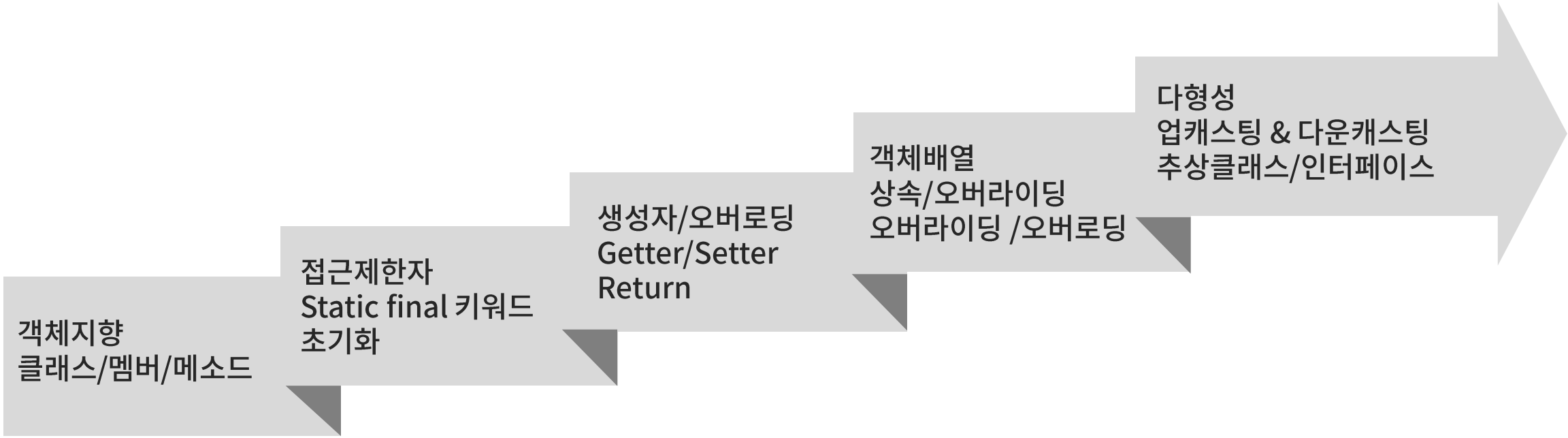
객체 (Object)

-

2조 발표자 주수현

목차 (Index)


객체지향
클래스/멤버/메소드
접근제한자
Static final 키워드
초기화
생성자/오버로딩
Getter/Setter
Return
객체배열
상속/오버라이딩
오버라이딩 /오버로딩
다형성
업캐스팅 & 다운캐스팅
추상클래스/인터페이스

-

객체지향
프로그래밍

< 객체 >

객체란 자신만의 특성을 가지고 구분되어지는 사물이나 개념등을 칭한다.

< 객체지향 프로그래밍 >

객체들 간의 상호작용 개념을 컴퓨터에서 언어를 사용하여 구현하는 것을 객체지향프로그래밍이라 하며 이 언어를 객체지향언어라 한다.

< Java >

JAVA는 객체지향언어 중 하나이다. JAVA에서 객체란 클래스에 정의된 내용대로 new연산자를 통하여 heap 메모리 공간에 할당된 결과물(object)을 말한다.

-

객체지향언어
특징

< 캡슐화 >

공통요소를 추출하는 추상화를 통해 정리된 데이터들과 기능을 하나로 묶어 관리하는 기법을 말한다.

외부에서 데이터에 직접 접근하는 것을 원칙적으로 방지하고 그 데이터를 처리하는 외부에서 접근 가능한 메소드를 클래스 내부에 작성한다. (정보은닉)

< 상속 >

다른 클래스가 가지고 있는 멤버들을 상속을 받은 새 클래스가 자신의 멤버처럼 사용할 수 있는 기능을 말한다.

연관된 클래스들에 대한 공통점을 정하고 코드의 중복을 제거하여 프로그램의 효율성을 높인다.

< 다형성 >

여러 가지 객체의 타입을 한가지 타입의 객체로 처리할 수 있는 기법을 말한다.

-

클래스

< 클래스 >

클래스는 **객체의 특성에 대한 정의를 한 일종의 객체 설계도**이다.

클래스는 필드와 멤버함수(메소드)로 이루어지며 클래스의 구조는 다음과 같다.

```
[접근제한자][예약어]class 클래스명{
    [접근제한자][예약어]자료형 변수명; //필드
    [접근제한자]생성자명(){}
    [접근제한자]반환형 메소드명(매개변수){}//메소드
}
```

-

필드

멤버 변수와 멤버 상수를 필드라 하며 클래스 내부에 선언한다.

