**객체지향 프로그램의 다섯 개의 기반 개념.**

1. Objects

2. Classes

3. Encapsulation

4. Inheritance

5. Polymorphism

"객체지향 프로그램은 어떠한 목적을 이루기 위해 협업하도록 구성되어지는 객체들의 집합이다."

**객체란 무엇인가? What Are Objects?**

프로시저를 구조적 프로그램을 작성하는데 사용 하다고 단정하면, 객체란 객체지향 프로그램을 작성하는데 사용 되는 것 입니다. 객체지향 프로그램은 어떠한 목적을 이루기 위해 협업하도록 구성되어지는 객체들의 집합입니다.

모든 객체는 :

**데이터를 가지고 있습니다.** - 데이터는 객체의 상태를 기술하는 정보를 저장합니다.

**행위의 집합을 가지고 있습니다.** - 이 행위들은 메시지를 받았을 때 객체가 어떻게 해야 하는지 알고 있는 것 입니다.

**개체를 구분하는 아이덴티티를 가지고 있습니다.** - 어떠한 객체를 다른 객체와 구분하는 것을 가능케 합니다.

구조적 프로그램에서 사용되는 레코드,구조체처럼 객체도 데이터를 담고 있습니다. 이처럼 생각하면 객체는 이전 섹션에서 보았던 급여관리 프로그램에서 보았던 하나의 직원 레코드처럼 보입니다. 객체의 데이터는 객체의 상태를 표현하기 위해 사용됩니다. 예를 들어, 객체의 데이터를 가지고 그 직원이 정규직원인지 파트타임인지 감지할 수 있습니다.

직원 객체는 작업 행위를 가지고 있다는 데서, 절차적 프로그램의 직원 레코드와는 다릅니다. 이 작업들은 아마도 객체의 데이터를 읽거나 변경하는데 사용될 겁니다. 객체는 자신의 등에 데이터를 메고 있는 작은 프로그램처럼 동작합니다.

여러분이 어떤 것을 알고 싶거나, 어떤 일을 하기를 원한다면 객체에게 작업을 수행하도록 요청하면 됩니다. 이것을 객체지향 세계의 말로 "메시지" 보낸다 라고 합니다. 응답은 객체의 두 번째 특성입니다.  직원 객체는 내장된 행위로 급여에 대해 어떻게 말해야 하는지, 우편물 주소 라벨을 어떻게 인쇄해야 하는지 알고 있을 것입니다.

객체의 세번째 특성은 유일한 아이덴티티를 가지고 있다는 것 입니다. 이것이 모든 객체가 관계형 데이터베이스에서 말하는 ID 값이나 Primary Key를 가져야 함을 의미하는 것은 아닙니다. 객체는 구조적 언어의 '변수'와 매우 닮았다고 할 수 있습니다. 정수 변수 i 와 j는 같은 값을 가질 수 있지만 여전히 구분 될 수 있습니다.

**클래스란 무엇인가? What Are Classes?**

자바 프로그램이 객체의 집합으로 이루어진다면, 대체 클래스란 뭘까? "학생" 객체의 집합인가? 아닙니다. 클래스는 객체 생성의 청사진 blueprint 입니다. 여러분은 실제 객체에 대한 코드를 절대로 작성할 수 없으며, 객체를 만드는데 사용되는 패턴을 작성할 수 있을 뿐입니다. 클래스와 객체의 구별하는 것은 미묘하지만 객체지향 디자인을 이해하기 위한 기본입니다.

클래스와 객체 사이의 관계를 이해하는 좋은 방법은 구조적 프로그래밍 언어의 '타입 Type' 과 '변수 Variable'를 떠올리면 됩니다. 변수의 타입을 이야기 할 때 그 변수로 어떤 작업을 할 수 있는지, 저장할 수 있는 범위가 어떻게 되는지로 기술합니다. 예를 들어, 정수 변수는 복소수를 저장 할 수 없습니다. 같은 방법으로 정수와 소수의 곱하기 연산은 가능하지만, 맞춤법 검사는 오로지 문자열에 대해서만 가능합니다.

변수는 값을 저장한다는데 주목하라 : 타입이 하는 일이 아닙니다. 타입은 추상이라고 말할 수 있습니다. 실제로 정수 값을 저장하려면 정수 변수를 생성해야만 합니다. 정수 타입으로 지정된 청사진으로 변수가 생성되면 그 변수는 정수입니다. 정수 타입에 결정된 규칙에 의해 저장되고, 행동합니다. 각각의 i,j,k 정수를 정수 타입의 '인스턴스'라고 말합니다.

