스트림이다. 포인터를 사용하여 접근할 수 있다.
- TBytesStream: 메모리안에 있는 바이트의 스트림을 나타낸다. 여러분이 바이트 배열 처럼 접근할 수도 있다.
- TStringStream: 스트림을 메모리 안에서 문자열과 연결한다.
- TResourceStream: 리소스 데이터를 읽을 수 있는 스트림을 정의한다. 그 리소스 데이터는 애플리케이션의 실행 파일에 연결된 것이다.
- TPointerStream: 해당 TStream 인터페이스를 사용하여 메모리 내 데이터 블록을 읽고 쓸 수 있도록 한다. 포인터 위치와 메모리 블록 크기를 알려주기 때문이다. 이 클래스는 Delphi 11에서 추가되었다.

스트림 사용하기 Using Streams

스트림을 만들고 사용하기는 간단하다. 특정 타입의 변수를 만들고 컴포넌트의 메서드를 호출해서 파일에서 내용을 가져오면 된다. 예를 들어, 스트림과 메모 컴포넌트가 주어진다면, 여러분은 다음과 같이 작성할 수 있다:

```
AStream := TFileStream.Create(FileName, fmOpenRead);
Memo1.Lines.LoadFromStream(AStream);
```

이 코드에서 볼 수 있듯이, 파일 스트림을 위한 Create 메서드는 두 개의 파라미터를 가진다: 파일의 이름 그리고 요청하는 접근 모드 access mode를 가리키는 몇몇 플래그 flag다. 언급한 대로 스트림은 읽고 쓰는 동작을 지원한다. 하지만, 그것들은 더 저-수준 low-level이다. 그래서 읽기 클래스와 쓰기 클래스들을 사용하는 것을 권한다. 다음 소단원에서 설명한다. 위에서는 그 대신, 스트림을 직접 사용했다. 이 경우 복잡한 연산들의 집합이 수행된다. 위 코드에서는 전체 스트림을 적재한다. 예를 들어 스트림을 한 곳에서 다른 곳으로 복사하는 코드는 아래와 같이 복잡한 연산들의 집합이 수행된다:

```
procedure CopyFile(SourceName, TargetName: String);
var
  Stream1, Stream2: TFileStream;
begin
  Stream1 := TFileStream.Create(SourceName, fmOpenRead);
  try
    Stream2 := TFileStream.Create(TargetName,
      fmOpenWrite or fmCreate);
    try
      Stream2.CopyFrom(Stream1, Stream1.Size);
    finally
      Stream2.Free;
    end
  finally
    Stream1.Free;
  end
end;
```


Reader와 Writer 사용하기 Using Readers and Writers

스트림에 쓰고 읽는 매우 좋은 방식은 RTL에 있는 읽기 [reader](#) 클래스와 쓰기 [writer](#) 클래스를 사용하는 것이다. Classes 유닛에 정의된 여섯 개의 읽고 쓰는 클래스들이 있다:

- TStringReader와 TStringWriter: 메모리 안의 문자열을 대상으로 작업한다 (직접 혹은 TStringBuilder를 사용한다)
- TStreamReader와 TStreamWriter: 일반적인 스트림을 대상으로 작업한다(파일 스트림, 메모리 스트림, 기타 TStream에서 파생된 클래스들을 대상으로 함)
- TBinaryReader와 TBinaryWriter: 텍스트가 아닌 이진 데이터 [binary data](#)에 작업한다

텍스트 [읽기](#) 클래스에는 기본적인 읽기 기법들 몇 가지가 구현되어 있다:

```
function Read: Integer; overload;
function ReadLine: string;
function ReadToEnd: string;
```

텍스트 쓰기 클래스에는 두 종류의 오버로드된 연산들이 있다. 줄바꿈 구분자가 없는 (Write) 혹은 있는(WriteLine) 연산이다. 첫 번째 종류를 보자면 아래와 같다:

```
procedure Write(Value: Boolean); overload;
procedure Write(Value: Char); overload;
procedure Write(const Value: TCharArray); overload;
procedure Write(Value: Double); overload;
procedure Write(Value: Integer); overload;
procedure Write(Value: Int64); overload;
procedure Write(Value: TObject); overload;
procedure Write(Value: Single); overload;
procedure Write(const Value: string); overload;
procedure Write(Value: Cardinal); overload;
procedure Write(Value: UInt64); overload;
procedure Write(const Format: string; Args: array of const); overload;
procedure Write(Value: TCharArray; Index, Count: Integer); overload;
```

Text Reader 와 Text Writer Text Readers and Writers

스트림에 쓰기 위해, TStreamWriter 클래스는 스트림을 사용하거나 생성한다. 스트림 생성자에게는 파라미터로 파일 이름, create/append 옵션, 유니코드 인코딩을 전달한다.

그래서 다음과 같이 쓸 수 있다 (ReaderWriter 예제에서 발췌함):

```
var
    Sw: TStreamWriter;
begin
    Sw := TStreamWriter.Create( 'test.txt',
        False, TEncoding.UTF8);
    try
        Sw.WriteLine( 'Hello, world');
        Sw.WriteLine( 'Have a nice day');
        Sw.WriteLine(Left);
    finally
        Sw.Free;
    end;
```


TStreamReader를 읽기 위해, 여러분은 다시 스트림이나 파일을 대상으로 작업할 수 있다 (아래 경우는 UTF BOM 마커에서 해당 인코딩을 감지하는 선택을 지정했다):

```
var
  SR: TStreamReader;
begin
  SR := TStreamReader.Create('test.txt', True);
  try
    while not SR.EndOfStream do
      Memo1.Lines.Add(SR.ReadLine);
  finally
    SR.Free;
  end;
```

EndOfStream의 상태를 어떻게 확인하는지 잘 보자. 텍스트 스트림(혹은 문자열)들을 직접 사용하는 것에 비해, 이 클래스들은 특히 사용하기가 편하고, 성능이 좋다.

Binary Reader 와 Binary Writer Binary Reader and Writer

TBinaryReader와 TBinaryWriter 클래스는 텍스트 파일이 아니라 이진 데이터용이다. 이들 클래스는 일반적으로 스트림(파일 스트림 혹은 모든 종류의 메모리 내 스트림, 예: 소켓, 데이터베이스 BLOB 필드 등)을 캡슐화 한다. 그리고 오버로드된 Read와 Write 메서드를 가진다.

그 (더 간단한) 예로, BinaryFiles 프로그램을 작성했다. 앞에서는 파일에 이진 요소 (프로퍼티 값과 현재 시각)들을 쓴다. 그리고 도로 읽어서 프로퍼티 값에 대입한다:

```
procedure TFormBinary.BtnWriteClick(Sender: TObject);
var
  BW: TBinaryWriter;
begin
  BW := TBinaryWriter.Create('test.data', False);
  try
    BW.Write(Left);
    BW.Write(Now);
    Log('File size: ' + IntToStr(BW.BaseStream.Size));
  finally
    BW.Free;
  end;
end;

procedure TFormBinary.BtnReadClick(Sender: TObject);
var
  BR: TBinaryReader;
  ATime: TDateTime;
begin
  BR := TBinaryReader.Create('test.data');
  try
    Left := BR.ReadInt32;
    Log('Left read: ' + IntToStr(Left));
    ATime := BR.ReadDouble;
    Log('Time read: ' + TimeToStr(ATime));
  finally
    BR.Free;
  end;
end;
```


이들 읽기와 쓰기 클래스들을 사용하는 핵심 규칙이 있다. 여러분이 쓴 순서와 똑같이 데이터를 읽어야 하며, 그렇지 않으면 데이터를 망칠 수 있다는 것이다. 사실, 오직 각 필드의 이진 데이터만 저장된다. 필드 자체의 정보는 저장되지 않는다. 하지만 여러분이 파일에 데이터와 메타데이터를 끼워 넣는 것^{interposing}을 막지는 않는다. 예를 들어, 해당 타입에 대한 데이터를 쓰기 전에 다음 데이터 타입의 크기를 저장할 수 있다.

