
K G 아 이 티 뱅 크
JAVA
J A V A

클래스 기본

클래스 기본

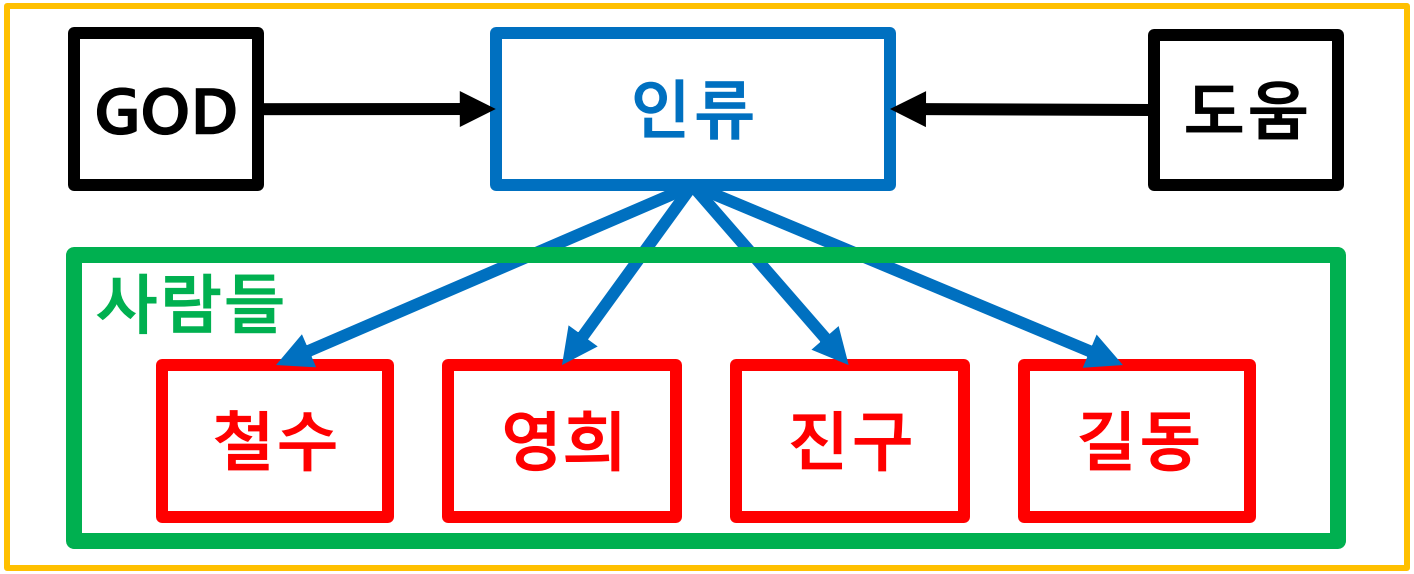
- ❖ **프로그램은 언제나 현실의 문제를 해결하는 것이 목표**
  - ➢ 심심하다 -> 재미를 느끼는 프로그램 -> 게임
  - ➢ 계산이 어렵다 -> 계산하는 프로그램 -> 계산기
- ❖ **프로그램에는 코드로 현실의 상황이 묘사되어야 함**
  - ➢ 절차지향 : 정해진 순서/절차대로 현실은 진행된다
- ❖ **문제 : 우리의 현실은 절차기반으로만 묘사될 수 없음**
  - ➢ 강의를 들으며, 필기하고, 딴 짓도 할 수 있음

클래스 기본

- ❖ **해결 : 현실을 절차가 아닌 그대로 묘사해야 비슷해짐**
  - ➢ 컴퓨터로 이동하여 LOL을 확인하고 실행하여 이용한다
  - ➢ **나**는 **컴퓨터**로 **LOL**을 즐긴다.
- ❖ **현실을 최대한 흡사하게 모방하기 위한 개념 : OOP**
  - ➢ **Object Oriented Programming** :객체지향 프로그래밍
  - ➢ 절차지향 : 대상을 가지고 어떤 처리하여 뭘 만드는가?
  - ➢ 객체지향 : 대상끼리 동작을 하여 어떤 결과가 나오나?

