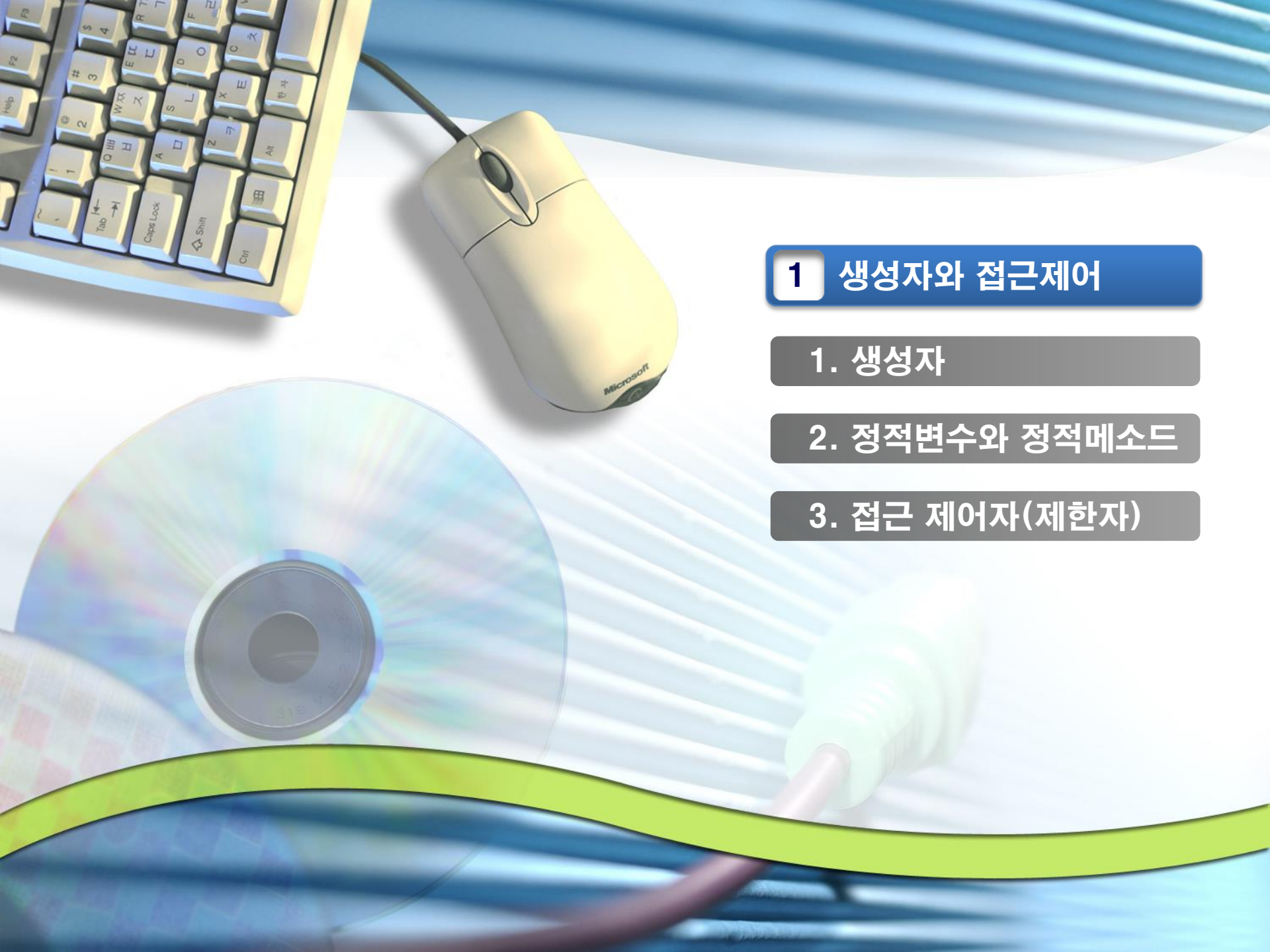
JAVA

생성자와 접근 제어



1 생성자와 접근제어

1. 생성자

2. 정적변수와 정적메소드

3. 접근 제어자(제한자)

생성자

일반적으로 객체가 생성된 후에 가장 먼저 하는 작업은 객체의 데이터를 초기값으로 설정하는 것이다.

자바에서는 이러한 작업을 위하여 생성자라는 특수한 메소드를 도입하였다. 생성자(constructor)는 클래스 안에 선언되어서 객체가 생성될 때에 필드에게 초기값을 제공하는 메소드이다.

생성자는 일반 메소드와 아주 흡사하다.

다만 메소드 이름이 클래스 이름과 같고 반환값을 가지지 않는다는 점만 다르다.

