오브젝트 파스칼, C++ 그리고 자바의 특징
(Characterisitcs of Object Pascal, C++ and Java)

자바는 가장 일반적인 인터넷용 언어이며, C++은 아마도 가장 흔하게 사용되는 OOP 언어일 것이다. 이들과 델파이에서 사용되는 오브젝트 파스칼의 언어적인 측면에서의 비교를 해보면서 OOP 언어에 대한 감을 조금 더 높여 보자.

OOP 언어의 특징

객체지향 프로그래밍(OOP)라는 것은 일종의 프로그래밍 테크닉으로, 가장 최초로 구현된 것은 스몰토크에서 였다. 그러다가, 80년대에 들어서면서 C++, Objective-C, 터보 파스칼, 에펠, Ada, 자바 등에 적용되면서 프로그래밍 언어에서 가장 중요한 기법으로 각광받게 되었다.
OOP 언어의 특징은 앞 장에서도 언급한 바 있지만, 다시 한번 이를 간단하게 정리해 보도록 하겠다.

1. 클래스라는 추상적인 데이터 형의 개념을 도입한다. 이를 이용해서 캡슐화, 모듈화, 데이터 추상화 등을 구현하게 된다.
2. 이미 존재하는 요소에서 그 기능을 상속할 수 있다. 이를 이용해서 몇가지 기능을 수정하거나, 추가만 하면 새로운 형태의 데이터 형을 정의할 수 있게 된다. 이를 통해 데이터의 세분화와 일반화가 가능해진다.
3. 마지막으로 다형성이라는 특징을 들 수 있다. 이 특징을 이용하면 같은 방법으로 서로 다른 클래스의 객체를 참조할 수 있다. 다형성을 이용하면 클래스의 재사용성이 보다 높아지고, 프로그램을 보다 쉽게 확장, 유지할 수 있게 된다.

위의 클래스, 상속성, 다형성의 3 가지 특징은 OOP 언어라고 불려지는 언어라면 모두 지원해야 한다. 참고로, 위의 3 가지 특징 중 상속성과 다형성을 지원하지 않는 언어를 '객체기반(class-based)' 언어라고 한다.
OOP 언어들은 위의 3 가지 특징을 모두 지원하지만, 데이터 형을 검사하는 방법과 프로그래밍 모델, 객체 모델 등에서 서로 차이가 있게 된다. 여기에 대해서 좀더 심층적으로 알아보기로 하자.

컴파일 vs 런타임 형 검사 (Compile-Time vs. Runtime Type Checking)

프로그래밍 언어를 구별할 때, 얼마나 데이터 형을 검사하는 방법이 까다로운가에 대한 것을 많이 고려하곤 한다. 그러니까 호출된 메소드의 존재 여부, 파라미터의 데이터 형, 배열

의 범위 측정 등을 얼마나 정확하고 까다롭게 하는지 등의 여부가 각 언어들끼리 조금씩 차이가 있다.
C++, 자바, 오브젝트 파스칼 모두 컴파일 시에 주로 데이터 형을 검사한다. 그 중에서도 자바가 가장 엄격하게 컴팔일 시에 데이터 형을 검사하며, C++은 가장 덜하다. C++은 C 언어와의 호환성을 위해서 데이터 형 간의 구별이 덜 엄격한 편이다. C 를 많이 써본 독자들은 아마도 float 형으로 선언한 변수의 값과 int 로 선언한 변수의 값을 혼용하는 것을 많이 경험했을 것이다. 그에 비해 오브젝트 파스칼이나 자바에서는 정수형은 어디까지나 정수형이며, 실수형은 어디까지나 실수형이다. 이들 간의 호환성을 위해서는 엄격한 형변환이 필요하다.
자바 가상기계(VM)는 바이트 코드를 런타임에서 해석할 수 있기 때문에, 자바 언어가 컴파일 시에 형 검사를 하지 않는다고 생각하기 쉽지만, 그와 반대로 훨씬 더 엄격하게 형 검사를 한다는 것은 잘못 알기 쉬운 사실이다.

하이브리드 vs. 순수 OOP 언어

순수한 OOP 언어는 OOP 라는 하나의 프로그래밍 모델 만을 지원하는 언어이다. 즉, 개발자는 클래스와 메소드를 선언할 수 있지만, 과거에 사용해 왔던 일반적인 함수나 프로시저, 전역 변수 등은 사용할 수 없다.
자바는 OOP 프로그래밍 모델 만을 지원하는 순수 OOP 언어이다. 자바 외에 에벨(Eiffel)과 스몰토크 등을 순수 OOP 언어라고 말할 수 있다. 일반적으로 이러한 순수 OOP 언어는 프로그래머로 하여금 반드시 OOP 모델을 사용하게 만들기 때문에 OOP 를 처음 시작하는 사람에게 OOP 를 익히게 하는데 유리하다. 그에 비해 C++과 오브젝트 파스칼은 OOP 프로그래밍 모델 이외에 전통적인 C 와 파스칼 프로그래밍 모델을 같이 지원하는 하이브리드 언어이다.
하이브리드 언어는 호환성을 유지해야 하기 때문에, 원시적인 데이터 형들이 클래스 및 객체 들과 혼용되는 형태를 가질 수 밖에 없다. 또한, 기존의 컴파일러 언어의 특성상 정적으로 프로시저나 함수를 사용할 수 있어야 하며, 사실상 이것이 주로 사용되는 방법이기 때문에 순수 객체지향 언어와 같이 동적으로 다형성을 지원하는 언어에 비해 다소 복잡한 함수 호출과 구현 방식을 가질 수 밖에 없다. 이렇게 서로 다른 프로그래밍 모델을 지원해야 하는 하이브리드 언어는 다소 몸집이 크고, 복잡하지만 유연성이라는 측면에서는 장점을 가지고 있다.