클래스 기본

- **클래스(Class)** : 현실에 있는 것을 언어로 묘사한 코드
  - 프로그래밍 언어에서는 **자료형**으로 묘사됨
- **객체(Obejct)** : 클래스를 통해 만들어지는 것들
  - 값을 저장하기 위해 만들어진 **저장공간들**에 대응됨
- **인스턴스(Instance)** : 만든 것들 중 값을 가지는 하나
  - 값이 있으면서 다른 것들과 **구별된 저장공간**에 대응됨

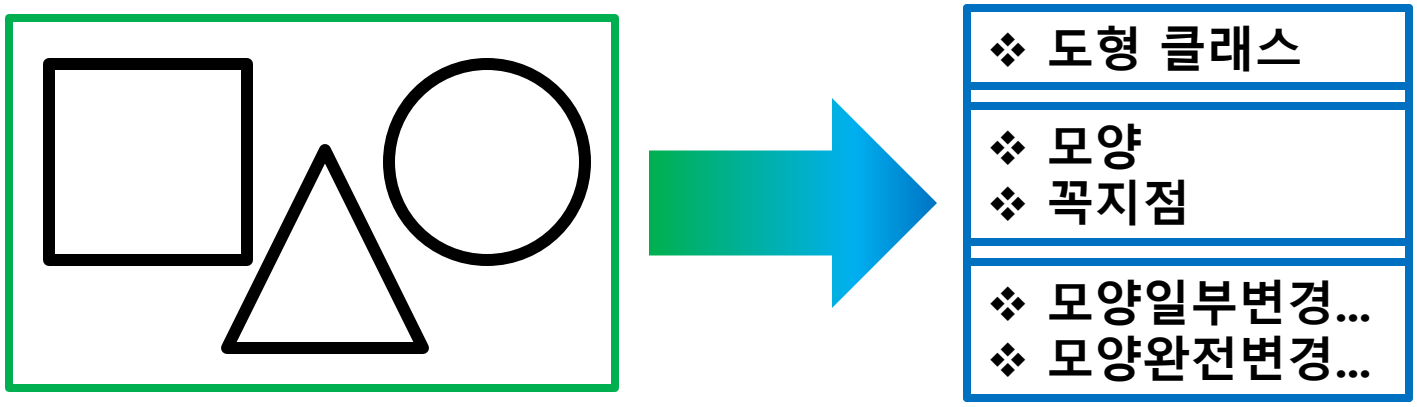


클래스 기본

❖ **클래스의 생성 : 추상화와 캡슐화가 잘 되어야 함**

- 2가지가 제대로 이루어지지 않으면 클래스가 아님

1. **추상화(Abstraction)**
   - 현실의 객체들이 가지고 있는 자료와 동작을 묘사
   - 서로 다른 객체에서 공통점을 뽑아내는 것
2. **캡슐화(Encapsulation)**
   - 공개될 것과 공개되면 안되는 것을 구분
   - 보호해야할 것과 보호할 필요가 없는 것을 구분



추상화(Abstraction)

추상화

- ❖ 프로그램은 언제나 현실의 문제를 해결하는 것이 목표
  - ➢ 묘사/모방하는 것은 좋지만 **정교하게 묘사할 수 없음**
  - ➢ 현실은 **자료가 넘쳐나기 때문에 한계가 존재**함
- ❖ 프로그램의 목적에 따라 적절하게 생략/축약해야 함
  - ➢ 전부다 적을 수도 없고, **현대 컴퓨터로는 한계가 있음**