생성자(Constructor)

❖생성자

- new 연산자에 의해 호출되어 객체의 초기화 담당

```
new 클래스();
```

  - 필드의 값 설정
  - 메소드 호출해 객체를 사용할 수 있도록 준비하는 역할 수행

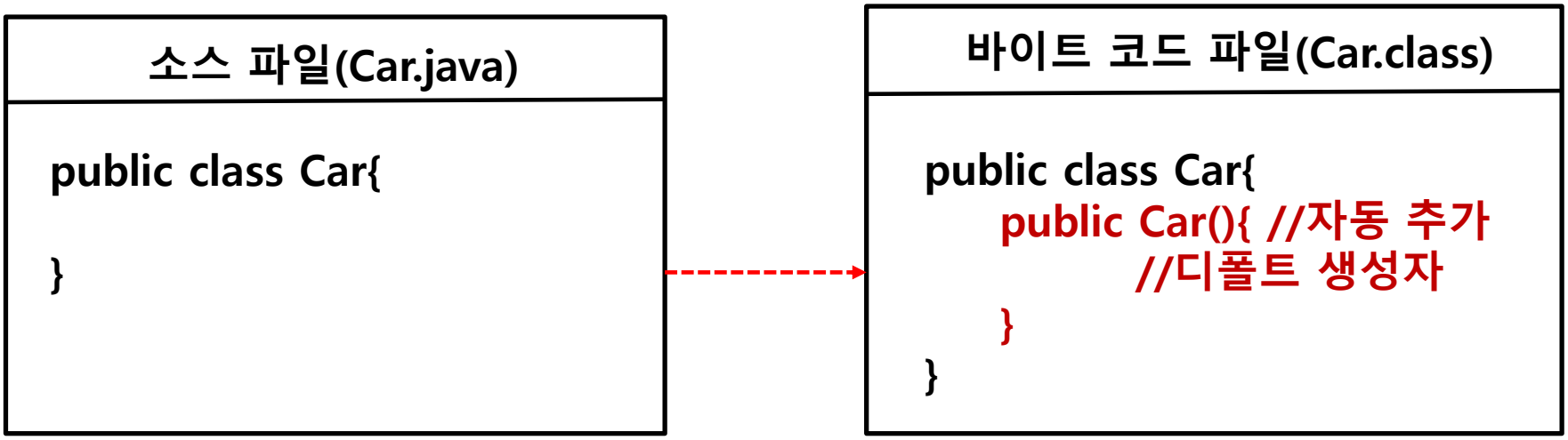
❖기본 생성자(Default Constructor)

- 모든 클래스는 생성자가 반드시 존재하며 하나 이상 가질 수 있음
- 생성자 선언을 생략하면 컴파일러는 다음과 같은 기본 생성자 추가

```
public 클래스() { }
```

디폴트 생성자

만약 클래스 작성시에 생성자를 하나도 만들지 않는 경우에는 자동적으로 메소드의 몸체 부분이 비어있는 생성자가 만들어진다. 이를 디폴트 생성자(default constructor)라고 한다. 디폴트 생성자는 몸체가 비어 있기 때문에 실행 결과에 아무런 영향을 끼치지 않는다.




```
Car myCar = new Car();
```

디폴트 생성자

```
class Car {
        private int speed; // 속도
        private int gear; // 기어
        private String color; // 색상
}

public class Car {
        public static void main(String[] args) {
           Car c = new Car();
        }
}
```

```
//디폴트 생성자
public Car(){ }
```

생성자(Constructor)

❖생성자 선언

- 디폴트 생성자 대신 개발자가 직접 선언

```
접근 제어자 클래스(매개변수, 매개변수...){
   // 필드의 초기화 코드
}
```

- 개발자 선언한 생성자 존재 시 컴파일러는 기본 생성자 추가하지 않음
  - new 연산자로 객체 생성시 개발자가 선언한 생성자 반드시 사용

```
public class Car{
        // 생성자
        public Car(String model, String color, int maxSpeed){
            ....
        }
}
```

```
Car myCar = new Car("그랜저", "검정", 300);
```

디폴트 생성자

```
class Car {
        private int speed; // 속도
        private int mileage; // 주행 거리
        private String color; // 색상
        public Car(int s, int m, String c) {
                speed = s;
                mileage = m;
                color = c;
        }
}
public class CarTest{
    public static void main(String args[]) {
        Car c1 = new Car (); //오류
        Car c2 = new Car (1,10,"red");
        }
}
```

생성자(Constructor)