단순 객체 모델 vs. 객체 참조 모델 (Plain Object Model vs. Object Reference Model)

전통적인 OOP 언어는 프로그래머가 객체를 스택과 힙, 정적인 저장소에 생성할 수 있도록

허용하고 있다. 이러한 언어에서는 클래스 데이터 형의 변수가 메모리에 있는 객체와 직접적으로 연관된다. C++이 이러한 단순 객체 모델을 따르는 예가 된다.
그에 비해, 오브젝트 파스칼과 자바는 각각의 객체는 힙에 동적으로 메모리를 할당받게 되며, 클래스 데이터 형의 변수는 실제로 메모리 상의 객체가 아니라 객체에 대한 메모리의 핸들을 담고 있게 된다 (포인터와 비슷한 개념이다).

클래스와 객체, 그리고 참조 (Classes, Objects and References)

클래스란 일종의 데이터 형이며, 객체는 클래스가 실제로 메모리에서 인스턴스화 된 것이다. 오브젝트 파스칼과 C++, 자바는 모두 클래스에 바탕을 둔 언어이다. 각각의 객체들은 직접 정의되지 않고, 일단 클래스로 정의된 후 객체는 클래스에 대한 하나의 인스턴스로 나타나는 형태를 가지고 있다. 클래스는 객체와 1:n 의 관계를 가지게 된다. 클래스는 코드를 메소드의 형태로 저장하며, 여러 객체 들과 이를 공유하게 된다. 어쨌든 각 객체는 자신의 데이터 필드를 저장하게 되며, 이것으로 클래스와 객체가 구별된다.
객체 내의 필드는 원시적인 데이터 형이거나 객체일 수 있다. 정상적으로 각 객체는 클래스에서 선언된 데이터 필드의 복사본을 자신의 것으로 가지고 있다.
이들의 관계가 각각의 언어에서 어떻게 특징지워 지는지 살펴보도록 하자.

- C++

C++에서 MyMethod 라는 메소드를 가진 MyClass 란 클래스가 있다고 하면, 다음과 같이 객체를 사용할 수 있다.

```
MyClass Obj;
Obj.MyMethod();
```

첫째 줄의 선언으로 Obj 라는 MyClass 형의 객체를 얻게 된다. 이때 이 객체에 대한 메모리는 전형적으로 스택에서 할당받게 되므로, 바로 그 다음 줄에서 이 객체를 사용할 수 있게 된다.

- 자바

자바에서는 객체에 해당하는 핸들에 대한 메모리를 할당받기 때문에, 다음과 같이 사용해야 한다.

```
MyClass Obj;
```

```
Obj = new MyClass();
Obj.MyMethod;
```

이처럼 객체를 사용하기 전에 'new' 키워드를 이용해서 객체에 대한 메모리를 할당받아야 한다. 물론, 위의 문장을 아래와 같이 2 줄로 줄여 쓸 수 있다.

```
MyClass Obj = new MyClass();
Obj.MyMethod();
```

- 오브젝트 파스칼

오브젝트 파스칼도 근본적으로는 자바와 비슷하지만, 문법의 구조가 다음과 같이 다소 차이가 날 뿐이다.

```
var
  Obj: MyClass;
begin
  Obj := MyClass.Create;
  Obj.MyMethod;
```

위의 비교에서, 객체참조 모델과 단순객체 모델의 차이점을 이해할 수 있을 것이다. 프로그래머의 입장에서 보면 객체참조 모델이 다소 복잡해 보인다. 그렇지만, 단순객체 모델에서는 각 변수가 직접 객체를 가리키기 때문에, 객체를 지정하거나 참조하기 위해서 포인터를 사용해야 하는 빈도가 늘어날 수 밖에 없다. 그에 비해 객체참조 모델을 사용하는 언어에서는 객체를 참조할 때 포인터를 쓰지 않아도 되며, 각각의 객체를 직접 제어할 필요도 없다. 이런 이유로 자바에서는 포인터를 지원하지 않는다.