< 인스턴스 변수 >

클래스 안에 선언 되며 클래스에 대한 객체를 heap영역에 할당할 때 객체 공간안에 생성되는 필드를 말한다.

< 상수 >

필드 예약어에 final을 붙인 변수 를 말하며 하나의 값만 저장해야 한다.

< 클래스 변수 >

클래스 안에 선언시 static 예약어를 붙인 변수 를 말하며 해당 클래스 객체들이 모두 공유하는 필드를 말한다.

< 지역 변수 >

메소드{}내에 선언된 변수로 메소드 내에서만 사용 가능하며 메소드 실행 시 메모리에 생성되었다가 메소드 종료 시 소멸된다.

-

필드

< 멤버함수(메소드) >

메소드는 수학의 함수와 비슷하며 호출하여 사용한다. 이때 호출을 마친 후 반환 값이 있거나 없을 수 있다. 메소드 표현식은 다음과 같다.

```
[접근제한자][예약어] 반환형 메소드명(매개변수){}
```

< 인스턴스메소드 >

클래스 내의 일반 메소드 로 객체의 참조변수를 통해 호출 한다.

< 클래스 메소드 >

메소드의 예약어에 *static 키워드를 붙인 메소드* 이며 참조변수 대신 클래스명을 사용하여 호출한다.

-

필드

< 메소드 반환형 >

void : 반환 값이 없을 경우 반드시 작성

기본 자료형 : 반환 값이 기본 자료형일 경우

배열 : 반환 값이 배열인 경우 주소값 반환

클래스 : 반환 값이 해당 타입 객체일 경우 주소값 반환

< 메소드 매개변수 >

() : 매개변수가 없을 시

기본 자료형 : 기본 자료형 매개변수 사용 시

배열, 클래스 : 해당 배열과 클래스의 참조변수

-

접근제한자

< 접근제한자 >

객체 지향에서는 사용자가 굳이 알 필요가 없는 정보를 사용자로부터 숨겨야한다는 개념이 있는데 이럴 때 쓰는 것이 접근 제한자 입니다.

접근제한자를 사용하여 **사용자가 알 필요가 없는 정보를 은닉하고**
최소한의 정보만으로 프로그램을 손쉽게 사용할 수 있게 됩니다.

(-)private : 캡슐화를 위해 사용되는 제한자로 클래스 안에서만 접근이 가능하며
클래스 밖에서는 접근이 불가능합니다.(클래스 선언 불가)

(#)protected : 비 상속시에는 default와 동일하며, 같은 패키지말고는 접근이 제한되며
다른 패키지에 있는 클래스일 경우 상속관계의 후손클래스 내에서만
부모클래스에 접근할 수 있습니다. (클래스 선언 불가)

(~)default : 같은 패키지말고는 접근이 제한되며 ,
접근 제한자가 없다면 자동으로 default접근 제한을 가지게 됩니다.

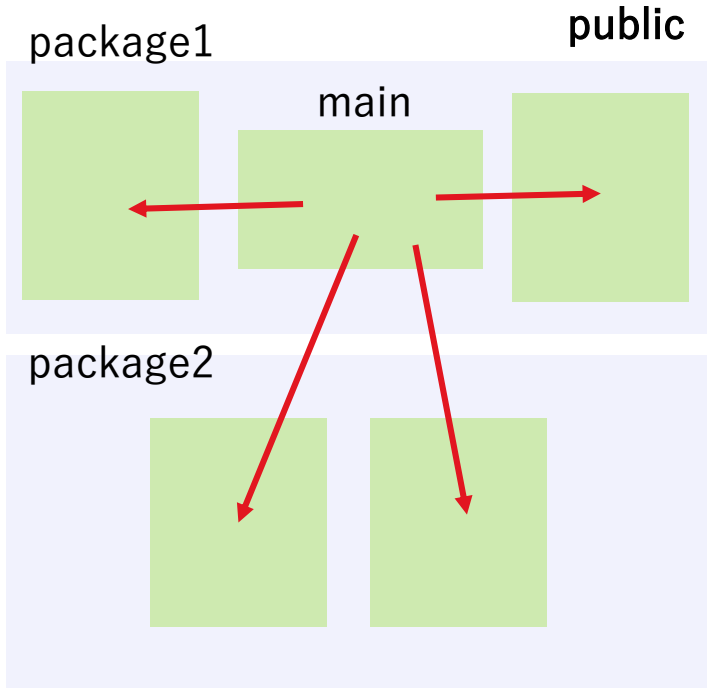
(+)public : 클래스와 패키지 안과 밖에 있는 모든 클래스들이 접근 가능합니다.