❖필드 초기화

- 초기값 없이 선언된 필드는 객체가 생성될 때 기본값으로 자동 설정

- 다른 값으로 필드 초기화하는 방법
  - 필드 선언할 때 초기값 설정
  - 생성자의 매개값으로 초기값 설정

```
Address a1 = new Address(“홍길동”, “010-3425-7854”);
Address a2 = new Address(“김철수”, “010-8624-1364”);
```

  - 매개 변수와 필드명 같은 경우 this 사용

생성자(Constructor)

❖생성자 오버로딩(Overloading)

- 매개변수의 타입, 개수, 순서가 다른 생성자 여러 개 선언

```
public class 클래스 {
        public 클래스([타입 매개변수, ...]){
          ...
        }

        public 클래스([타입 매개변수, ...]){
          ...
        }
}
```

[생성자 오버로딩]
매개변수의 타입, 개수, 순서가 다르게 선언

생성자(Constructor)

```
class Car {
        private int speed; // 속도
        private String model; // 모델명
        private String color; // 색상
        public Car() { .. }
        public Car(int speed) { .. }
        public Car(int speed, String model) { .. }
        public Car(int speed, String model, String color) { .. }
}
```

```
public Car(String model, String color) { .. }
public Car(String color, String model) { .. } // 오버로딩이 아님.
```

```
Car car1 = new Car();
Car car2 = new Car(100);
Car car3 = new Car(100, "그랜저");
Car car4 = new Car(100, "그랜저", "검정색");
```

예제 4-4 : 생성자 선언 및 활용 연습

제목과 저자를 나타내는 title과 author 필드를 가진 Book 클래스를 작성하고, 생성자를 작성하여 필드를 초기화하라.

```
public class Book {
   String title;
   String author;

   public Book(String t) { // 생성자
      title = t; author = "작자미상";
   }

   public Book(String t, String a) { // 생성자
      title = t; author = a;
   }

   public static void main(String [] args) {
      Book littlePrince = new Book("어린왕자", "생텍쥐페리");
      Book loveStory = new Book("춘향전");
      System.out.println(littlePrince.title + " " +
littlePrince.author);
      System.out.println(loveStory.title + " " + loveStory.author);
   }
}
```

```
어린왕자 생텍쥐페리
춘향전 작자미상
```

생성자(Constructor)

❖다른 생성자 호출(this())

- 생성자 오버로딩되면 생성자 간의 중복된 코드 발생
- 초기화 내용이 비슷한 생성자들에서 이러한 현상을 많이 볼 수 있음
  - 초기화 내용을 한 생성자에 몰아 작성
  - 다른 생성자는 초기화 내용을 작성한 생성자를 this(...)로 호출

```
public Car() {
    this.speed = 10;
    this.model = "그랜저";
    this.color = "검정색";
}

public Car(String model) {
    this.speed = 10;
    this.model = model;
    this.color = "은색";
}

public Car(int speed, String model, String color) {
    this.speed = speed;
    this.model = model;
    this.color = color;
}
```

생성자(Constructor)

❖다른 생성자 호출(this())

- 생성자 오버로딩되면 생성자 간의 중복된 코드 발생
- 초기화 내용이 비슷한 생성자들에서 이러한 현상을 많이 볼 수 있음
  - 초기화 내용을 한 생성자에 몰아 작성
  - 다른 생성자는 초기화 내용을 작성한 생성자를 this(...)로 호출

```
public Car() {
    this(10, "그랜저", "검정색")
}

public Car(String model) {
    this(10, model, "은색")
}

public Car(int speed, String model, String color) {
    this.speed = speed;
    this.model = model;
    this.color = color;
}
```

this() 메서드

```
class Car {
        private int speed; // 속도
        private int mileage; // 주행 거리
        private String color; // 색상
        public Car(int s, int m, String c) {
                speed = s;
                mileage = m;
                color = c;
        }
        public Car() {
                this(100, 40,  " red");  //첫 번째 생성자를 호출한다.
        }
}
```

생성자가 다른 생성자를 호출하여 초기화할 경우 this() 라는 예약어를 사용하여야 한다. 메서드를 사용한다.

this()로 다른 생성자 호출

예제 4-4에서 작성한 Book 클래스의 생성자를 this()를 이용하여 수정하라.

```
public class Book {
    String title;
    String author;
    void show() { System.out.println(title + " " + author); }

    public Book() {
        this("", "");
        System.out.println("생성자 호출됨");
    }

    public Book(String title) {
        this(title, "작자미상");
    }

    public Book(String title, String author) {
        this.title = title; this.author = author;
    }
    public static void main(String [] args) {
        Book littlePrince = new Book("어린왕자", "생텍쥐페리");
        Book loveStory = new Book("춘향전");
        Book emptyBook = new Book();
        loveStory.show();
    }
}
```

title = " 춘향전"
author = "작자미상"