메모리 모델

오브젝트 파스칼과 C++은 모두 기존의 record(오브젝트 파스칼), struct(C++) 데이터 형에다가 클래스의 VMT 에 대한 포인터를 추가한 것을 클래스로 메모리에서 관리하고 있다. C++의 경우 프로그램 변수를 할당하는데 static, automatic, dynamic 의 3 가지 저장 클래스를 지원한다. Static 객체는 프로그램의 데이터 세그먼트에 생성되며, automatic 객체는 스택에, dynamic 객체는 힙에 생성된다. Static storage 는 전역 객체에 사용되며, automatic storage 는 로컬 객체에, dynamic storage 는 런타임에서 생성된 객체에 사용된다.

C++은 객체를 생성하는 프로세스를 메모리의 할당과 초기화하는 크게 두 과정으로 나눈다. 메모리 할당은 new 메소드를 이용해서 이루어진다. C++의 경우에는 new 메소드를 오버라이드 하는 경우도 많은데, 이를 통해 사용자가 메모리를 할당하는 저수준 프로그래밍 루틴을 직접 작성할 수 있다.
오브젝트 파스칼은 dynamic 객체 만을 지원한다. 로컬, 전역 객체가 선언될 수 있지만, 이들은 단지 레퍼런스로서 데이터 세그먼트나 스택에 저장된다. 객체 자체는 반드시 직접 생성해서 항상 힙에 저장해야 한다. 오브젝트 파스칼은 기본적으로 사용자 정의 메모리 할당 루틴을 제공하지는 않는다.
그렇지만, 오브젝트 파스칼과 C++은 모두 자동 garbage collection 은 지원하지 않으므로, 프로그래머가 각 객체를 동적으로 생성한 경우 이를 제거할 필요가 있다.
자바의 경우는 dynamic 객체와 메모리 관리를 동적으로 해준다. 또한, garbage collection 을 해주므로 동적으로 생성한 객체에 대해 언어 차원에서 관리를 해준다.

생성자 (Constructors)

그러면, 각 언어의 생성자에 대해서 알아보자. 이를 이용해서 객체를 초기화하게 된다.

- C++

C++은 클래스와 같은 이름의 생성자를 가진다. 만약에 프로그래머가 클래스에 생성자를 정의하지 않으면, 컴파일러가 디폴트 생성자를 클래스에 추가한다. 메소드 오버로딩을 이용해서 여러 개의 생성자를 가질 수 있다.

- 자바

C++과 비슷하게 사용하지만, ‘initializer’라고도 불린다. 이것의 의미는 객체를 실제로 생성하는 것은 자바의 VM 이 하는 일이고, 생성자에 써 넣은 코드는 단순히 새롭게 생성된 객체를 초기화하는 역할을 한다는 의미이다.

- 오브젝트 파스칼

오브젝트 파스칼에서는 constructor 라는 키워드를 사용해서 생성자를 정의한다. 메소드 오버로딩을 지원하지 않지만, 생성자가 여러 가지 이름을 가질 수 있기 때문에 여러 개의 생성자를 가질 수 있다. 오브젝트 파스칼에서는 각각의 클래스가 디폴트 Create 생성자를 가지고 있게 되는데, 이 생성자는 기본적인 기초 클래스에서 상속받게 된다.

파괴자 (Destructor)

파괴자는 생성자와 반대되는 일을 한다. 즉, 객체가 메모리에서 해제될 때 호출되며 생성자에 의해 할당된 리소스를 해제하는 역할을 하게 된다.

- C++

C++의 파괴자는 객체가 범위를 벗어나거나, 동적으로 할당된 객체를 삭제할 때 자동으로 호출된다. 모든 클래스는 오직 하나의 파괴자를 가지게 된다.

- 자바

자바는 파괴자를 가지지 않는다. 참조되지 않는 객체는 백그라운드에서 실행되고 있는 가비지 컬렉션(garbage collection) 알고리즘을 통해 자동으로 파괴되며, 객체가 파괴되기 전에 finalize() 메소드를 호출한다.

- 오브젝트 파스칼

오브젝트 파스칼의 파괴자는 C++과 비슷한 역할을 한다. 오브젝트 파스칼에서는 표준 가상 파괴자인 Destroy 를 이용하는데, 이 파괴자는 Free 메소드에 의해 호출된다. 모든 객체가 동적으로 처리되기 때문에, 일단 객체가 생성되면 그 객체의 소유주(owner) 객체가 파괴될 때 자동으로 Free 메소드가 호출된다. 이론적으로 여러 개의 파괴자를 선언할 수 있다.

클래스 캡슐화

클래스 캡슐화의 레벨을 지정하는 지시어로 private, protected, public 이 사용된다. 그러나, 약간의 차이가 있으니 이를 살펴 보자.

- C++

C++에서 friend 키워드를 사용하면 캡슐화의 범위를 벗어날 수 있다. 기본적으로 클래스의 캡슐화 정도는 private 에 준하며, 구조체의 경우에는 public 에 준한다.