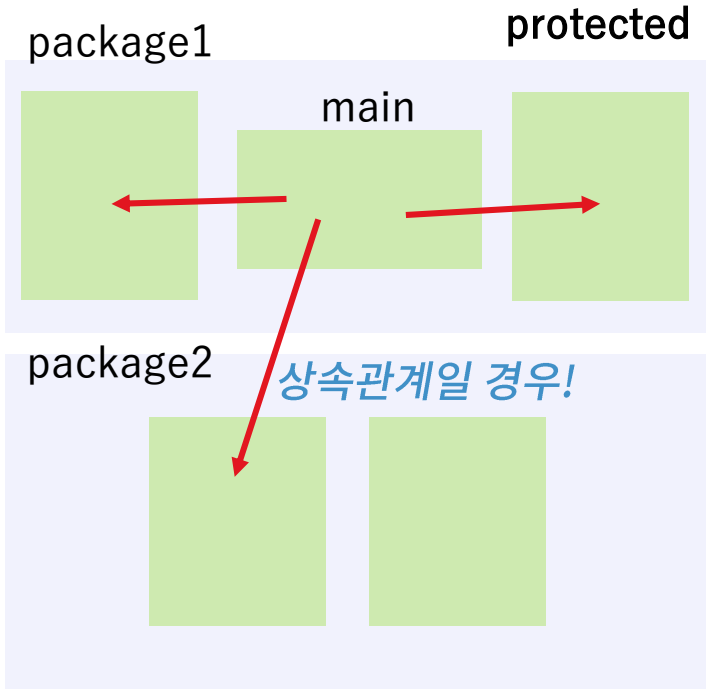
접근제어자의 공개범위는 **public >protected >default>private**로 **private**가 가장 낮고 public이 가장 높습니다.