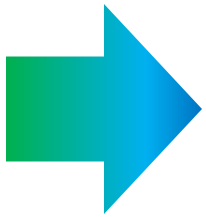
추상화

- **추상화는 객체지향 프로그래밍의 첫번째 특징이자 과정**
  - 특징 : 객체를 **공통된 특징과 동작으로 묘사**할 수 있다
  - 과정 : 대상으로부터 **고유의 특징과 동작을 뽑아냄**
- **프로그램에 필요한 것들만 골라내서 구현해야만 함**
  - 과도하게 뽑아내면, **불필요한 변수로 인하여 관리가 힘듬**

추상화

❖ 추상화 첫번째 단계 : 분석하여 선정하기

- 특징은 하나의 값이며, 변수로 바뀌어 표현됨
  - **필드, 멤버변수, 속성** 등으로 불림
- 동작은 절차이며, 메서드로 표현됨
  - **메서드, 기능, 동작, 액션** 등으로 부름



❖ 특징	
이름 : 강철맨 나이 : 150 비행능력 : 유 공격력 : 100 방어력 : 100	❖ 동작 대상공격 공격방어 비행하기 긴급회피 긴급탈출

추상화

❖ **추상화 두번째 단계 : 약속된 문법대로 작성하기**

- 중구난방으로 마구잡이로 작성하면 안됨
- 관리하기 위해서로 일정한 규칙이 필요함
- **정해진 규칙 내에서 최대한 구현해야 함**
- **규칙을 무시하고 구현하면 되는 것이 없음**

❖ **특징**
이름 : 강철맨
나이 : 150
비행능력 : 유
공격력 : 100
방어력 : 100

❖ **동작**
대상공격
공격방어
비행하기
긴급회피
긴급탈출

❖ **이건 뭐죠 ? ? ? ? ? ? ? ? ?**

```
// 이건 클래스임
int name = "강철맨";
double age = 150;
char 비행능력 = "있음";
String 공격력 = 100;
int def = 100;
atk(n1, n2) {
    n1 + n2;
}
```

? ? ?
? ? ?
? ? ?
? ? ?
? ? ?
? ? ?
? ? ?
? ? ?

추상화

❖ 클래스의 정의 : 작성할 내용이 많으니 주의

➢ 편집기의 문법 자동완성을 잘 이용해야 함

❖ 클래스 정의 예시

```
class SampleClass {
  String value;
  int num;
  double fnum;
  .....
  int method1(void) {
    return num;
  }
  void method2(double fnum) {
    this.fnum += fnum;
  }
  .......
}
```

❖ 클래스 정의을 시작 / 종료

중괄호 내에 해당 클래스로 만들어지는 객체가 이용할 **변수/메서드를 선언/정의함**

❖ 객체의 특징을 묘사함

1. 원하는 자료형으로 원하는 변수를 원하는 만큼 선언
2. **메서드에서 사용 가능**

❖ 객체의 동작을 묘사함

1) 동작을 묘사하는 코드를 메서드로 묶어서 작성함
2) **this라는 키워드가 사용됨**

추상화

❖ 클래스의 사용 : 자료형으로 사용하며, 객체가 생성됨

➢ 편집기의 문법 자동완성을 잘 이용해야 함

❖ 클래스의 사용 예시

```
SampleClass Object
        = new SampleClass();

Object.value1 = 100;
Object.value2 = 3.14;
Object.value3 = "맞는 값";

Object.method1();
Object.method2();
Object.method3();
```

❖ **객체를 생성 = 변수 생성**

1. 클래스는 설계도이며 참조 자료형으로 분류됨
2. **new 키워드를 이용함**

❖ **객체의 특성을 이용함**

1. 저장공간 중 일부를 변수명으로 구분하여 사용
2. **이런 방식으로는 잘 안씀**

❖ **객체에게 동작을 시킴**

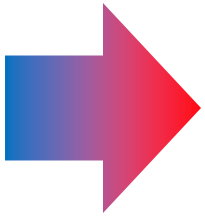
1. 정의한 메서드를 통해 객체가 동작을 수행
2. **객체의 주 사용방법**