- 자바

자바에서는 디폴트로 각 요소들이 friend 로 간주된다. 즉, 각 요소 들은 같은 패키지 안에 있거나 같은 소스 코드 파일 내에 있으면 다른 클래스에서도 접근할 수 있다. protected 키워드로 지정한 경우에 서브 클래스와 같은 패키지 내의 다른 클래스에서 접근할 수 있다. 즉, private proteced 를 복합적으로 사용할 때 C++의 protected 와 같은 의미가 된다.

- 오브젝트 파스칼

자바와 비슷하다. private, protected 키워드는 다른 유닛일 경우에는 접근할 수 없지만, 같은 유닛(같은 소스 코드 파일)에 있으면 접근이 가능하다. 델파이의 경우에는 published 키워드를 제공하는데, 이 키워드를 이용하면 그 요소에 대한 RTTI 정보가 생성된다.
그 밖에, 오브젝트 파스칼은 디자인 시에 필드의 값을 보고, 편집할 수 있는 프로퍼티를 제공한다. 프로퍼티는 간접적으로 객체의 필드에 접근하기 때문에, 캡슐화 개념을 충실하게 지키고 있다. 프로퍼티는 데이터 필드에 대한 단순한 레퍼런스일 수도 있고, 참조 무결성을 유지하는 등의 보다 여러가지 조작을 가할 수 있는 함수를 호출하는 것일 수도 있다.

파일, 유닛 그리고 패키지

파일에 소스 코드가 조직되어 있는 방법에 많은 차이점이 있다. C++ 컴파일러는 파일에 대한 정보를 무시하는데 비해, 자바와 오브젝트 파스칼의 경우에는 파일 단위가 일종의 모듈 단위로 인식된다.

- C++

C++ 프로그래머는 클래스의 정의 부분을 헤더 파일에 저장하고, 실제 메소드의 정의는 분리된 코드 파일에 저장하는 경향이 있다. 이런 파일 들은 대부분 같은 이름을 가지고, 다른 확장자를 가지게 된다. 이와 같이 C++에서는 파일 단위에 대한 특별한 제한점이 존재하지 않는다. 이는 다르게 말하면, 컴파일러가 서로 다른 파일에 존재하는 여러가지 선언들에 대해서 별로 신경을 쓰지 않고 컴파일을 하기 때문에, 링커가 많은 일을 해야 한다는 것을 의미하며 이것이 C++의 링킹 속도가 저하되는 원인이라고 할 수 있다.

- 자바

자바에서는 각각의 소스 코드 파일이나 컴파일 단위가 서로 분리되어 있다. 그렇기 때문에, 컴파일 유닛을 하나의 그룹으로 묶을 수가 있는데, 이를 패키지라고 한다. 클래스를 선언하면 클래스의 메소드에 대한 코드도 같이 써주어야 한다. 'import' 키워드를 이용해서 다른 파일의 내용을 읽을 수 있는데, 이 때에는 그 파일에서 public 으로 선언된 부분 만을 읽

어올 수 있다.

- 오브젝트 파스칼

오브젝트 파스칼의 경우에는 각각의 소스코드 파일을 유닛이라고 한다. 이러한 유닛은 크게 나누어 interface, implementation 이라는 2 개의 파트로 이루어져 있다. Interface 섹션은 클래스의 정의와 메소드에 대한 선언 부분을 포함하며, implementation 섹션에는 interface 섹션에서 선언한 메소드의 구현 부분이 위치하게 된다. 다른 파일에서 선언된 interface 부분을 'uses' 키워드를 통해 접근할 수 있다.

또다른 특징을 언급하자면, 자바나 오브젝트 파스칼의 컴파일러는 컴파일이 된 파일(델파이의 경우 .dcu 파일)에서 선언부분의 내용을 읽어올 수 있는데 비해, C++에서는 이러한 모듈 구조를 허용하지 않는다.

클래스 메소드와 데이터

OOP 언어는 일반적으로 특정 객체에만 해당하지 않고, 클래스에 전반적으로 적용시킬 수 있는 메소드와 데이터를 허용한다. 클래스 메소드는 클래스의 객체와 클래스 자체에서 호출할 수 있으며, 클래스 데이터는 각각의 객체에 의해 복제되지 않고, 공통으로 사용되는 데이터를 말한다. 이들을 다른 말로 정적 메소드(static method), 정적 데이터(static data)라고도 한다.

- C++

C++의 클래스 메소드와 데이터는 'static' 키워드에 의해 지정된다. 클래스 데이터는 특정한 선언문에 의해서 초기화 되어야 한다.

- 자바

자바는 C++과 마찬가지로 'static' 키워드를 이용해서 클래스 메소드와 데이터를 지정할 수 있다. 클래스 메소드는 매우 자주 사용되는데, 이는 자바가 전역 함수를 허용하지 않기 때문이다. 클래스 데이터는 클래스의 선언부에서 직접 초기화될 수 있다.

- 오브젝트 파스칼

오브젝트 파스칼은 클래스 메소드만 지원한다. 클래스 메소드는 'class' 키워드에 의해 지

정될 수 있다. 클래스 데이터는 직접 지원하지 않고, 대신 클래스를 정의한 유닛에 private 전역 변수를 추가해서 이 기능을 대치한다.