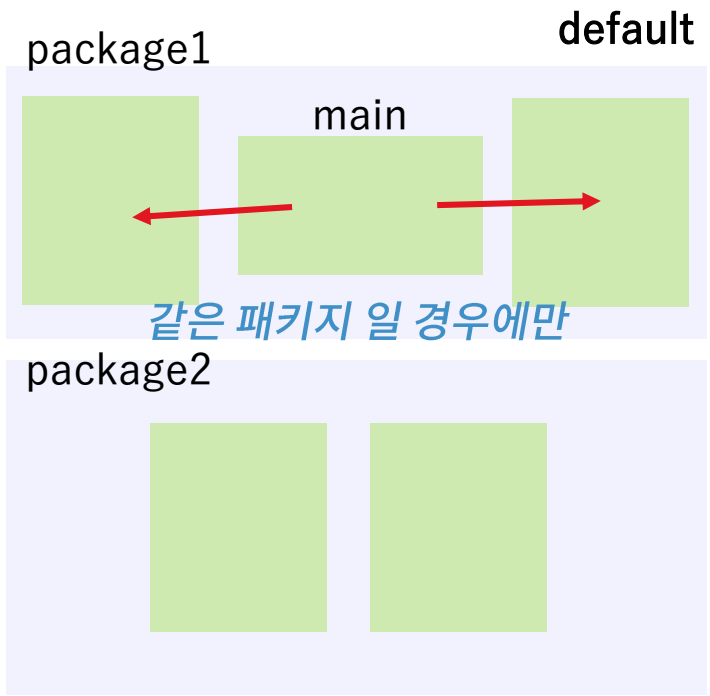
Java

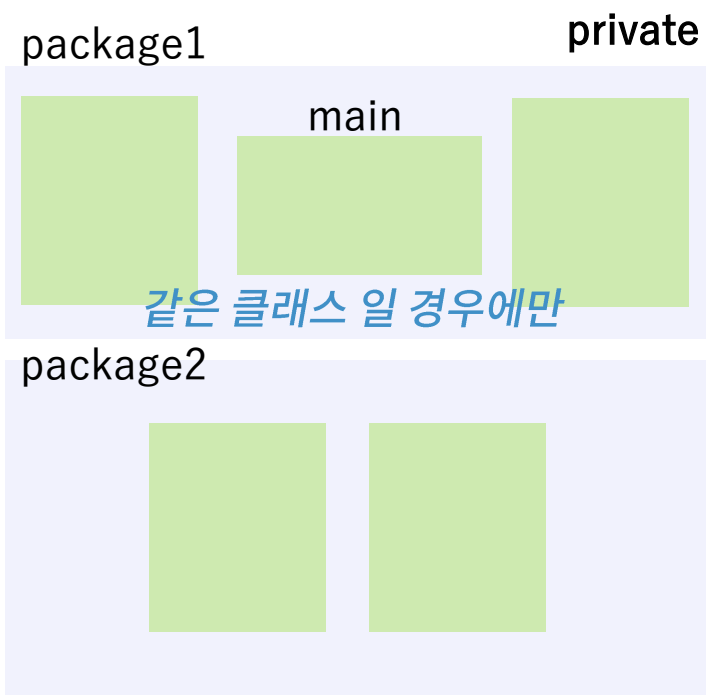
-

접근제한자


package1
public
main
package2


package1
protected
main
package2
상속관계일 경우!


package1
default
main
같은 패키지 일 경우에만
package2


package1
private
main
같은 클래스 일 경우에만
package2

-

접근제한자

< class접근 제한자 >

class의 접근제한자는 public과 default 2가지만 쓸 수 있습니다.
만약 class 앞에 public이 없다면 자동으로 default class가 되며, 같은 패키지 외에는
접근이 불가하게 됩니다.

< method접근 제한자 >

모든 접근제한자를 쓸 수 있습니다.

< 지역변수 접근 제한자 >

지역변수에는 접근 제한자를 쓸 수 없습니다.

< 필드 접근 제한자 >

필드는 객체의 정보(상태)를 나타내는 것이며,
클래스 안에서 선언되는 멤버 변수를 필드(field)라고 합니다.
필드를 선언할 때는 접근제한자->필드타입->필드변수명 순으로 선언해주며
모든 접근제어자가 올 수 있지만 정보은닉의 목적으로 private를 가장 많이 씁니다.

-

초기화

필드의 멤버 변수값을 변경하는 방법 중에 하나인 클래스 초기화 블럭은 *인스턴스 변수와 클래스 변수의 초기화 방법이 다릅니다.*

인스턴스 = {} 안에서 초기화 시키며 앞에 접근제한자나 예약어 아무 것도 쓰지 않음
객체 생성시마다 초기화

클래스 = 앞에 static이라는 예약어를 쓴 후 {} 안에 초기화 시키며 프로그램 시작 시 한 번만 초기화

클래스 변수
JVM기본값 ==>명시적 초기값 ==>클래스 초기화 블록 초기값

인스턴스 변수
JVM기본값 ==> 명시적 초기값 ==>인스턴스 초기화 블록 초기값 ==> 생성자를 통한 초기값

-

생성자

< 생성자 >

생성자는 객체의 생성시에만 호출되며 메모리 생성과 동시에 객체의 데이터를 초기화 하는 역할을 한다.

```java
package com.kh.object;

public class ClassS {

    private String name;
    private int age;

    public ClassS() {}

    public ClassS(String name, int age) {

        this.name = name;
        this.age = age;

    }

}
```

기본생성자 (디폴트 생성자)

매개변수 생성자

위의 그림처럼 생성자 명은 반환값이 없고 클래스의 이름과 동일해야 한다.

- 기본생성자는 객체가 생성될 때 가장먼저 호출되는 생성자이며 개발자가 명시하지 않아도 컴파일 시점에서 자동으로 생성된다.

- 매개변수 생성자는 기본생성자 외에 특정 목적에 의해서 개발자가 만든 생성자

(매개변수가 있는 생성자만 만들었을 시, 매개변수가 없는 생성자를 호출하면 에러발생)

-

오버로딩

< 오버로딩 >

오버로딩은 같은 클래스 내에서 동일한 이름의 메소드를 여러개 정의할 수 있는 기능을 한다.

```java
public ClassS() {}

public ClassS(String name) {
    this.name = name;
}

public ClassS(String name, int age) {

    this.name = name;
    this.age = age;

}
```

매개변수 갯수와 종류가 다름

- 매개변수의 값 따라 다양하게 처리를 해 줄 수 있다는 장점이 있다.

-

this

< this >

this란 클래스를 통해 만들어진 객체 자기 자신을 지칭하는 키워드이다.

```
private String name;

public ClassS() {}

public ClassS(String name) {

    this.name = name;

}
```

- 필드와 매개변수가 같은 이름일 경우 둘 다 매개변수로 인식하기 때문에 필드를 가리킬 때 this를 붙여 구분해준다

- **this()**란 같은 클래스에 정의된 생성자를 부를때 사용한다. (반드시 첫 번째 줄에 선언해야 한다.)