캡슐화(Encapsulation)

캡슐화

- ❖ 추상화가 완료되면 대상은 프로그래밍언어로 묘사됨
  - ➢ 특징은 변수로, 동작은 메서드로 변형되어 완성됨
- ❖ 완성된 설계도는 **객체를 만들어 쓸 때 핵심 정보임**
  - ➢ 내부의 특징과 동작은 묘사한 그대로의 작동하게 됨
  - ➢ 이러한 것들이 **제한없이 사용되면 스파게티코드가 됨**
  - ➢ **코드의 작성은 자유로워도 그 사용은 통제되어야 함**

강철맨 클래스	
❖ **특징** 이름 : 문자열 나이 : 정수 비행능력 : 논리 공격력 : 정수 방어력 : 정수	❖ **동작** 대상공격 공격방어 비행하기 긴급회피 긴급탈출



```
강철맨.이름 = "강철맨";
강철맨.공격( );
강철맨.이름 = "아이언맨";
강철맨.방어력 = 99999;
강철맨.방어( );
강철맨.공격력 = 1;
강철맨.공격( );
```

캡슐화

- ❖ 클래스의 각 요소는 필요에 따라 사용에 제한을 둠
  - ➢ 이러한 제한을 위해 **접근제어 지시자**가 설정됨
- ❖ 객체지향에서 접근제어는 다양한 형태로 존재함
  - ➢ 객체의 변수와 메서드의 무분별한 사용을 통제
  - ➢ 부가적으로 현실에 존재하는 개념을 구현하게 됨
    - ➢ **가변/불변 및 공개/비공개 등의 개념이 해당됨**

종류	private	default	protected	public
동일 클래스	가능	가능	가능	가능
동일 패키지의 자식 클래스	불가	가능	가능	가능
동일 패키지의 다른 클래스	불가	가능	가능	가능
다른 패키지의 자식 클래스	불가	불가	가능	가능
다른 패키지의 다른 클래스	불가	불가	불가	가능

캡슐화

❖ 접근제어 지시자 작성요령 : 클래스/메서드/변수 앞에 작성

1. **일반적으로** 필드는 private / 메서드는 public으로 설정
2. 클래스에는 **어디에서 이용할 수 있는가를 조정**할 때 사용
   - ➢ **용도(독립/상속 등)에 따라서 결정되니 주의**
3. default는 **기본설정이라는 의미**이며, 작성하지 않음
   - ➢ **다른 용도의 예약어가 있어 작성해도 사용할 수 없음**

❖ 접근제어 지시자 사용 예시

```
public class SampleClass {
   private String value;
   private int num;
   .....
   public int method1(void){}
   public void method2(void){}
   .......
}
```

❖ 클래스 사용범위

다른 소스파일에서 사용가능

❖ 객체의 필드 사용범위

만들어진 객체의 필드는
클래스 내에서만 사용가능

❖ 객체의 메서드 사용범위

만들어진 객체의 메서드는
자유롭게 사용할 수 있음

캡슐화

❖ **접근제어를 통해 정보를 보호하지만 동시에 사용도 불가능**

- ➢ 생성된 객체에서 **private 필드는 볼 수도 알 수도 없음**
- ➢ 하지만, 일정하게 **통제된 방법에 의한 사용은 막지 않음**

❖ **getter / setter** - private 필드를 이용하기 위한 메서드

❖ **setter / getter의 예시**

```
public class SampleClass {
  private String value;
  private int num;
  .....
  String get(void)
  { return value; }
  void set(String value)
  { this.value = value; }
  .......
}
```

❖ **private로 설정된 필드**
클래스 내부에서만 사용가능

❖ **getter : 가져오는 메서드**
필드의 값을 **외부로 return만** 시켜주는 메서드

❖ **setter : 저장하는 메서드**
외부의 **값을 필드로 저장만** 시켜주는 메서드

생성자(Constructor)