전체 클래스의 조상

일부의 OOP 언어에서는 최소한 하나의 기초 클래스를 가지는 경우가 있다. 이러한 클래스는 모든 클래스에서 동일하게 가져야 하는 기본적인 특징 들을 가지고 있다. 즉, 다르게 말하면 모든 클래스는 가장 기본적인 기초 클래스를 상속받는 것이다. 이러한 개념은 스몰토크에서부터 시작된 것으로 비교적 많은 OOP 언어에서 채택되고 있는 방식이다.

- C++

C++은 기본적으로 이러한 개념을 지원하지 않는다. 그렇지만 C++에 기초한 많은 어플리케이션 프레임워크(MFC. OWL 등)에서는 이러한 기초 클래스 개념을 받아들이고 있다. 예를 들어, MFC 의 경우 CObject 클래스가 기초 클래스로 사용된다.

- 자바

자바의 모든 클래스는 Object 클래스에서 상속받는다.

- 오브젝트 파스칼

오브젝트 파스칼은 TObject 클래스를 공통의 조상으로 가진다. 오브젝트 파스칼은 기본적으로 다중 상속을 지원하지 않기 때문에, 상당히 커다란 상속 트리를 가지게 된다. TObject 클래스는 RTTI를 사용할 수 있으며, 그 밖에 몇 가지 기본적인 특징을 가진다.

하위형 호환성 (Subtype Compatibility)

모든 OOP 언어가 형 검사를 엄격하게 하는 것은 아니지만, 우리가 지금 논의하고 있는 세 가지 언어는 비교적 형 검사가 엄격한 편이다. 이는 기본적으로 서로 다른 클래스의 객체들간의 형-호환성(type-compatibility)이 보장되지 않는다는 것이다. 이러한 규칙의 예외가 있는데, 특정 클래스를 상속한 클래스의 객체는 부모 클래스와 데이터 형의 호환성이 인정된다 (역은 성립하지 않는다.). 이러한 하위형 호환성은 다형성(polymorphism)과 late binding을 지원하는데 중요한 역할을 한다.

- C++

C++의 경우에는 이러한 하위형 호환성이 포인터와 참조(reference)에 대해서만 허용된다. 서로 다른 객체는 서로 다른 크기의 메모리를 사용하기 때문에, 이들 객체에 대한 직접적인 하위형 호환성을 보장할 수가 없다.

- 자바, 오브젝트 파스칼

이들은 모든 객체에 대해서 하위형 호환성을 지원한다. 이것이 가능한 이유는 앞에서도 설명했듯이 이들이 모두 객체참조 모델을 사용하기 때문이다. 오브젝트 파스칼의 예를 들면, 모든 객체는 TObject 클래스와 호환된다.

다형성(Polymorphism)과 late 바인딩

오브젝트 파스칼과 C++은 메소드를 디스패치할 때 정적, 동적 바인딩을 모두 지원한다. 정적 바인딩은 다형성을 지원하지 않으며, 컴파일 시에 동작하며 전통적인 함수 호출에서 사용되는 방식이다. 정적 메소드의 주소는 링커에 의해 코드 세그먼트에 직접 저장된다. 이에 비해 동적 바인딩 또는 가상 메소드는 다형성을 지원한다. 이를 위해 객체의 정확한 데이터 형을 모르는 런타임에서 동작하게 된다.
오브젝트 파스칼과 C++은 모두 가상 메모리 테이블(VMT)를 통해 가상 메소드의 주소를 저장한다. 각 클래스는 자신의 VMT를 가지게 되며, VMT에 있는 함수 주소의 배열을 이용한다. 가상 메소드는 이 주소를 직접 이용하므로 동작하는 속도는 그렇게 느리지 않다.
오브젝트 파스칼과 C++은 모두 명시적으로 virtual로 선언한 메소드만 다형성을 지원한다. 이는 하이브리드 언어라면 조상 언어의 호환성을 유지해야 하며, 성능 상의 문제 때문에 어쩔 수 없는 것이라고 생각된다.
부모 클래스의 메소드를 새롭게 정의한 클래스가 있을 때, 일반적인 객체에 대하여 그 메소드를 호출할 때 적절한 클래스의 메소드가 호출된다면 무척 편리할 것이다. 이러한 특징을 다형성이라고 한다. 이를 지원하려면, 위에서 지원한 하위형 호환성이 매우 중요한 역할을 하게 된다. 컴파일러는 다형성을 지원하기 위해서 late binding 이라는 기법을 사용하게 되는데, 이것은 특정 함수를 호출하지 않고 런타임에서 객체가 실제로 어떤 클래스의 함수를 호출하게 될지를 알아낸 후에 호출될 함수를 결정하는 방식이다.

- C++

C++에서는 이러한 late binding이 가상 메소드에 대해서만 허용된다. 가상 메소드가 아닌 메소드는 late binding을 지원하지 않는다.