객체와 클래스는 비슷한 관계를 가지고 있습니다. 클래스는 객체가 소유하게 될 속성 attributes 과 행위 behaviors 를 정의합니다. 클래스는 객체 생성의 청사진 blueprint 입니다. 자바 프로그램을 작성할 때 남이 작성한 클래스를 사용하기도 하고, 자신의 장치에 맞는 새로운 클래스를 정의하기도 합니다. 새로운 클래스를 생성하는 일은 두 부분으로 나누어져 있습니다.

1. 객체의 상태를 저장하는데 사용될 속성을 정의 합니다.

2. 객체가 이해할 수 있는 메시지와 메시지에 응답하는 과정을 정의 합니다. 각각의 메시지에 대해 메소드 method 라고 불리 우는 프로시저를 만들고, 이것을 구현합니다.

**캡슐화란 무엇인가? What is Encapsulation?**

객체,클래스와는 다르게 캡슐화는 자바 언어의 요소가 아닙니다. 캡슐화는 잘-설계된 클래스를 만드는데 사용되는 기법입니다. 잘-설계된 객체-지향 클래스는 캡슐화 기법을 요구합니다.

"좋아, 캡슐화라는 것에 기꺼이 주목 할 게, 하지만 마치 우주인이 가지고 있을 만한 물건처럼 들리는데, 도대체 그게 뭐야." 캡슐화는 프로그램을 포장 packaging 하는 작업입니다. 클래스는 두 부분으로 나눌 수 있습니다: **인터페이스 interface 와 구현 implementation.**

"잠깐, 당신은 이미 클래스는 속성 attributes 과 메소드 methods 의 두-부분으로 구성되어 있다고 말했잖아요, 인터페이스와 구현은 속성과 메소드를 부르는 새로운 용어 인가요?" 라고 반응할 수 있습니다.

여러분의 객체는 속성과 메소드로 만들어져 있습니다. 일부 속성과 메소드는 객체의 외부에서 접근할 수 있고 이것을 **인터페이스 interface**라고 합니다. 다른 속성,메소드는 객체 자신만의 사적인 용도로 예약되어 있고 이것을 **구현 implement**이라고 합니다. 구현으로 부터 인터페이스를 분리하는 것은 객체-지향 프로그램을 설계할 때 가장 중요한 결정 입니다.

구현으로 부터 인터페이스를 나누는 일의 가치를 살펴보자면, 자동차를 연상하면 쉬울 것입니다. 자동차의 인터페이스는  핸들, 가속 페달, 브레이크로 간단하고 규격화 되어 있습니다. 운전방법을 한번 배우기만 하면 됩니다. 반면에 자동차의 내부 동작은 점화, 실린더, 연료 분사 등등 매년 역동적으로 변경됩니다. 여러분이 각기 다른 타입의 자동차의 점화 시스템을 직접 통제해야 한다면, 새차를 운전하기란 매우 어렵다는 걸 알게 될 겁니다.

잘-설계된 클래스 Well-Designed Class 는 이런 특성을 가지고 있습니다. 인터페이스는 여러분의 클래스와 어떻게 상호작용 해야 하는지 완벽하게 묘사합니다. 그리고 클래스의 대부분의 속성은 감추어 진다는 것을 의미합니다. 사용자는 데이터를 수정하기 위해서 메소드를 사용할 것입니다.

Ex)

public class Car{  
    private Engine engine;  
    protected Handle handle;  
      
    public Car(){  
        //생성자  
    }  
}

위에서 class 앞의 public이 존재하기 때문에 이 클래스는 누구나 사용할 수 있는 클래스가 됩니다. 인스턴스 변수인 engine은 private이기 때문에 Car 클래스 안의 메소드에서만 접근이 가능합니다. Car 이외의 클래스에서 접근은 불가능합니다. handle은 protected이기 때문에 Car를 상속받은 클래스에서는 접근이 가능하게 되는 것입니다.  
  
접근이 커지는 순서대로 나열하면 private(클래스 안) < protected(하위 클래스까지) < (default)(패키지 내의 클래스까지) < public(누구든지) 의 순서가 됩니다.

**상속이란 무엇인가? What Is Inheritance?**

상속 계층을 구조화 하는 것은 클래스 설계의 두 번째 결정입니다. 캡슐화는 변경으로 부터 클래스를 견고하게 만드는데 필요합니다. 상속은 클래스 관계에서 "가족 famillis" 개념과 연관이 있습니다.