문자열과 스트링 리스트 만들기 Building Strings and String Lists

파일과 스트림을 봤으니, 약간의 시간을 들여 문자열과 문자열의 리스트를 조작하는 방법에 집중하겠다. 이것들은 매우 흔한 동작들이다. 여기에 집중하는 많은 기능들이 RTL 안에 있다. 여기서는 조금만 소개한다.

TStringBuilder 클래스 The TStringBuilder class

6장에서 봤듯이, 다른 언어들과 달리 오브젝트 파스칼은 문자열 직접 이어 붙이기 ^{concatenation}를 완벽히 지원하고 오히려 빠르다. 그런데, 이 언어의 RTL에는, 서로 다른 데이터 타입의 조각들을 문자열로 조립하는 TStringBuilder라는 특별한 클래스가 있다.

TStringBuilder 클래스를 사용하는 간단한 예제로, 다음 코드 조각을 보자:

```
var
  SBuilder: TStringBuilder;
  Str1: string;
begin
  SBuilder := TStringBuilder.Create;
  try
    SBuilder.Append(12);
    SBuilder.Append('Hello');
    Str1 := SBuilder.ToString;
  finally
    SBuilder.Free;
  end;
end;
```

여기서 TStringBuilder 오브젝트를 생성하고 소멸해야 하는 것을 보자. 위에서 알 수 있는 또 다른 요소로 Append 함수에 파라미터들로 전달할 수 있는 서로 다른 많은 데이터 타입이 있다.

TStringBuilder 클래스의 다른 흥미로운 메서드로는 AppendFormat(내부에서 Format을 호출한다) 그리고 sLineBreak값을 추가하는 AppendLine이 있다. Append와 더불어, 상응하는 오버로드된 Insert 메서드들이 있으며, Remove와 몇몇 Replace 메서드들도 있다.

참고 TStringBuilder 클래스는 훌륭한 인터페이스를 가지고 좋은 유용성을 제공한다. 성능 측면에서 보면, 변경 불가능한 문자열을 정의하여 순수하게 문자열을 연결할 경우 매우 나쁜 성능을 보이는 다른 프로그래밍 언어와 달리 오브젝트 파스칼 표준 문자열 연결 ^{concatenation} 및 형식화 ^{formatting} 함수를 사용하면 더 나은 결과를 제공할 수 있다.

StringBuilder에서 메서드를 이어 붙이기 Method Chaining in StringBuilder

TStringBuilder 클래스는 매우 고유한 특징이 있다. 그 클래스의 메서드들 대부분이 함수들이고 그 함수를 적용한 오브젝트를 다시 반환한다.

이 코딩 관행 덕분에 메서드 이어 붙이기 method chaining가 가능하다. 즉, 앞에 있는 메서드에서 반환한 오브젝트의 메서드를 호출할 수 있다. 다음과 같이 쓰는 대신:

```
SBuilder.Append(12);
SBuilder.AppendLine;
SBuilder.Append( 'Hello ' );
```

여러분은 이렇게 쓸 수 있다:

```
SBuilder.Append(12).AppendLine.Append( 'Hello ' );
```

또 이런 형태로도 가능하다:

```
SBuilder.
  Append(12).
  AppendLine.
  Append( 'Hello ' );
```

나는 이 구문을 맨 위의 원래 구문보다 더 좋아하는 경향이 있다. 하지만, 이는 단지 문법 상의 조미료 syntactic sugar일 뿐이며 어떤 사람들은 각 줄마다 오브젝트가 적힌 원래 버전을 선호할 수도 있다. 하지만, 명심해야 할 점이 있다. 여러 Append 메서드들을 호출했을 때 반환되는 것은 어떠한 경우에도 새 오브젝트가 아니다(그래서 잠재적인 메모리 누수가 없다). 여러분이 그 메서드를 불러낸 그 오브젝트를 그대로 반환한다.

스트링 리스트 사용하기 Using String Lists

문자열들의 리스트들은 많은 시각화 컴포넌트에서 사용되는 매우 흔한 추상화이다. 또한, 문자열의 컬렉션을 조작하는 방법으로도 사용한다. 문자열의 리스트를 처리하는 두 개의 주요 클래스가 있다:

- TStrings: 모든 형태의 스트링 리스트에 대한 추상 클래스다. 그 저장소 구현과는 상관이 없다. 이 클래스는 문자열의 추상 리스트 하나를 정의한다. 그런 이유로, TStrings 오브젝트들은 컴포넌트의 프로퍼티들로만 사용한다. 문자열 자체를 저장하는 데에 적합한 프로퍼티에 말이다.
- TStringList: TStrings의 하위 클래스다. 문자열들의 리스트를 그 저장소와 함께 정의한다. 이 클래스는 프로그램에서 문자열들의 리스트를 정의하는데 쓸 수 있다.

두 스트링 리스트 클래스에는 미리 준비된 ready-to-use 메서드들도 있다. 텍스트 파일에 내용을 저장하거나 적재하는 SaveToFile과 LoadFromFile이다 (유니코드가 완전하게 활성화되어 있음). 리스트를 순회하려면, 여러분은 인덱스에 기반해 간단한 for 문을 사용해 그 리스트가 배열인 것처럼 쓸 수 있다. 혹은 for-in 열거를 쓸 수 있다.

런타임 라이브러리는 꽤 방대하다 The Run-Time Library is Quite Large

오브젝트 파스칼 코드에서 여러분이 사용할 수 있는 RTL들은 훨씬 더 많다. 이것들은 여러분이 수많은 핵심 기능들을 여러 운영체제를 타겟으로 할 수 있도록 한다. 전체 런타임 라이브러리를 자세히 다루려면, 이 책 전체 분량만큼 한 권을 더 써도 부족하다.

만일 라이브러리의 핵심 부분만 고려한다면, 즉 `System` 네임스페이스(namespace)는, 다음 유닛들을 포함한다(정말 드물게 사용하는 것들은 배제했다):