생성자

❖ getter / setter를 통해 필드의 무분별한 사용은 방지됨

- 객체의 필드명은 보호되며, 통제된 방법으로만 이용됨
- 필요하면 이용하면 되고, 필요하지 않으면 안 해도 됨
- 하지만 생성하고 나서 최초 한번의 사용은 반드시 필요함

❖ 객체를 여러개 생성한다면...

```
SampleClass obj1 =
        new SampleClass();
SampleClass obj2 =
        new SampleClass();
SampleClass obj3 =
        new SampleClass();
obj1.set("ABC", 123);
obj2.set("ZXC", 432);
obj3.set("AAA", 999);
```

❖ **객체들을 위한 참조변수**
배열같은 것으로도 대체 가능

❖ **새로운 객체를 생성중**
객체는 기본적으로 복사안됨
객체의 이름만 늘어남

❖ **각 객체별로 값을 저장**
최소 한번은 해야 하는 작업
따로 하기에는 번거로움

생성자

❖ 객체생성 & setter실행을 별개로 하는 과정은 귀찮음

- 접근제어가 적용된 클래스의 필드를 쓰기 위한 방법
- 이게 귀찮으니 이를 준비해주는 정적 메서드로 수행
- 단, 이럴 경우 관리소요가 많아지게 되어 복잡해짐

❖ 메서드를 따로 만든다면…

```
static SampleClass getObj
( String value1, int value2 )
{ SampleClass newObj =
          new SampleClass();
  newObj.set(value1, value2);
  return newObj; }
SampleClass obA = getObj("A",1);
SampleClass obB = getObj("B",2);
SampleClass obC = getObj("C",3);
```

❖ **하나로 합친 메서드**

필요한 곳에 만들어야 함

❖ **필요한 곳에 만드는 중**

만드는 과정은 동일하게 진행
필요한 곳에 필요한 만큼…

❖ **생성된 객체를 받아서 연결**

직접 작성보다는 편리함
해당 메서드가 있다면…

생성자

❖ **생성자 : 객체 생성 메서드를 클래스에 포함시켜 놓은 것**

- 클래스의 원래 달려 있는 것을 구체적으로 만드는 것
- 클래스명과 동일한 이름으로 메서드를 정의해야 함
- **new 키워드를 이용할 때 자동으로 호출됨**

❖ **생성자를 클래스에 작성**

```
public class SampleClass {
   ......
   public SampleClass(
   String value1, int value2) {
     this.value1 = value1;
     this.value2 = value2;
   }
}
SampleClass objA =
          new SampleClass("A",1);
```

❖ **클래스 내부에 작성됨**
동일한 이름으로 메서드 준비

❖ **필요한 곳에 만드는 중**
참조 변수를 준비하는 과정은
변화없이 똑같음

❖ **생성하며 필드를 채우게 됨**
1. 값 저장을 깜빡할 수 없음
2. **이용 및 관리가 편리해짐**

생성자

❖ 생성자도 메서드이기 때문에 오버로딩이 가능함

- ➢ 매번 값을 넣은 것은 번거롭기 때문에 작성함
- ➢ **매개변수 수량에 따라 알맞는 동작으로 정의함**

❖ 생성자 오버로딩

```
public class SampleClass {
 ......
 public SampleClass() {
   value1 = null; value2 = 0;
 }
 public SampleClass(String value1) {
   this.value1 = value1; value2 = 0;
 }
 public SampleClass(
 String value1, int value2) {
   this.value1 = value1;
   this.value2 = value2;
 }
}
```

❖ **기본 생성자**

1. 모든 필드를 0으로 초기화
2. **다른 생성가 없으면 자동생성**

❖ **일부 생략한 생성자**

1. 일부 필드에만 값을 저장함
2. **그 외에는 상수로 초기화**

❖ **둘 다 있는 생성자**

1. 모든 필드에 값을 저장함
2. **각 필드에 모두 넣어야 함**