- 자바

자바의 경우에는 'final' 키워드로 지정하지 않는한 모든 메소드가 late binding 을 지원한다. Final 메소드는 재정의될 수 없고, 이들은 다른 메소드보다 빠르게 동작한다. 그러니까, 기본적으로 C++은 early binding 이 디폴트이며, 자바는 late binding 이 디폴트인 것이다. 다르게 말하자면, C++은 효율성을 위해서 OOP 모델을 희생하는 경우가 많다.

- 오브젝트 파스칼

오브젝트 파스칼의 경우에는 C++과 마찬가지로 디폴트는 early binding 이다. 그런데, 오브젝트 파스칼은 'virtual' 키워드로 지정하는 가상 함수 외에 'dynamic' 키워드로 지정하는 동적 함수가 지원된다. 그러나, 이들은 기본적인 개념으로 보아서는 동일한 것이다. 오브젝트 파스칼의 경우에는 'override' 키워드를 사용해야만 이들 가상 함수를 재정의할 수 있다. 이러한 'override' 키워드의 의미는 이 키워드로 지정한 메소드만 컴파일러가 다시 검사하게 된다는 것이다. 이런 방법으로 비교적 효율적인 퍼포먼스를 유지할 수 있다. 또한, 오브젝트 파스칼에서는 가상 생성자(virtual constructor)를 정의할 수 있다.
오브젝트 파스칼은 가상 메소드 이외에 동적 메소드를 지원한다. 이 메소드는 원래 윈도우의 메시지를 사용하기 위해 고안된 것이다. 즉, 수백 개가 넘은 윈도우 메시지를 처리할 때에는 이들 각각에 대한 VMT 를 관리한다는 것은 메모리 공간의 낭비가 될 수 있으므로, 동적 메소드 테이블(Dynamic method table, DMT)을 통해 오버 라이드된 메시지 핸들러에 대한 주소만 관리함으로써 이러한 오버헤드를 줄일 수 있다. 그렇지만, 아무래도 수행능력이라는 측면에서는 다소 핸디캡을 가지고 있다고 보면 된다.

추상 메소드와 클래스

비교적 복잡한 클래스 구조를 만들 때에는, 프로그래머가 다형성을 지원하는데 유리하도록 실제로 구현되지 않는 메소드를 선언할 필요가 있을 수 있다. 이러한 메소드를 추상 메소드(abstract method)라고 하며, 이러한 추상 메소드를 하나 이상 가지고 있는 클래스를 추상 클래스라고 한다.

- C++, 자바

C++의 추상 메소드는 순수한 가상 함수라고 생각하면 된다. 추상 클래스는 하나 이상의 추상 메소드를 가진 클래스로, 이러한 추상 클래스 객체는 생성할 수 없게 되어 있다. 즉, 이런 추상 클래스를 상속받아서 추상 메소드를 실제로 구현한 클래스만 객체를 생성할 수 있다. 자바의 경우에도 C++과 마찬가지로 추상 클래스의 인스턴스는 생성할 수 없다.

- 오브젝트 파스칼

오브젝트 파스칼의 경우에는 추상 메소드를 가지고 있는 추상 클래스의 인스턴스를 생성하는 것이 가능하다. 그렇기 때문에, 프로그램이 추상 메소드를 호출할 가능성도 있는데 이렇게 되면 런타임 에러가 발생하게 된다.

다중상속과 인터페이스(Interfaces)

일부의 OOP 언어는 하나 이상의 기초 클래스를 상속 받을 수 있다. 이를 다중상속(multiple inheritance)라고 한다. 그에 비해 다른 언어에서는 오직 하나의 클래스만 상속받을 수 있지만, 옵션으로 다중 인터페이스나 순수한 추상 클래스(클래스의 순수한 추상 함수로만 이루어진 경우)들을 상속 받아 다중상속의 기능을 대치한다.

- C++

C++은 다중상속을 지원하는 언어이다. 이점은 장점도 단점도 될 수 있다. 다중상속의 장단점에 대한 논의는 이 책의 범위를 넘기 때문에 생략하도록 하겠다.

- 자바, 오브젝트 파스칼

자바와 오브젝트 파스칼은 공히 다중상속을 지원하지 않고, 다중 인터페이스를 지원한다. 이를 이용해서 다형성을 구현할 수 있으며, 이런 특징을 바탕으로 COM 모델에 적합한 언어환경이 지원된다. 오브젝트 파스칼은 자바보다 더욱 COM 에 가까운 인터페이스 모델을 가지고 있다.

RTTI (Runtime Type Identification/Information)

형 검사가 엄격한 OOP 언어는 컴파일러가 이러한 형 검사를 해주어야 한다. 그러므로, 실행 중인 프로그램에는 클래스와 데이터 형에 대한 정보가 별로 필요가 없다. 그러나, 어떤 경우에는 이러한 데이터 형에 대한 정보가 필요한 경우가 있는데, 이런 경우를 위해 각 언어들은 정도에 차이는 있지만 RTTI 를 지원하고 있다.