- `System.Actions`: 액션 아키텍처의 핵심 지원을 담고 있다. 액션은 사용자의 명령이 사용자 인터페이스 계층에서 연결되면서도 추상화되는 `abstracted` 방법이다.
- `System.AnsiStrings`: ANSI 문자열을 처리하는 오래된 함수들이 있다 (오직 윈도우 Windows에서만 작동한다). 6장에서 다뤘다.
- `System.Character`: 유니코드 문자들(Char 타입)을 위한 내장된 타입 헬퍼들이 있다. 3장에서 다뤘다.
- `System.Classes`: 핵심 시스템 클래스들을 제공한다. 이 유닛은 이 장의 첫 부분에서 자세히 다뤘다.
- `System.Contnrs`: 넓고, 제네릭인 아닌 컨테이너 클래스들이 있다. 오브젝트 리스트, 딕셔너리, 큐, 스택 등이다. 가능하면 같은 클래스의 제네릭 버전 사용을 권한다.
- `System.ConvUtils`: 서로 다른 측정 단위들 간의 변환 유틸리티 라이브러리가 있다.
- `System.DateUtils`: 날짜와 시간 값을 처리하는 함수들이 있다.
- `System.Devices`: 시스템 장비(GPS, 가속계, 기타 등등)들과 인터페이스 한다.
- `System.Diagnostics`: 시험할 코드의 경과 시간을 정확히 측정하는 레코드 구조를 정의하고 있다. 이 책에서도 필요할 때 가끔 사용했다.
- `System.Generics`: 사실 두 개의 유닛으로 나뉘져 있다. 하나는 제네릭 컬렉션을 위한 것이고, 다른 하나는 제네릭 타입을 위한 것이다. 이 유닛들은 14장에서 다뤘다.
- `System.Hash`: 해시 값 정의를 위한 핵심 지원이 들어 있다.
- `System.ImageList`: 추상화되고, 라이브러리에 독립적인 구현이 있다. 이미지들의 리스트들과 단일 이미지의 부분들을 하나의 컬렉션의 요소들로 관리한다.
- `System.IniFiles`: INI 구성 파일을 처리하는 인터페이스를 정의한다. INI 파일은 종종 윈도우에서 보게 된다.
- `System.IOUtils`: 파일 시스템 접근(파일, 폴더, 경로들)을 위한 레코드들을 정의한다. 이 장의 앞부분에서 다뤘다.
- `System.JSON`: 흔히 사용하는 자바스크립트 오브젝트 표현(JavaScript Object Notation) 줄여서 JSON 데이터를 처리하는 핵심 클래스들이 있다.
- `System.Math`: 수학적인 연산을 위한 함수들을 정의한다. 삼각함수, 금융 관련 함수 등이 있다. 이 네임스페이스 아래에 다른 유닛들도 있다. 벡터, 행렬 등이 해당된다.
- `System.Messaging`: 서로 다른 운영체제들을 위한 메시지 제어를 제공한다.
- `System.NetEncoding`: 몇 가지 흔한 네트워크 인코딩(base64, HTML, URL 등)에 대한 처리가 있다.

- `System.RegularExpressions`: 정규 표현식 [regular expression](#) (*regex*) 지원을 정의한다.
- `System.Rtti`: RTTI 클래스들의 전체 세트가 들어 있다. 16장에서 다뤘다.
- `System.StrUtils`: 핵심적이고 전통적인 문자열 처리 함수들이 있다.
- `System.SyncObjs`: 멀티-쓰레드 애플리케이션의 동기화를 위한 클래스들 몇 가지가 정의되어 있다.
- `System.SysUtils`: 시스템 유틸리티의 기본 모음집이 있다. 가장 전통적인 것들 몇 개도 여기에 있다. 그것들은 델파이 초기부터 있던 것들이다.
- `System.Threading`: 꽤 최근의 병렬 프로그래밍 라이브러리의 인터페이스, 레코드, 클래스들이 들어 있다.
- `System.Types`: 몇 가지 추가적인 핵심 데이터 타입들이 있다. `TPoint`, `TRectangle`, `TSize` 레코드, `TBitConverter` 클래스와 기타 RTL에서 사용하는 많은 기본 데이터 타입들이 있다.
- `System.TypeInfo`: 오래된 RTTI 인터페이스가 정의되어 있다. 역시 16장에서 다뤘다. 기본적으로, `System.Rtti` 유닛에 있는 것들이 이것들보다 우월하다.
- `System.Variants`와 `System.VarUtils`: 배리언트들을 가지고 작업하는 함수들이 있다 (배리언트 [variant](#)s는 5장에서 다뤘던 언어 기능이다).
- `System.Zip`: 보통의 파일 압축 및 압축해제 라이브러리를 위한 인터페이스를 제공한다.

또한 RTL에는 다른 부분들도 많다. `System` 네임스페이스의 하위 영역들도 그렇다. `System` 네임스페이스의 하위 영역들은 저마다 여러 개의 유닛들을 가진다 (상황에 따라 꽤 많은 유닛들을 가지기도 한다. 예: `System.Win` 네임스페이스). `System` 네임스페이스의 하위 영역들 중 몇 개를 꼽아보면, HTTP 클라이언트(`System.Net`), 사물 인터넷 지원 (`System.Beacon`, `System.Bluetooth`, `System.Sensors`, `System.Tether`) 등이 있다. 또한 (델파이가 지원하는 모든 운영체제들에 대한 인터페이스를 위해) 번역되어 있는 API 들과 헤더 파일들도 역시 `System` 네임스페이스의 하위 영역 안에 있다.

바로 사용할 수 있는 RTL 함수들, 타입들, 레코드들, 인터페이스들, 클래스들이 매우 많다. 잘 알아두면, 오브젝트 파스칼의 능력을 더 잘 활용할 수 있다. 도움말 문서의 `system` 부분에서 더 많은 내용을 살펴보기 바란다.

글을 마치며

18장은 이 책의 끝을 표시하며, 세 개의 부록을 남겨두었다. 이 책은 원래 오브젝트 파스칼 언어에만 초점을 맞춘 나의 첫 번째 책이었으며, 나는 시간이 지나도 이 책을 계속 업데이트하고 책의 내용과 소스 코드를 유지하기 위해 최선을 다하고 있다. 델파이 10.1 베를린에 대한 PDF 전용 업데이트가 있었으며, 10.4 시드니에 대응한 새 버전의 인쇄가 있었고, 여러분이 지금 읽고 있는 델파이 11 알렉산드리아의 PDF 전용 업데이트가 있다.

GitHub(10.4 버전과 동일한 저장소 기반)에서 책의 최신 소스 코드를 가져오는 방법에 대한 소개를 다시 참조하고 향후 정보 및 업데이트를 보려면 도서의 웹 사이트나 나의 블로그를 방문하면 된다.

여러분이 이 책을 즐겁게 읽었기를 바란다. 나도 이 책을 쓰고 델파이에 관해 글을 쓰면서 지난 25년 동안 즐거웠다. 델파이와 함께 즐겁게 코딩 하세요!

끝.

이 책의 마지막 소단원에는 다룰 만한 가치가 있는 결가지 문제지만 이 책의 흐름에 맞지 않았던 문제들에 집중하는 몇 가지 부록이 있다. Pascal 및 Object Pascal 언어에 대한 간략한 역사와 용어집도 있다.

부록 요약

부록 A: 오브젝트 파스칼의 진화 [The Evolution of Object Pascal](#)

부록 B: 용어집 [Glossary of Terms](#)

a: 오브젝트 파스칼의 진화

오브젝트 파스칼은 스마트폰과 태블릿부터 데스크탑과 서버에 이르기까지 점점 더 다양해지는 오늘날의 컴퓨팅 장치를 위해 만들어진 언어다. 이 언어는 갑자기 튀어나온 것이 아니다. 현대 프로그래머들이 선택할 수 있는 도구가 되도록 단단한 기초 위에 세심하게 설계되었다. 이는 프로그래밍 속도와 결과 프로그램의 속도, 구문의 명확성 및 표현력 사이의 거의 이상적인 균형을 제공한다.

오브젝트 파스칼의 기반은 파스칼 프로그래밍 언어 계통이다. 구글의 Go, 애플의 Objective-C가 C언어에 기반한 언어인 것처럼, 오브젝트 파스칼은 파스칼에 기반한다. 여러분은 언어의 이름에서 이것을 알 수 있었을 것이다.

이 짧은 부록은 파스칼, 터보 파스칼, 델파이 파스칼, 그리고 오브젝트 파스칼의 언어 계통과 실제 도구들에 대한 간단한 요약에 포함한다. 언어를 배우기 위해 꼭 이 책을 읽을 필요는 없지만, 언어의 진화와 오늘날의 위치를 이해하는 것은 확실히 가치가 있다.

현재 우리가 엠바카데로 개발 도구들 안에서 사용하고 있는 오브젝트 파스칼 프로그래밍 언어는 1995년에 볼랜드가 새로운 시각화 개발 환경으로 델파이를 도입했을 때 발명되었다. 첫 번째 오브젝트 파스칼 언어는 일반적으로 터보 파스칼 [Turbo Pascal](#)로 참조되는 터보 파스칼 제품에서 이미 사용 중인 언어에서 확장되었다.