-

메소드

< 메소드 >

메소드(함수)란 어떤 기능을 수행하는 명령어들의 집합이다.

표현식 >> [접근제한자] [예약어] 반환형 메소드명 ([매개변수]) { 기능정의 }

```java
public String study(int age) {
    return name+age;
}
```

매개변수 O, 리턴값 O

```java
public String study() {
    return name;
}
```

매개변수 X, 리턴값 O

```java
public void study() {
    System.out.println(name);
}
```

매개변수 O, 리턴값 X

```java
public void study() {
    System.out.println(name);
}
```

매개변수 X, 리턴값 X

-

Getter
Setter
Return

< getter와 setter >

getter: private 멤버 변수값을 클래스 외부에서 가져가는 메소드를 말한다.

setter: 클래스 외부에서 private 멤버 변수값을 변경하는 메소드를 말한다.

getter/setter를 사용하는 이유

1. 외부에서 변수에 마음대로 접근해서 변경 및 조회 하는것을 막기 위해서
2. 세분화된 접근 제어 기능 (읽기전용, 쓰기전용, 읽기/쓰기 허용, 비허용)

< Return >

- 메소드 종료: 메소드의 종료를 알린다.
- 데이터 반환: 메소드에서 정의한 값을 반환한다.

-

객체배열

< 객체배열 >

기존의 기본자료형과 참조형(String)으로 선언된 배열과는 다르게 **객체를 배열에 저장하는 배열**

(배열의 각 index에는 할당된 객체들이 들어가 있으며, 각 객체들은 해당 객체를 가리키는 주소 값을 가지고 있음)

선언 : **객체명[] 배열명;** 또는 객체명 변수명[];

할당 : **배열명 = new 객체명[배열 크기];**

선언과 동시에 할당 : **객체명[] 배열명 = new 객체명[배열 크기];**

-

객체배열

member	Member[] member = new Member[3];		
	Member 객체	Member 객체	Member 객체

위의 표와 같이 선언 및 할당이 가능하다.

```
Member[] member = {
            new Member("홍길동", 46, 'M'),
            new Member("박말자", 62, 'F'),
            new Member("김철수", 38, 'M')
}
```

위의 표와 같이 선언 및 할당과 동시에 초기화가 가능하다.

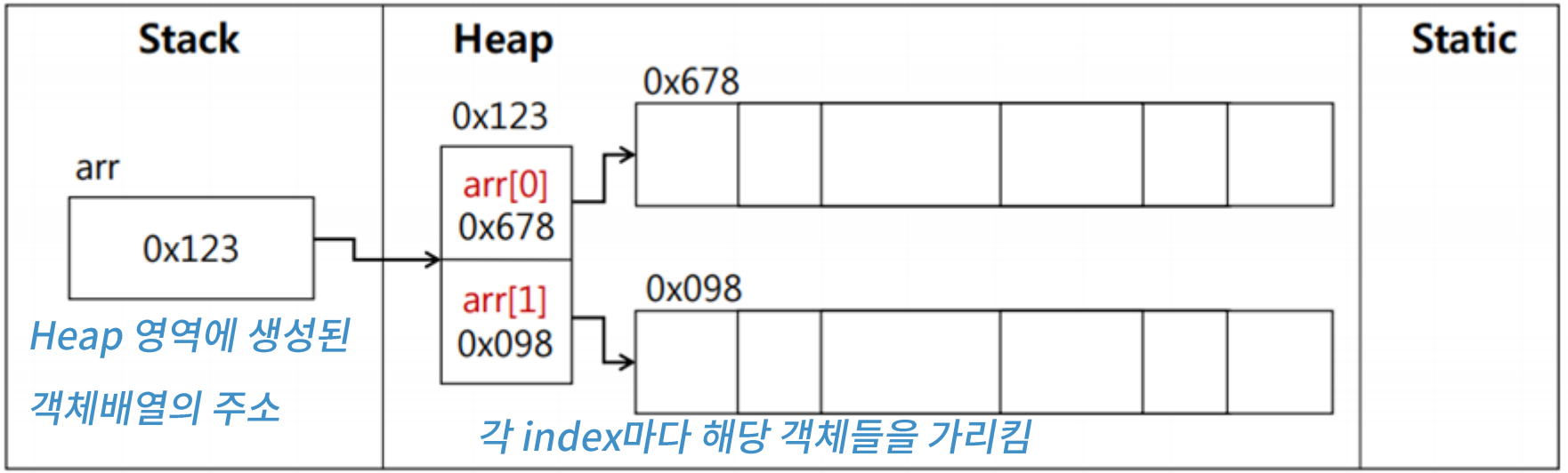
-

객체배열

메모리상에 저장되는 형태로는 아래 표와 같이 저장된다.