- C++

C++ 언어는 본래 RTTI 를 지원하지 않는다. 그러나, 이 개념의 지원을 위해 downcast

(dynamic_cast)라는 형태로 일부 기능이 추가 되었다. 이를 이용해서 각 객체에 대한 데이터 형을 확인하거나, 2 개의 객체가 같은 클래스인지 검사할 수 있다.

- 자바

자바는 기본적인 기초 클래스로 Object 클래스가 제공된다. 이 클래스를 통해 클래스에 대한 정보를 추적할 수 있다. Object 클래스의 getClass() 메소드를 사용해서 메타클래스(클래스를 설명하는 클래스의 객체) 정보를 얻을 수 있고, getName() 함수를 사용해서 특정 문자열을 클래스의 이름으로 지정할 수 있다. 또한, instanceof 연산자를 사용할 수도 있다. 자바의 1.0 버전은 클래스에 대한 RTTI 를 광범위하게 지원하고 있지 않지만, 컴포넌트와 비주얼 환경의 발달로 이러한 RTTI 를 많이 지원하게 되었는데, 이것이 자바 빈스(Java Beans)이다.

- 오브젝트 파스칼

오브젝트 파스칼은 가장 광범위한 RTTI 정보를 제공한다. 이를 이용해서 단순한 데이터 형 검사와 downcast 를 할 수 있으며 (is 와 as 연산자를 사용한다), 더 나아가서는 published 로 선언된 요소들을 새로운 RTTI 정보로 등록할 수도 있다. 프로퍼티와 스트리밍 메커니즘(폼 파일 등의)과 오브젝트 인스펙터로 표현되는 델파이의 환경은 기본적으로 이런 RTTI 가 있기에 가능한 것이다. 델파이 클래스의 기초 클래스인 TObject 클래스에는 ClassName, ClassType 메소드가 있는데, ClassType 메소드를 사용해서 특정 객체의 클래스에 대한 참조 값을 돌려 받을 수 있다. 이 메소드를 통해서 반환되는 클래스 형은 TClass 로 이 클래스는 TObject 의 클래스로 선언되어 있는데, 이는 이렇게 TClass 형으로 반환된 클래스는 반드시 사용되기 전에 특정 클래스로 형 변환되어야 한다는 것을 의미한다.

예외처리

예외처리란 프로그램의 에러 처리 부분을 보다 손쉽게 하기 위해, 언어에서 제공하는 표준 메커니즘을 말한다. 예외처리 방식은 언어들 마다 비슷하지만, 내부 동작에는 다소간의 차이가 존재한다.

- C++

C++에서는 throw 라는 키워드를 사용해서 예외를 발생시키며, try 키워드로 예외를 처리하게 될 블록을 설정하고, catch 키워드로 실제로 예외를 처리할 루틴을 작성하게 된다. C++은 예외처리를 할 때 파괴자를 호출해서 스택에 할당된 객체의 메모리를 해제한다.

- 자바

자바는 C++과 같은 키워드를 사용한다. 자바에는 finally 키워드가 있는데, 이 키워드는 객체참조 모델을 지원하는 언어에는 대부분 존재한다. 자바는 기본적으로 가비지 컬렉션을 하기 때문에 이러한 finally 키워드를 사용해서 메모리 이외에 리소스를 해제하는 것을 제한한다. 자바는 예외를 일으킬 수 있는 모든 함수와 예외 클래스가 기본적으로 잘 대응되어야 하며, 이를 컴파일러가 일일이 검사한다. 프로그래머에게는 다소 부담되는 일이지만, 오류가 적은 프로그램을 만드는 데에는 상당히 강력한 무기가 된다. 모든 자바의 예외 객체들은 Throwable 클래스에서 상속받는다.

- 오브젝트 파스칼

오브젝트 파스칼은 raise, try, except 키워드를 사용한다. C++ 과의 차이점은 기본적으로 스택에 객체에 대한 메모리가 할당되어 있지 않기 때문에, 스택을 처리할 필요가 없다는 것이다. 또한, 자바와 마찬가지로 finally 키워드를 지원한다. 델파이의 모든 예외 클래스는 Exception 클래스를 상속한다.

그 밖의 특징들

- C++

C++은 연산자 오버로딩(operator overloading)을 지원한다. 또한, 메소드 오버로딩도 지원하지만 이것은 자바와 델파이 4 에서도 지원된다. 그리고, C++은 프로그래머가 전역 함수를 오버로드할 수도 있으며, 형변환을 담당하는 메소드를 정의해서 형변환 연산자도 오버로딩할 수 있다. 그리고, 템플릿이라는 개념을 지원하여 일반적인 클래스의 재사용성을 높일 수 있다.