볼랜드가 파스칼을 발명한 것은 아니며, 그저 매우 유명하게 만드는 데 도움이 되었을 뿐이고, C언어와 비교해서 존재했던 몇 가지 한계를 극복하기 위해 그 기반을 확장했다.

다음 소단원에서는 Wirth의 Pascal부터 ARM 칩 및 모바일 장치용 최신 LLVM 기반 Delphi의 Object Pascal 컴파일러까지 언어의 역사를 다룬다.

Wirth의 파스칼 언어

파스칼 언어는 원래 1971년에 스위스 취리히 연방 공과대학교의 교수였던 Niklaus Wirth가 설계하였다. Wirth의 전체 전기 [biography](http://www.cs.inf.ethz.ch/~wirth)는 <http://www.cs.inf.ethz.ch/~wirth>에서 찾을 수 있다.

파스칼은 교육 목적에서 알골(Algol) 언어를 간략한 버전으로 바꿔 설계하였다. 알골은 1960년에 만들어졌다. 파스칼이 개발되었을 때, 많은 프로그래밍 언어들이 존재했으나, 몇 개만이 널리 쓰이고 있었다: 포트란(FORTRAN), 어셈블러, 코볼(COBOL), 그리고 베이직(BASIC)이었다. 그때 이 새로운 언어의 핵심 아이디어는 데이터 타입, 변수 선언, 그리고 구조화된 프로그램 제어에 대한 강력한 개념을 통해 관리되는 순서였다. 또한 이 언어는 교육 도구, 즉 모범 사례를 사용한 프로그래밍 교육을 위해 설계되었다.

말할 필요도 없이 Wirth의 Pascal의 핵심 교리는 여전히 Pascal 구문을 기반으로 하는 언어를 훨씬 넘어 모든 프로그래밍 언어의 역사에 큰 영향을 끼쳤다. 프로그래밍 언어 교육의 경우, 학교와 대학에서는 프로그래밍의 핵심 개념을 학습하는 데 어떤 언어가 더 도움이 되는지 알아보기보다는 다른 기준(예: 구직 요청, 개발 도구 공급업체의 기부 등)을 따르는 경우가 굉장히 많다. 하지만 그건 다른 이야기이다.

터보 파스칼

터보 파스칼 Turbo Pascal이라고 불리는 볼랜드의 세계적으로 유명한 파스칼 컴파일러는 1983년에 출시되었다. 볼랜드는 Jensen과 Wirth가 "Pascal: User Manual and Report"에서 정의한 사양에 따라 컴파일러를 구현했다. 터보 파스칼 컴파일러는 역대가장 많이 팔린 컴파일러 시리즈 중 하나였으며 단순성, 성능 및 가격의 균형 덕분에 PC 플랫폼에서 이 언어를 특히 인기 있게 만들었다. 컴파일러의 원작자는 나중에 Microsoft에서 배포한 매우 인기 있는 C# 및 TypeScript 프로그래밍 언어의 아버지인 Anders Hejlsberg였다.

터보 파스칼은 코드를 편집하고(WordStar 호환 편집기에서) 컴파일러를 실행하고 오류를 확인하면서 해당 오류가 포함된 줄로 돌아갈 수 있는 통합 개발 환경(Integrated Development Environment, IDE)을 도입했다. 지금은 당연하게 들리지만 이전에는 편집기를 종료하고 DOS로 돌아가서: 명령줄 컴파일러를 실행하고 오류 줄을 기록한 다음 편집기를 열고 오류 줄을 찾아야 했다.

게다가 마이크로소프트의 파스칼 컴파일러는 몇 백 달러에 팔았는데, 볼랜드는 터보 파스칼을 49달러에 팔았다. 터보 파스칼의 수 년간의 성공은 마이크로소프트가 결국 파스칼 컴파일러 제품을 포기하는 데 기여했다.

엠바카데로 개발자 네트워크의 *Museum* 섹션에서 볼랜드 터보 파스칼의 복사본을 실제로 다운로드할 수 있다:

<http://edn.embarcadero.com/museum>

역사 원래의 파스칼 언어 이후, Niklaus Wirth는 파스칼 구문의 확장인 Modula-2 언어를 설계했다. 지금은 거의 잊혀진 언어이지만, 초기 터보 파스칼과 오늘날의 오브젝트 파스칼에 있는 유닛 개념과 매우 유사한 모듈화(modularization) 개념을 처음으로 도입했었다. Modula-2의 추가 확장으로, 오브젝트 파스칼과 유사한 객체 지향 기능을 갖춘 Modula-3이 있다. 그러나 Modula-3은 Modula-2보다 훨씬 덜 사용되었다. 파스칼 언어를 사용하는 상업용 개발 대부분은 볼랜드와 애플의 컴파일러로 이동했다. 그런데, 애플이 Objective-C를 위해 오브젝트 파스칼을 포기하면서 볼랜드가 파스칼 언어를 거의 독점하게 되었다.

초기 델파이의 오브젝트 파스칼

언어를 객체 지향 프로그래밍(OOP) 영역으로 점차 확장한 9개의 버전의 터보 파스칼 및 볼랜드 파스칼 컴파일러 이후, 볼랜드는 1995년에 델파이를 출시하여 파스칼을 시각적 프로그래밍 언어로 전환하고 아마도 최고의 Windows용 라이브러리일 시각적 컴포넌트 라이브러리(즉 VCL)를 개발했다.

델파이는 Borland Pascal with Objects 컴파일러(터보 파스칼의 마지막 버전)를 포함하여 다른 오브젝트 파스칼과는 다른 많은 객체 지향 확장을 포함하여 다양한 방법으로 파스칼 언어를 확장했다.

참고 여러분은 델파이 제품 출시에 관해 <https://delphi.embarcadero.com/>에서 더 많은 것을 볼 수 있고, <https://www.marcocantu.com/delphibirth/>에서 매년 2월 14일마다 저자가 제품 출시일을 기념하기 위해 작성하는 블로그 게시물을 볼 수 있다.

역사 1995년은 프로그래밍 언어들에게 특별한 한 해였는데, 델파이 오브젝트 파스칼과, 자바와, 자바스크립트, 그리고 PHP가 등장했다. 이들은 현재까지도 여전히 쓰이는 가장 대중적인 프로그래밍 언어들이다. 사실, 다른 유명한 언어들(C, C++, Objective-C, 그리고 COBOL)은 더 오래 되었으며, 유일하게 새로운 대중적인 언어는 C#뿐이다. 프로그래밍 언어들의 역사는 http://en.wikipedia.org/wiki/History_of_programming_languages에서 볼 수 있다.

델파이 2에서, 볼랜드는 파스칼 컴파일러를 32비트 세계로 옮겨, C++ 컴파일러와 같은 코드 생성기(code generator)를 제공하기 위해 다시 제작하였다. 이것은 이전에 C/C++ 컴파일러에만 있었던 많은 최적화를 파스칼 언어에 제공했다.

델파이 3에서는, 볼랜드가 인터페이스의 개념을 언어에 추가하여, 클래스들과 그 관계에 대한 표현력의 진전을 만들었다.