"상속의 진가는 강력한 추상 구조화 이다."

클래스의 상속관계를 정의하면 두 가지 의 큰 이점을 얻을 수 있습니다. 새로운 종류의 서브클래스를 작성하면, 여러분은 부모 super class 에 이미 내장된 기능들을 사용할 수 있습니다. 이는 상속을 사용하는 가장 일반적인 이점입니다. 그러나 이게 전부는 아닙니다. 프로그래밍 언어에서 프로시저가 처음 사용 될 때, 중복 코드를 줄이라고 요구합니다. 그리고, 복잡한 컴퓨터를 구성하고 정복하는 엄청난 힘이라는 것이 증명되었습니다.  상속의 진가는 추상화를 구조화 한다는데 있습니다. 프로시저는 복잡한 문제를 단순한 부분으로 나눌 수 있게 합니다. 상속을 사용하면 공통적인 요소를 슈퍼클래스에 정의 함으로써 일반화 할 수 있습니다.

구조적 프로그래밍은 분해 decomposition (프로시저 형태로의 분해) 에 의한 추상화에 기반하고 있습니다. 객체-지향 프로그래밍은 추상 메커니즘을 이용해 추상화를 포함합니다. 클래스화 classification 에 의한 추상화를 기반으로 상속을 적용하는 메커니즘을 이용합니다. 자연과학에서처럼 계층구조를 묘사할 수 있음이 계층적 클래스화의 힘입니다. 계층구조는 우리 행성의  수백만 종에 달하는 동,식물의 정보를 구조화하는 강력한 도구 입니다.

Ex) class A { **// A라는 클래스 생성**

int num;

}

class B extends A {**// B 클래스가 생성되고 그와 동시에 A 클래스를 상속받는다.**

int num1;

}

class C extends B { **// C 클래스가 생성되고 그와 동시에 B 클래스를 상속받고 B 클래스에 상속된 A 클래스도 함께 상속받는다.**

int num2;

}

**다형성이란 무엇인가? What Is Polymorphism?**

마지막으로, 다형성은 객체-지향 프로그래밍의 기반 원칙입니다. "많은 형태 many shapes" 라는 뜻을 가진 그리스어에서 유래한 말로, 다형성은 캡슐화, 상속과 함께 작동해서 객체-지향 프로그램의 **흐름 제어 flow of control**를 단순화 합니다.

"다형성은 캡슐화, 상속과 함께 동작 함으로써 객체-지향 프로그램의 흐름 제어를 단순화 합니다."

흐름 제어, 다음에 일어날 일을 알아야 한다는 것은 컴퓨터 프로그램의 아킬레스 건 입니다. 이게 바로 프로그램이 복잡하게 되는 주요 원인입니다. 모든 경로를 추적하지 않고서는 객체가 가질 상태의 가능성을 검사 할 방법이 없습니다. 따라서 여러분은 프로그램을 테스트 하다가 지쳐 버릴 것입니다. 구조적 프로그래밍에서 프로그램의 흐름 제어를 단순화 하는 초기 시도는 조건 없는 분기로 규범화된 순서 구조, 선택, 반복을 제공하는 것 이었습니다. 같은 방법으로 다형성은 상속 계층의 연관된 객체에 메시지를 보냄으로써 단순화 합니다.

객체에 메시지를 보낼 때, 그 객체는 메시지에 응답할 메소드를 가지고 있어야 합니다. 클래스가 상속 계층에 연결되어 있다면, 모든 서브클래스는 부모의 인터페이스를 자동으로 상속 받습니다. 어떤 일이라도 슈퍼클래스 객체가 할 수 있는 것이라면 서브클래스 객체도 할 수 있습니다. 예를 들어, 클래스 A가 클래스 B의 서브클래스 라면, A 객체는 B객체가 할 수 있는 모든 일을 할 수 있습니다. A객체는 B객체라고 말할 수 있고, 이런 경우의 상속 관계를 ISA 관계 라고 부릅니다. 비록 서브클래스 객체가 슈퍼클래스 객체가 하는 것처럼 동일한 메시지에 응답할 책임이 있다 하더라도, 메시지는 동일한 동작을 하도록 강제하지 않습니다. 이것을 이해할 필요가 있습니다. 각각의 서브클래스는 슈퍼클래스가 정의하고 있는 적절한 응답 또는 새로 정의된 특화된 응답에 의존하고 있습니다. 그래서, 각각의 서브클래스는 같은 메시지에 다른 응답을 할 수 있습니다. 이게 "많은 형태"라고 말하는 의미 입니다.