메모리 Heap영역에 생성된 객체배열을 메모리 Stack영역에 생성된 객체배열 변수가 가리키고 있으며,

Heap영역에 생성된 객체배열은 각 index마다 해당 객체들을 가리키고 있다.



-

상속

< 상속 >

다른 클래스가 가지고 있는 멤버(필드, 메소드)들을 새로 작성할 클래스에서 직접 만들지 않고 상속을 받음으로써 새 클래스가 자신의 멤버처럼 사용할 수 있는 기능이다.

클래스의 재사용과 연관된 일련의 클래스들에 대한 공통적인 규약에 대한 정의가 상속의 목적이라고 할 수 있다.

< 장점 >

1. 동일한 필드와 메소드의 구현이 적어지므로 보다 적은 양의 코드로 새로운 클래스 작성이 가능
2. 동일한 필드와 메소드 즉, 코드를 공통적으로 관리하기 때문에 코드의 추가 및 변경에 용이
3. 새로 추가할 필드와 메소드만 추가하여 사용하므로 프로그램의 생산성과 유지보수에 크게 기여

-

상속

클래스 간의 상속 시에는 extends 키워드를 사용하며, 인터페이스 간의 상속 시에도 extends 키워드를 사용한다.

단, 클래스가 인터페이스를 상속받을 때에는 implements 키워드를 사용한다.

```
public class School {
        private String name;
        private int age;
}

public School() {}
public School(String name, int age) {
        this.stuName = name;
        this.stuAge = age;
}
```

```
public class Student extends School {
        private String grade;
}

public Student() { super(); }      // super();는 생략 가능
public Student(String name, int age, String grade) {
        super(name, age);      // 부모의 생성자 호출 및 초기화
        this.grade = grade;
}
```

-

상속

< 단일상속과 다중상속 >

JAVA는 **단일 상속만 지원**한다.

그 이유로는 *다중 상속으로 인해 서로 다른 부모 클래스로부터 상속 받은 멤버 간의 변수명이 같은 경우* 에 **문제가 발생하기 때문이다.** 그러므로 단일 상속으로 인해 클래스간의 관계가 다중 상속보다 명확하고 신뢰성 있는 코드를 작성할 수 있다.

-

Super()와 Super

< super >

부모 클래스로부터 상속받은 필드나 메소드를 자식 클래스에서 참조하는데 사용하는 참조 변수로,
자식 클래스 내에서 부모 클래스 객체에 접근하여 필드나 메소드 호출 시에 사용

< super() >

부모 객체의 생성자를 호출할 때 사용한다.

상속받은 후손 클래스에서 부모 클래스의 필드에 접근하려면, 부모 객체의 생성자를 통해 접근하거나, getter()메소드를 통해 가져와야 하는데, **후손 객체의 생성자 첫 줄에 super()메소드로 부모 객체의 생성자를 호출하여 부모 객체의 필드값을 초기화**할 수 있다.

첫 줄에 작성하는 이유는 부모 객체의 생성자가 가장 먼저 실행되어야 하기 때문이다. **매개변수가 있는 부모 객체의 생성자의 경우, super(매개변수, 매개변수, ...)로 작성**하면 된다.

-

오버라이딩

< 오버라이딩 >

자식 클래스가 상속 받은 부모 클래스의 메소드를 재작성하여 사용하는 것으로, 부모가 제공하는 메소드의 기능을 후손이 일부분 고쳐 사용하겠다는 의미이다.

자식 객체를 통해 실행이 될 경우, 자식의 메소드가 부모의 메소드보다 우선권을 가지게 된다.

오버라이딩을 할 경우, Annotation마크 즉, @Override를 메소드 위에 적어주어야 한다.

접근제어자는 부모 메소드와 같거나 넓은 범위로 변경이 가능하다.