델파이 7의 배포와 함께, 볼랜드는 공식적으로 오브젝트 파스칼 언어를 델파이 언어

라고 부르기 시작했지만, 당시에는 이전과 크게 달라진 것이 없었다. 당시 볼랜드는 Linux용 델파이 버전인 Kylix도 만들었고 나중에 델파이 8에 포함된 마이크로소프트 닷넷(.NET) 프레임워크용 델파이 컴파일러도 만들었다. 두 프로젝트 모두 나중에 폐기되었지만 2003년 말에 출시된 델파이 8은 언어에 대한 매우 광범위한 변경 사항, .NET 지원 필요성, 나중에 Win32용 델파이 컴파일러 및 기타 모든 후속 컴파일러에 채택된 변경 사항을 표시했다.

코드기어부터 엠바카데로까지의 오브젝트 파스칼

볼랜드가 개발 도구에 대한 투자에 대한 확신이 없었기 때문에 델파이 2007과 같은 이후 버전은 자회사인 코드기어에서 개발하였다. 이 자회사 (혹은 사업 부문)은 나중에 엠바카데로 테크놀로지에 매각되었다. 해당 배포판 이후 회사는 유니코드 지원(델파이 2009), 제네릭, 익명 메서드 또는 클로저, 확장된 런타임 유형 정보 또는 리플렉션 및 기타 여러 중요한 기능(대부분 이 책의 3부에서 다루었다)과 같이 개발자들이 오랫동안 기다려온 기능을 추가하여 오브젝트 파스칼 언어를 성장 및 확장하는 데 다시 집중했다.

동시에 Win32 컴파일러와 함께 회사는 델파이 XE2의 일부로 Win64 컴파일러와 macOS 컴파일러를 도입하여 이전 Linux용 델파이 버전인 볼랜드 및 Kylix가 실패했던 멀티 플랫폼 전략으로 돌아갔다. 그러나 이번에는 윈도우 개발 환경 하나에서 개발하여 다른 플랫폼으로 크로스 컴파일하는 방법을 사용했다. Mac 지원은 이후 iOS 및 Android와 같은 데스크톱 및 모바일 플랫폼을 수용하기로 한 멀티 디바이스 전략의 시작이었다. 이 전략은 FireMonkey라고 부르는 새로운 GUI 프레임워크의 도입으로 가능했다.

모바일 환경으로

모든 이전 버전의 델파이는 인텔 x86 CPU를 목표로 삼았지만, 모바일로의 전환으로 ARM 칩을 위한 최초의 오브젝트 파스칼 컴파일러가 도입되었다. 이러한 변화는 개방형 LLVM 컴파일러 아키텍처를 기반으로 하는 컴파일러 및 관련 도구인 “컴파일러 도구 체인”^{compiler toolchain}의 전반적인 재설계로 이어졌다.

참고 LLVM은 LLVM 컴파일러 하부 구조 혹은 “재사용 가능한 모듈식 컴파일러 및 도구 체인 기술 모음”을 줄인 이름이며, 관련된 내용은 다음 링크에 있다: <https://llvm.org/>

Delphi XE4와 함께 출시된 iOS용 ARM 컴파일러는 LLVM을 기반으로 한 최초의 오브젝트 파스칼 컴파일러였을 뿐만 아니라, 나중에 언어에서 제거되었지만 자동 참조 계산(ARC)과 같은 몇 가지 새로운 기능을 최초로 도입한 컴파일러였다.

이후 같은 해(2013)에 Delphi XE5에는 LLVM 기반의 두 번째 ARM 컴파일러와 함께 Android 플랫폼에 대한 지원을 추가했다. 요약하자면, Delphi XE5에는 오브젝트 파스칼 언어(Win32, Win64, macOS, Mac의 iOS 시뮬레이터, iOS ARM 및 안드로이드 [Android](#) ARM 지원)용 컴파일러 6개를 함께 제공한다. 이 모든 컴파일러는 오늘날의 델파이의 일부이며, 책 전반에 걸쳐 자세히 다루었던 몇 가지 중요한 차이점을 제외하고는 대체로 공통된 언어 정의를 지원한다.

2014년 초반에 Embarcadero는 동일한 핵심 모바일 기술을 기반으로 하는 AppMethod라는 새로운 개발 도구를 출시했다. 2014년 4월에는 Delphi의 XE6 버전도 출시했으며, 2014년 9월에는 AppMethod와 Delphi XE7의 세 번째 버전이 출시되었고, 이어서 2015년 봄에는 iOS를 대상으로 하는 최초의 ARM 64비트 컴파일러가 포함된 Delphi XE8이 출시되었다.

델파이 10.x 시기

델파이 10 Seattle 이후 Idera Corp.가 회사를 인수하면서 엠바카데로는 10.x 시리즈를 배포했다: 델파이 10.1 Berlin(베를린), 델파이 10.2 Tokyo(도쿄), 델파이 10.3 Rio(리오), 그리고 델파이 10.4 Sydney(시드니)다. 이 기간 동안 엠바카데로는 새로운 대상 플랫폼과 운영체제, 즉 Linux 64비트, 안드로이드 64비트 및 macOS 64비트에 대한 지원을 추가했다. 또한 회사는 윈도우 10 운영체제에 대한 특정한 지원을 추가하여 윈도우 VCL 라이브러리에 다시 집중했다.

10.x 시리즈 전반에 걸쳐 엠바카데로는 인라인 변수 선언, 사용자 지정 매니지드 레코드와 같은 기능을 도입하고 이 책에서 다루었던 기타 여러 가지 작은 개선 사항을 도입하여 오브젝트 파스칼 언어를 계속해서 발전시켰다.

델파이 11 배포

마이크로소프트가 윈도우 11을 출시하고, 애플 macOS 운영체제의 길었던 10.x 버전이 끝나면서, 엠바카데로는 10.x 시리즈에서 벗어나 11 알렉산드리아로 메이저 버전을 옮겨서 다시 번호를 이어 가기로 했다. 그래서 윈도우 11과 버전 번호와 맞췄다. 이 제품이 윈도우 운영체제와 긴밀하게 묶여 있음을 강조하기 위함이다. 멀티-디바이스 개발을 현대 델파이의 핵심 원칙으로 포용한 후에도 여전히 변함없다. 델파이 11에서 주목할만한 점은 macOS ARM-64비트 플랫폼에 대한 지원, 애플 M1 CPU용 네이티브 코드 생성, High-DPI를 지원하는 IDE 등이다.

b: 용어집

A

추상 클래스 (Abstract class)	완전하게 정의되지 않고 메서드의 인터페이스만 제공하는 클래스다. 그 서브-클래스는 해당 메서드를 구현해야 한다.
모호한 호출 (Ambiguous call)	컴파일러가 함수 호출을 해석하는 옵션이 두 개 이상인데, 자동으로 하나를 결정할 방법이 없는 경우, 여러분이 만나게 되는 오류 메시지다.
안드로이드 (Android)	구글의 모바일 장비용 운영체제의 이름이다. (구글 외에) 수백 개의 하드웨어 공급업체가 채택하고 있다. 오픈 OS이기 때문이다. 안드로이드는 현재 전 세계에서 가장 많이 쓰이는 운영체제다. 마이크로소프트 윈도우를 넘어섰다.
익명 메서드 (Anonymous method)	<p>익명 메서드, 또는 익명 함수는 함수 이름이 연결되지 않은 함수다. 변수에 할당되거나 다른 함수에 파라미터로 전달될 수 있다. 그래서 그 코드는 나중에 실행될 수 있다.</p> <p>익명 메서드는 일반 함수와 비교하면 조금 마법과도 같다. 정말 신기하게도, 익명 메서드가 결국 실행되는 곳은 다른 코드 블록의 안이지만, 자신이 선언된 블록 안에 있는 변수에 접근할 수 있다.</p> <p>익명 함수 그리고 그 함수가 접근할 수 있는 변수들을 하나의 <i>클로저</i>^{closure}라고 한다. 클로저는 익명 메서드의 또 다른 이름이다.</p>

API

API(Application Programming Interface, 애플리케이션 프로그래밍 인터페이스)는 운영체제 등 소프트웨어가 제공한다. 애플리케이션 프로그램들이 그 소프트웨어와 소통하고 동작할 수 있도록 하기 위해서다. 예를 들어, 애플리케이션이 화면에 텍스트 한 줄을 표시할 때는, 통상적으로 운영체제 안에 있는 함수를 호출하여 해당 GUI를 핸들링 하도록 한다. 이때 그 운영체제의 GUI와 인터페이스 할 수 있도록 제공되는 함수들의 모음을 그 운영체제 GUI의 API라고 한다. 일반적으로, 소프트웨어에서 제공하는 API는 자신이 작성된 그 언어로 되어 있다. 예를 들어, 마이크로소프트 윈도우는 C/C++로 작성되었고, C와 C++에 맞춰진 API를 제공한다.