- 자바

자바는 멀티쓰레딩을 언어에서 지원한다. 객체와 메소드는 동기화(synchronization) 메커니즘을 지원하는데, 2 개의 동기화된 메소드는 같은 클래스에서 동시에 수행될 수 없다. 이렇게 새로운 쓰레드를 생성하려면 단순히 Thread 클래스를 상속받아서 run() 메소드를 오버라이드하거나, Runnable 인터페이스를 구현하면 된다. 그리고, 백 그라운드에서 가비지 컬렉션을 해주기 때문에, 다소간의 퍼포먼스의 희생을 감수해야 하지만 보다 완벽한 메모리 관리가 가능하다. 그 밖에 자바는 포터블 바이트-코드 아이디어를 채용했기 때문에, 여러

가지 플랫폼에 적용되기 적합한 형태를 가지고 있다.

- 오브젝트 파스칼

오브젝트 파스칼은 클래스 참조를 지원하기 때문에, 메소드 포인터를 아주 쉽게 사용할 수 있다. 이러한 메소드 포인터는 이벤트 모델의 기초가 되며, 이를 프로퍼티로 사용할 수 있다. 프로퍼티는 메소드가 데이터에 접근하는 방법을 숨겨주는 방법으로 사용되는데, 데이터를 직접 읽거나 쓸 수도 있고, 접근 메소드(access method)를 사용해서 데이터를 조작하는 방법도 가능하다. 데이터에 접근하는 방법을 바꾸더라도 코드를 호출하는 방법을 바꿀 필요가 없기 때문에, 재사용성이라는 측면에서 대단히 유용하다. 다르게 말하면, 다른 어떤 OOP 언어보다도 강력한 캡슐화 방법을 지원하는 것이다. 또한, 델파이 4 에서는 그동안 지원하지 않았던 메소드 오버로딩과 파라미터의 디폴트 값도 지원하므로 보다 강력한 언어적 특성을 가지게 되었다.

정 리 (Summary)

이번 장에서는 OOP 언어로는 가장 많이 사용되는 대표적인 세가지 언어에 대해서 비록 수박 겉핥기에 가까운 방법으로나마 살펴 보았다. 이들 언어는 기본적으로는 OOP 의 개념을 공통적으로 구현하고 있지만, 그 구현의 목적과 방향성에 다소간의 차이가 있다는 것을 알 수 있을 것이다. C++은 다소간의 복잡성을 가지고, OOP 의 기본적인 개념에 배치되는 여러가지 모습을 가지고 있지만 파워와 유연성이라는 측면에 초점을 맞춘 언어라고 할 수 있고, 델파이는 쉽고, 비주얼 환경과 윈도우 환경에 적합하도록 설계 되었지만 파워도 포기하지 않은 언어이다. 자바는 이식성에 주안점을 두고, 스피드를 다소 희생하도록 설계 되었다.
오브젝트 파스칼과 C++에 대한 차이를 설명할 때에는 앞에서 둘러본 여러가지 항목들의 차이도 중요하겠지만, 보다 중요한 것은 기본적으로 이들 언어의 조상이 된 파스칼과 C 가 디자인되고 사용된 배경과 목적에서 차이점을 찾아볼 수 있다.
파스칼은 명확하고, 안전하며 동시에 모듈화가 용이하도록 디자인된 언어인데 비해, C 는 노력을 적게 들이면서도 가능한 저수준의 메모리를 쉽게 접근할 수 있도록 디자인 되었다. 파스칼은 데이터 추상화와 다중 레벨의 모듈화를 직접 지원하는 고수준의 언어이지만, C 는 중간 레벨로서 기계와 고수준 언어의 중간 수준을 목표로 만들어졌다. 파스칼은 프로그래머를 보호하기 위해서 만들어진 언어인데 비해 C 는 최대한 프로그래머를 믿고 만들어진 언어이다. 이러한 파스칼과 C 의 바탕에서 객체지향 언어의 장점을 접합시킨 것이 바로 오브젝트 파스칼과 C++인 것이다. 자바는 여기에 비해 직접적인 조상이 되는 언어가 없는 순수한 객체지향 언어이다.
여기서 확실히 말할 수 있는 것은 이러한 언어의 특장점 들이 실제 세계에서 사용되는 개발

환경을 좌우하지는 않는 다는 것이다. 실제로 중요한 것은 운영체제와 최근에 불어닥치는 인터넷 세계, 그리고 Win32API, 액티브 X 등의 모든 주변환경들과 이들의 관계, 개발자 들이 선택한 마켓 쉐어 등이라고 할 수 있다. 아무리 훌륭한 언어라도 실제로 사용되지 않는 경우는 부지기수이다. 예를 들어, 오브젝트 파스칼과 자바의 모델인 에펠(Eiffel)은 대단히 훌륭한 OOP 언어이지만 대학에서나 학문적으로 사용될 뿐 실제 시장에서는 외면받고 있는 언어이다.

결국, 어떤 언어가 좋은가 ? 하는 소모적인 논쟁 보다는 실제로 사용할 언어의 특징을 잘 파악하고, 이를 잘 이용하는 것이 좋은 프로그램을 만들 수 있는 기반이 될 것이다. 그런 측면에서 이번 장의 소기의 목적이 달성되었으면 하는 것이 필자의 소박한 바램이다.