-

오버라이딩
오버로딩

< 오버라이딩 VS 오버로딩 >

오버라이딩(Overriding)	오버로딩(Overloading)
하위 클래스에서 메소드 정의 (부모의 메소드를 가져와 재정의)	같은 클래스에서 메소드 정의 (동일한 기능을 수행하는 매소드, 매개변수가 다름)
메소드 이름 동일 매개변수의 개수와 타입이 동일 리턴 타입 동일 private 메소드는 오버라이딩 불가	메소드 이름 동일 매개변수의 개수와 타입이 다름 리턴 타입 상관 없음
자식의 메소드의 접근 범위가 부모 메소드의 접근 범위보다 넓거나 같아야 함	각 메소드의 접근 제어자는 상관 없음
부모의 메소드를 자식 메소드가 재정의하기 때문에 자식 메소드의 예외 수가 부모 메소드의 예외 수보다 적거나 범위가 좁아야 함	예외처리와 상관 없음

-

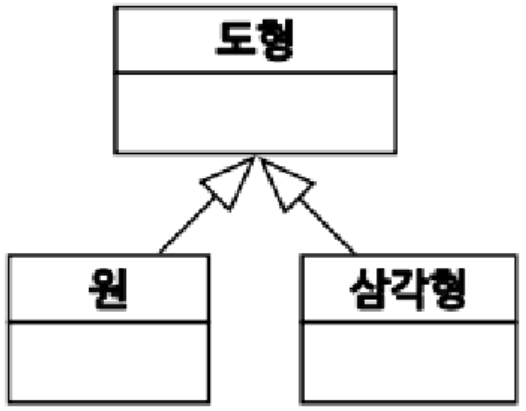
다형성

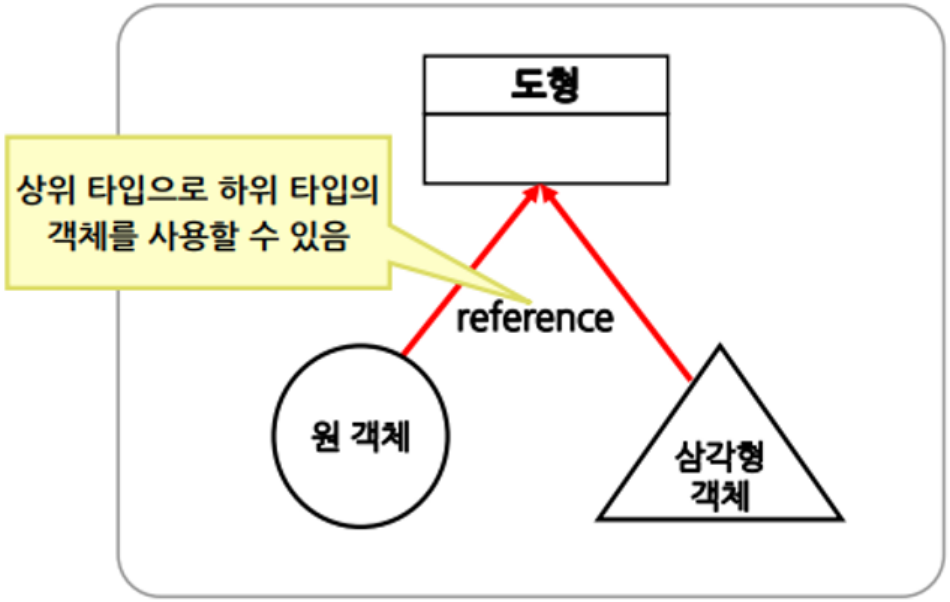
< 다형성>

다형성은 객체 지향 프로그래밍을 구성하는 중요한 특징 중 하나이다.

다형성(polymorphism)이란 하나의 객체가 여러 가지 타입을 가질 수 있는 것을 의미한다.

다형성을 통해 같은 자료형에 여러 가지 객체를 대입하여 다양한 결과를 얻어낼 수 있다.





-

업캐스팅과
다운캐스팅

< 업캐스팅 >

다형성을 통해 자바에서는 부모 클래스 타입의 참조 변수로 자식 클래스 타입의 인스턴스를 참조할 수 있다.

이러한 경우를 **업캐스팅(Up Casting)** 이라 한다.

부모클래스 변수 = new 자식클래스( );

Ex) 부모 클래스가 Parent이고 이를 상속받는 자식 클래스가 Child라고 할 때,

```
Parent parent = new Child();
```

부모 클래스 *자식 클래스*

부모타입인 Parent 클래스의 참조형 변수가 자식 타입인 Child의 객체주소를 받을 수 있는데 이것이 업캐스팅.