참고: 오브젝트 파스칼의 WinAPI.Windows 유닛은 오브젝트 파스칼 API를 제공하여 마이크로소프트 윈도우에 접근할 수 있도록 한다. 따라서, C/C++용으로 작성된 함수를 직접 호출하는 번거로움이 없다.

B

불리언 표현식
(Boolean
expression)

George Boole이라는 유명한 수학자의 이름을 따다. 불리언 표현식은 True 또는 False로 평가되는 표현식이다. 간단한 예시로는 $1=2$ 이 있다. 이것은 False가 된다. 불리언 표현식은 전통적인 수학적 표현식일 필요는 없다. 그저 불리언인 변수이어도 되고, 불리언 값을 반환하는 함수에 대한 호출이어도 된다.

C

기수
(Cardinal)

기수는 일종의 자연수다. 간단히 말해, 사물을 세는 데 쓸 수 있는 숫자다. 그러니 항상 0보다 크거나 같다.

클래스
(Class)

클래스는 (그 클래스의) 오브젝트가 생성될 때 가지게 될 프로퍼티, 메서드, 데이터 필드에 대한 정의 [definition](#)이다.

참고: 모든 객체 지향 언어들이 클래스로만 객체를 정의하는 것은 아니다. JavaScript, IO, Rebol에서는 클래스를 먼저 정의하지 않고 오브젝트를 직접 정의할 수 있다.

참고: 오브젝트 파스칼에서, 레코드 정의는 클래스 정의와 매우 비슷하다. 레코드의 멤버는 클래스의 프로퍼티와 같은 기능을 한다. 또한 레코드의 프로시저와 함수는 클래스의 메서드와 같은 기능을 한다.

코드 포인트 (Code point)	유니코드 문자 집합의 요소가 저마다 가지고 있는 고유한 숫자 값. 세상의 문자 집합들 안에 있는 모든 문자, 숫자, 구두점은 저마다 자신을 가리키는 유니코드 코드 포인트가 있다.
컴파일러 지시어 (Compiler directive)	컴파일러 지시어는 특별한 명령이다. 컴파일러가 자신의 표준 동작을 바꾸도록 한다. 컴파일러 지시어를 넣으려면 \$ 기호로 시작하는 특별한 단어를 적어 넣는다. 또는 프로젝트 옵션(Project Options)에서 설정할 수도 있다. 참고: 컴파일러 지시어들 중에는 긴 이름을 줄여 한 글자로 쓰는 것들이 있다. 매우 오래된 것들이 그렇다.
컴포넌트 (Components)	컴포넌트들은 미리 만들어져 있고, 바로 사용할 수 있는 코드 오브젝트들이다. 이 코드 오브젝트들은 애플리케이션과 기타 다른 컴포넌트들과 손쉽게 합쳐질 수 있다. 그래서 애플리케이션 개발에 드는 시간을 크게 줄인다. VCL과 파이어몽키 라이브러리는 이런 컴포넌트들이 잔뜩 있는 방대한 모음이며, 델파이와 함께 제공된다.
컴포넌트 오브젝트 모델 (COM)	COM(Component Object Model) 즉 컴포넌트 오브젝트 모델은 마이크로소프트 윈도우 설계 구조의 핵심이다.
컨트롤 (Control)	컨트롤은 GUI 요소다. 버튼, 텍스트 입력 필드, 이미지 컨테이너 등등이 해당된다. 시각적 컴포넌트 visual component 를 가리킬 때 컨트롤이란 용어가 종종 사용된다.
CPU	중앙 처리 장치(Central Processing Unit, CPU)는 모든 컴퓨터의 핵심이며 코드를 실제로 실행한다. 오브젝트 파스칼 언어의 문장은 CPU가 이해하도록 어셈블리 assembly 코드로 번역되어야 한다. CPU 뷰어를 델파이의 디버거에서 쓸 수 있다는 점도 알아두자. CPU는 종종 FPU(같은 틀에 통합되어 있음)와 함께 작동한다.

D

데이터 타입 (Data type)	데이터 타입은 저장 요구 조건 그리고 그 타입의 변수에 대해 수행할 수 있는 동작들을 가리킨다. 오브젝트 파스칼에서는, 타입이 엄격한 다른 프로그래밍 언어와 마찬가지로, 변수는 저마다 특정 데이터 타입을 가진다.
-----------------------	--

디자인 패턴 (Design patterns)

다양한 개발자들이 다양한 문제를 해결하기 위해 사용하는 소프트웨어 아키텍처들을 살펴보면 유사점과 공통 요소들이 있음을 알게 된다. 디자인 패턴은 이러한 공통 설계^{design}에 대한 승인^{acknowledgment}이다. 표준 방식으로 표현되며, 충분히 추상화되어 있어서 상황에 적용할 수 있다.

소프트웨어 세상의 디자인 패턴 운동은 1994년 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides가 “Design Patterns, Elements of Reusable Object-Oriented Software” (Addison-Wesley, 1994, ISBN: 0-201-633612)라는 책을 쓰면서 시작되었다. 이 저자들은 종종 “Gang of Four(네 명의 갱)” 간단히 줄여 “GoF” 라고 불린다. 이 책 역시 “GoF 책”이라 부르기도 한다.

GoF 책에서 저자들은 소프트웨어 패턴의 개념을 설명하고 이를 서술하는 정확한 방법을 제시한다. 이 책은 23 가지 패턴들을 나열한다. 크게 3 그룹 즉 생성 패턴, 구조 패턴, 동작 패턴으로 구분되어 있다.

DLL

동적 연결 라이브러리(Dynamic Link Library, DLL)은 함수 라이브러리다. 이것은 애플리케이션의 실행 코드 안에 포함되지 않고 별도의 파일에 저장된다. 그리고 애플리케이션이 실행될 때, 그 라이브러리가 메모리 안에 적재되어 그 안의 함수들이 호출될 수 있게 된다. 이런 라이브러리들은 주로 많은 애플리케이션들이 사용할 수 있도록 설계된다. 윈도우 플랫폼 밖에서는, 이런 라이브러리를 공유 오브젝트(Shared Object, SO)라고 부른다.

E

이벤트 (Event)

이벤트는 애플리케이션에서 발생하는 액션 즉 동작이다. 예를 들어, 마우스 클릭, 폼 크기 조절 등이다. 델파이에서 이벤트 구현은 클래스의 특별한 프로퍼티를 사용한다. 그 방식을 통해 오브젝트는 일부 “행동”을 외부 메서드에게 위임할 수 있다. 이벤트는 RAD 개발 모델의 한 축이다.

F

파이어몽키 (FireMonkey)

파이어몽키(또는 FMX)는 델파이에서 제공하는 시각적 및 비시각적 컴포넌트 라이브러리다 (델파이의 UI 라이브러리로는 FMX뿐만 아니라 VCL 라이브러리도 함께 제공됨). 이 컴포넌트들은 크로스-플랫폼^{cross-platform}이다. 즉, 윈도우, 맥OS, iOS, 안드로이드, 심지어 리눅스(FMXLinux 애드-온 라이브러리가 필요함)에서 똑같이 동작한다.

폼 (Form)	VCL과 FireMonkey 라이브러리에서 사용하는 창>window을 가리키는 용어다.
파일 시스템 (File system)	파일 시스템은 컴퓨터 운영체제의 일부다. 컴퓨터에 데이터 저장을 조직화해 정돈하고, 데이터 저장과 추출을 관리한다.
FPU	부동 소수점 연산기(Floating Point Unit, FPU)는 CPU에 동반된다. 복잡한 부동 소수점 숫자 계산을 매우 빠르게 수행하는 데 특화되어 있다.
함수 (Function)	함수는 일종의 코드 블록이다. 몇 가지 동작(과 계산)을 수행하고 그 결과를 반환한다. 함수는 미리 지정된 개수의 파라미터를 받아서 동작할 수도 있다. procedure와 method도 용어집에서 참고하라.

G

전역 메모리 (Global memory)	전역 메모리는 일종의 정적 메모리static memory다. 여러분의 애플리케이션에 있는 전역 변수global variable들을 위한 곳이다. 이 메모리는 애플리케이션의 전체 수명 동안 사용된다. 그리고 그 공간은 늘어나지 않는다. 이와 달리 힙heap은 동적으로 할당되는 메모리 영역을 제공한다. 오브젝트 파스칼 애플리케이션은 전역 메모리를 아껴서 사용한다.
---------------------------	--

GUI	그래픽 사용자 인터페이스(Graphical User Interface, GUI)는 사용자가 그래픽 아이콘과 기타 시각적 표현들을 통해 컴퓨터, 태블릿, 폰과 상호작용할 수 있도록 한다. GUI를 사용하는 사용자 상호작용의 대부분은 찍기, 터치, 누르기, 쓸기 및 기타 제스처들을 이며 마우스 (또는 이와 비슷한 포인팅 장치) 또는 손가락을 사용한다.
-----	---

H

힙 메모리 (Heap memory)	힙은 일종의 메모리 공간이다. 동적으로 할당되는 메모리 블록들이 들어간다. 이름에서 알 수 있듯이, 힙 메모리 할당은 정해진 구조와 순서가 없다. 블록이 필요하면 그때마다 빈 영역에서 가져온다. 블록마다 그 수명이 다르며, 할당 순서와 해제 순서는 서로 관계가 없다. 힙 메모리는 오브젝트, 문자열, 동적 배열, 기타 참조 (용어집 'Reference' 참고) 타입들의 데이터가 사용한다. 또한, 수동으로 할당하는 블록 (용어집 'Pointer' 참고)들 역시 힙을 사용한다. 힙은 매우 크다. 하지만 무한하지 않다. 여러분이 만약 사용하지 않는 오브젝트를 메모리에서 해제하지 않는 경우에는, 그 애플리케이션은 결국 메모리 부족에 처하게 될 것이다.
------------------------	---

I

통합 개발 환경
(IDE)

IDE (Integrated Development Environment) 즉 통합 개발 환경은 단일 애플리케이션이다. 개발자에게 다양한 도구를 제공하여 생산성을 크게 높여준다. IDE는 최소한 소스 코드 편집기, 빌드 자동화 도구, 디버거를 제공해야 한다. IDE에 대한 현대적인 아이디어의 발명은 (볼랜드가 내놓은)터보 파스칼 컴파일러들과 함께 시작됐다. 이는 지금 엠바카데로 테크놀로지가 만들고 있는 오브젝트 파스칼 IDE의 전신이다.

델파이가 제공하는 오브젝트 파스칼 IDE는 매우 복잡적이다. 예를 들어, GUI 디자이너, 코드 템플릿, 리팩토링, 내장된 유닛 테스트 등이 들어 있다.

타입 상속
(Type inheritance)

타입 상속은 객체 지향 프로그래밍(OOP)의 핵심 원칙 중 하나다. 한 데이터 타입이 기존 데이터 타입을 확장해 새 기능을 추가할 수 있다는 아이디어다. 이런 타입 확장을 타입 상속이라고 한다. 기반 클래스와 자손 클래스, 부모 클래스와 자식 클래스 등의 용어들이 함께 사용된다.

인터페이스
(Interface)

대체로, 소프트웨어 모듈이 수행하는 일에 대한 추상 선언을 가리키는 말이다. C#이나 Java처럼, 오브젝트 파스칼에서는 인터페이스를 순수한 추상 클래스(데이터가 없으며 메서드로만 구성됨)로 정의한다 (전체 설명은 11장을 참고).

또한, 이 언어에는 `unit`에도 인터페이스 개념이 있다. 유닛 안의 인터페이스 구역에 선언된 내용들은 다른 유닛에게 보여진다. 두 경우 모두, 사용되는 키워드는 `interface`다.

iOS

애플의 모바일 장치를 구동하는 운영체제의 이름이다.

M

macOS

애플 맥 컴퓨터의 운영체제 이름이다. 예전에는 OS X로 알려졌다.

메서드
(Method)

메서드는 함수 또는 프로시저다. 그런데, 오브젝트에 묶여 있다. 메서드들은 자신이 속한 오브젝트 안에 저장된 모든 데이터에 접근할 수 있다.

O

오브젝트
(Object)

오브젝트는 데이터 항목(프로퍼티와 필드)들과 코드(메서드)들의 조합이다. 오브젝트는 클래스의 인스턴스다. 클래스는 오브젝트가 소속되는 가족(즉 타입)을 정의한다.

OOP	객체 지향 프로그래밍(object-oriented programming, OOP)은 오브젝트 파스칼을 받쳐주는 모델이다. 클래스, 상속, 다형성과 같은 개념들을 기반으로 한다. 현대 오브젝트 파스칼은 다른 프로그래밍 패러다임들도 지원한다. 제네릭, 익명 메서드, 리플렉션 같은 기능들 덕분이다.
순서 타입 (Ordinal type)	순서 타입은 셀 수 있으며 순서가 정해진 요소들을 담는 데이터 타입이다. 정수를 생각하면 된다. 그런데, 문자에도 순서가 있다. 사용자 지정 열거 타입도 마찬가지다.
오버로딩 (Overloading)	함수 오버로딩은 변수 타입에 엄격한 프로그래밍 언어의 특징 중 하나다. 프로그래머는 함수 하나에 대해 다양한 버전 여러 개를 선언할 수 있다. 함수의 다양한 버전들은 서로 다른 파라미터 타입들을 받는다.
P	
포인터 (Pointer)	포인터는 변수의 일종이다. 그 변수 안에는 메모리 주소를 담는다. 포인터는 데이터 또는 함수의 메모리 내 위치를 가리킨다. 포인터는 널리 사용되지 않는다. 반면, 참조(아래의 'Reference' 참고)는 매우 일반적이고 훨씬 더 쉽게 사용할 수 있는 불투명opaque하고 관리되는managed 포인터다.
다형성 (Polymorphism)	다형성이란 메서드를 호출할 때 그 메서드가 “다른 형태”일 수도 있다는 가정을 할 수 있도록 하는 기능이다. 즉, 메서드의 동작은 달라질 수 있으며, 그 동작은 그 메서드가 적용되는 오브젝트가 무엇인가에 의해 결정된다. 이것은 OOP 언어의 표준 특성이다.
프로시저 (Procedure)	프로시저는 일종의 코드 블록이다. 프로그램의 다른 부분에서 호출할 수 있다. 프로시저는 파라미터를 받을 수 있어서 그에 따라 하는 일이 달라질 수 있다. 함수와 달리, 프로시저는 값을 반환하지 않는다. 파스칼과 달리, 다른 대부분의 언어들은 함수와 프로시저를 구분하지 않고 둘 다 '함수'라고 부른다.
프로젝트 옵션 (Project options)	구성configuration 옵션들의 묶음이다. 그 옵션들은 애플리케이션 프로젝트의 전체 구조에 영향을 줄 뿐만 아니라 컴파일러와 링커linker의 작동 방식에도 영향을 미친다.
프로퍼티 (Property)	프로퍼티는 오브젝트의 상태를 정의한다. 프로퍼티가 데이터에 매핑되거나 또는 메서드를 사용하여 데이터의 값을 읽고 쓸 수 있다는 사실을 고려하면, 실제 구현을 감추고 추상화를 하는 용도로 사용될 수 있다.