단, 자식 객체의 주소를 전달받은 부모 참조변수, 즉 변수 parent를 통해서 접근할 수 있는 정보는 원래 부모 타입이었던 parent의 멤버만 참조 가능

-

업캐스팅과
다운캐스팅

< 다운캐스팅 >

자식 객체의 주소를 받은 부모 참조형 변수를 가지고 자식의 멤버를 참조해야 할 경우, 자식클래스 타입으로 명시적으로 형변환 해주어야 한다.
이러한 경우를 다운 캐스팅(Down Casting) 이라 한다.

Ex) 부모 클래스가 Parent이고 이를 상속받는 자식 클래스가 Child라고 할 때,

```
Parent parent = new Child();
```
부모 클래스 *자식 클래스*

부모 참조변수인 parent로 자식클래스 Child의 멤버를 참조하고 싶다면

```
Child child = (Child)parent;
```
형변환

다음과 같이 자식클래스로 명시적 형변환을 해주어야 한다. 이것이 **다운캐스팅.**

-

instanceof

< instanceof >

업캐스팅된 객체가 어떤 본래 타입인지 알아보기 위해서는 instanceof를 사용한다.
Instanceof 는 참조변수가 참조하고 있는 인스턴스의 실제 타입을 알아볼 수 있다.
instanceof의 **왼쪽에는 참조변수**를, **오른쪽에는 타입(클래스명)**이 피연산자로 위치한다.
그리고 연산의 결과로 boolean값인 true, false 중의 하나를 반환 한다.

```
If (A instanceof B){
        …
}
```

참조변수 A가 B타입이라면 **true**
아니라면 **false** 반환

-

다형성의
이용

< 다형성의 이용 >

다형성을 이용하여 상속 관계에 있는 하나의 부모 클래스 타입의 배열 공간에 여러 종류의 자식 클래스 객체 저장 가능하다.

다형성을 이용하여 메소드 호출 시 부모타입의 변수 하나만 사용해 자식 타입의 객체를 받을 수 있음

```
Car[] carArr = new Car[5];
```

부모클래스 배열공간

```
carArr[0] = new Sonata();
carArr[1] = new Avante();
carArr[2] = new Grandure();
carArr[3] = new Spark();
carArr[4] = new Morning();
```

여러 종류의 자식클래스 객체저장 가능

```
public void execute() {
        driveCar(new Sonata());
        driveCar(new Avante());
        driveCar(new Grandure());
}
```

여러 종류의 자식타입 인스턴스

부모타입의 매개변수

```
public void driveCar(Car c) {}
```

-

추상메소드와
추상클래스

< 추상메소드와 추상클래스 >

추상메소드	추상클래스
몸체가 없는 메소드이다. 즉, 구현하는 { } 부분이 없다. 추상 메소드의 선언부에 abstract 키워드 사용 상속시 반드시 구현 즉, 오버라이딩으로 강제화	몸체가 없는 메소드라 불리는 추상메소드를 포함한 클래스를 뜻한다. (미완성 클래스) 추상클래스일 경우 클래스 선언부에 abstract 키워드 사용 추상클래스는 인스턴스를 생성할 수 없음.

```
    추상클래스
abstract class class_Name {
    ...
    abstract return_Type methodName();
    ...       추상메소드
}
```

-

인터페이스

< 인터페이스 >

상수형 필드와 추상 메소드만을 작성할 수 있는 추상 클래스의 변형체이다.
메소드의 통일성을 부여하기 위해 추상 메소드만 따로 모아놓은 것으로 상속 시 인터페이스 내에 정의된 모든 추상메소드 구현해야 한다.

```
접근제어자 interface interface_Name {  인터페이스 선언
    public static final type constance_name = value;
    ...   상수필드
    public abstract method_name(…);
    ...   추상 메소드
}
```

인터페이스는 추상 클래스와 마찬가지로 자신이 직접 인스턴스를 생성할 수는 없다.
따라서 다음과 같이 인터페이스가 포함하고 있는 추상 메소드를 구현해 줄 클래스를 작성해야 한다.

```
class class_Name implements interface_Name { ... }
```

-

추상클래스VS
인터페이스

< 인터페이스 VS 추상클래스 >

인터페이스	추상클래스
구현 객체의 같은 동작을 보장하기 위해	상속을 받아 기능을 이용 & 확장하기 위해
다중상속	단일상속
implements	extends
모든 메소드는 추상메소드 + 상수필드	추상메소드 0개 이상 + 일반변수 + 일반메소드
객체생성 불가	객체생성 불가