R

RAD	RAD (Rapid Application Development)는 쉽고 빠르게 애플리케이션을 구축할 수 있도록 하는 개발 환경의 특징이다. RAD 도구는 일반적으로 시각적 디자이너 ^{visual designer} 를 기반으로 한다. 이런 식의 정의는 다소 오래된 정의라서 오늘날 거의 이렇게 정의하지 않는다.
레코드 (Record)	단순 레코드란 데이터 항목들의 모음이다. 각 항목은 구조화된 방식으로 저장된다. 레코드는 타입 정의로 정의된다. 그 정의에는 레코드가 가지는 개별 데이터 항목의 순서와 타입을 명시한다. 오브젝트 파스칼에는 고급 레코드도 있다. 이것은 (오브젝트와 비슷하게) 메서드를 가질 수 있다.
재귀 (Recursion)	재귀 즉 재귀 호출은 함수가 주어진 조건이 충족될 때까지 자신을 계속 호출하는 것을 설명하는 말이다. 재귀 호출은 루프 또는 순환에 대한 보다 우아한 대안인 경우가 종종 있다. 요즘에는 대체로 재귀를 덜 사용한다. 곱셈 함수를 재귀적으로 구현하는 예를 들면, 함수는 전달된 첫 번째 숫자의 값을 취한다. 그리고 두 번째 숫자를 1 만큼 감소시키고 동일한 함수를 호출하고 그 결과 값을 앞에서 취한 값에 더한다. 이것을 두 번째 숫자가 0이 될 때까지 계속한다.
참조 (Reference)	참조는 변수의 일종이다. 그 변수 안에는 메모리 안의 다른 위치에 있는 데이터를 가리키는 값이 담긴다. 즉 데이터를 직접 저장하지 않는다. 오브젝트 파스칼에서, 참조는 타입 (예: 클래스, 문자열 등)을 가리키는 변수다. 뿐만 아니라, 인터페이스와 동적 배열 역시 참조다. 포인터(용어집 'Pointer' 참고)와 달리, 일반적으로 참조 관리는 컴파일러와 런타임 라이브러리가 한다. 그러므로 저수준 ^{low-level} 에 대한 지식이나 개발자가 그 메모리를 직접 관리할 필요가 거의 없다.
RTTI 혹은 리플렉션 (reflection)	RTTI는 런타임 타입 정보(Run Time Type Information)의 약어다. 애플리케이션 안의 타입 정보에 대한 접근(전통적으로는 컴파일러에서만 가능했음)을 실제 실행 중에 할 수 있도록 하는 기능이다. 다른 프로그래밍 환경들은 이 기능을 '리플렉션'이라고 부른다.

런타임 라이브러리
(Run-Time Library,
RTL)

미리 작성된 루틴들의 모음이다. 컴파일러는 그 루틴들을 자동으로 애플리케이션 코드 안에 붙여 넣는다. 그래서 그 애플리케이션이 실행되도록 지원한다. RTL에는 많은 기본 작업을 위한 지원들이 들어 있다. 특히 주요 지원 작업들은 운영체제와 상호 작용(예: 메모리 할당하기, 데이터 읽기 및 쓰기, 파일 시스템과 상호 작용하기)이 있다.

S

SDK

소프트웨어 개발 키트(Software Development Kit, SDK)는 소프트웨어 도구들이다. 소프트웨어를 특정 환경에 맞추어 구축할 때 사용된다. 각 운영체제는 저마다 SDK를 제공하며 애플리케이션 프로그래밍 인터페이스(API) 라이브러리들도 그 안에 들어 있다. 또한 개발자 도구들도 제공하여 해당 플랫폼용 애플리케이션을 구축, 테스트, 디버깅할 수 있게 한다.

탐색 경로
(Search path)

폴더들의 집합이다. 컴파일러는 `uses` 문 안에 명시되어 참조되는 외부 유닛을 찾을 때 여기를 검색한다.

스택 메모리
(Stack memory)

스택은 동적으로 그리고 순서대로 할당되는 메모리 공간이다. 메서드, 프로시저, 함수가 호출될 때마다 스택은 자체 메모리 영역을 확보한다(로컬 변수 즉 임시 변수와 파라미터 등을 위해 필요함). 그리고, 그 메서드가 반환되면, 해당 메모리는 정확히 차례대로 비워진다. 스택 메모리가 완전히 소진되는 유일한 실제 상황은 메서드가 무한^{infinite} 재귀 호출에 빠지는 경우다(용어집의 'Recursion' 참고).

참고: 대체로, 로컬 변수는 스택에 할당되며, 0으로 초기화되지 않는다: 여러분은 로컬 변수의 값을 사용하기 전에 미리 초기화하는 것이 좋다.

U

유니코드
(Unicode)

유니코드는 텍스트 안의 개별 문자를 이진 데이터(0과 1의 수열)로 기록하는 표준 방식 중 하나다. 유니코드 표준을 준수하면, 텍스트를 프로그램 간에 안정적으로 교환하고 처리 및 표시할 수 있다. 이 표준은 11만개 이상의 다양한 문자를 포괄할 정도로 매우 방대하며, 약 100 가지 다양한 문자 집합들과 스크립트들을 담고 있다.

V

VCL Visual Component Library (비주얼 컴포넌트 라이브러리, VCL)는 델파이와 함께 제공되는 방대한 시각적 컴포넌트들이다. VCL의 GUI 컴포넌트들은 네이티브 윈도우 GUI 컴포넌트다. 크로스-플랫폼 라이브러리에 대해서는 용어집 'FireMonkey'를 참고하라.

가상 메서드 (Virtual methods) 가상 메서드는 일종의 함수 또는 프로시저인데, 그 선언은 클래스의 타입 정의 안에 있다. 그 클래스의 하위 클래스 [sub-class](#)들은 가상 메서드를 재정의 [override](#) 할 수 있다. 이른바 '기반 클래스'에서는 가상 메서드들에 대한 구현 [implementation](#)이 없어도 되며, 그 구현은 해당 하위 클래스에서 정의될 수 있다. 이처럼 정의가 생략된 가상 메서드는 추상 메서드 [abstract method](#)라고 부른다.

W

창 (window) 창은 GUI 요소를 담는 화면 영역이다. 사용자가 상호작용을 하는 영역이기도 하다. GUI 애플리케이션은 여러 개의 창을 표시할 수 있다. VCL과 파이어몽키에서는, 창을 정의할 때 `Form`(폼) 오브젝트를 사용한다.

윈도우 (Windows) 마이크로소프트의 유비쿼터스 운영체제 이름이다. 이것은 당시 다른 운영체제들(예: 애플 매킨토시의 System X)과 함께 그래픽 창이라는 개념(용어집 'window' 참고)을 개척했다.