생성자 호출됨
춘향전 작자미상

this 키워드

모든 객체는 키워드 this를 사용하여서 자기 자신을 참조할 수 있다. this는 특히 필드 이름과 메소드 매개변수의 이름이 동일한 경우, 구분하기 위하여 자주 사용된다.

```
public void setSpeed(int speed) {
        this.speed = speed;
}
```

필드 speed와 매개변수 speed를 구별하기 위하여 this 사용

- 객체(인스턴스) 자신의 참조(번지)를 가지고 있는 키워드
- 객체 내부에서 인스턴스 멤버임을 명확히 하기 위해 this. 사용

중간점검

1. 만약 클래스 이름이 MyClass라면 생성자의 이름은 무엇이어야 하는가?
2. 생성자의 반환형은 무엇인가?
3. 생성자 안에서 this()의 의미는 무엇인가?

일반적 구조

클래스

객체를 생성하기 위한 틀

클래스 선언

class 클래스명

클래스 정의

생성자에서 하는 일은 주로 멤버변수의 초기화와 메모리 할당을 위해 사용된다

필드

객체의 속성, 특징, 특성을 정의

접근 제어자 자료형 변수명;

생성자

- 클래스의 이름과 동일.
- 반환형을 가지지 않는다.

메서드

객체의 기능을 정의

접근 제어자 제어자반환형 메서드(매개변수){

...

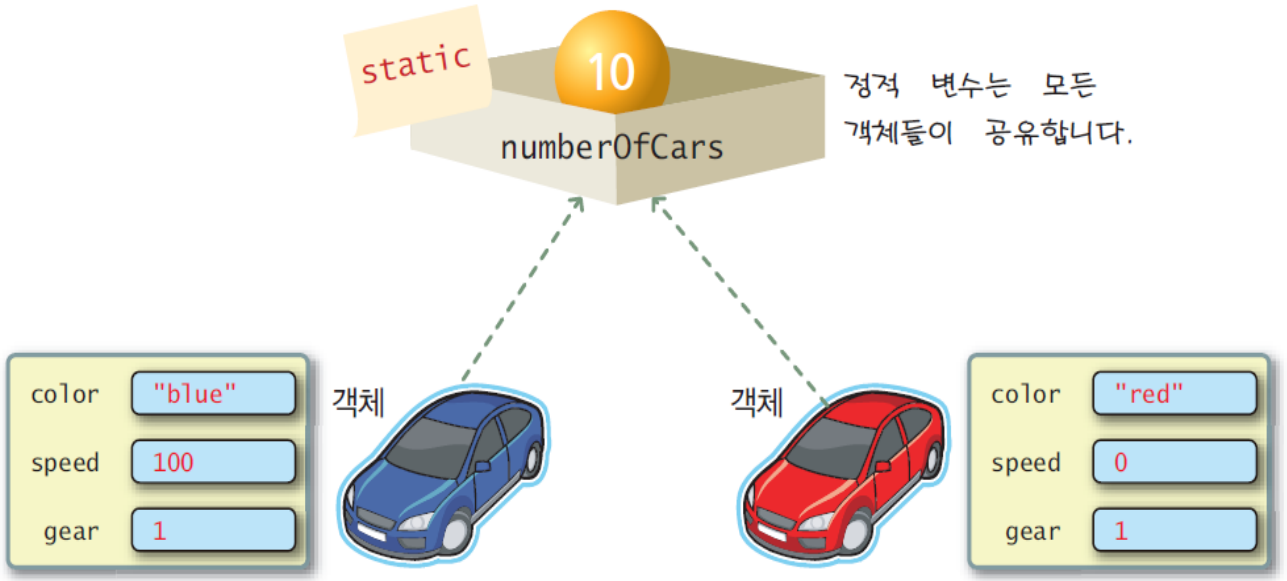
}

- void
- 자료형

정적 변수와 정적 메소드

1) 정적 변수(static variable)

같은 클래스 설계도를 이용하여 많은 객체들이 생성될 때 각각의 객체(인스턴스)들은 자신만의 필드들을 가진다. 이들 필드는 인스턴스마다 별도로 생성되기 때문에 인스턴스 변수(instance variable)라고도 한다.



정적 변수와 정적 메소드

모든 객체에 공통인 변수가 필요한 경우도 있다. 이것이 정적 변수(static variable)이다. 정적 변수는 클래스 변수(class variable)라고도 한다. 정적 변수는 하나의 클래스에 하나만 존재한다.

즉 정적 변수는 그 클래스의 모든 객체들에 의하여 공유된다. 정적 변수를 만들려면 변수를 정의할 때 맨 앞에 static 키워드를 붙이면 된다.

정적 변수와 정적 메소드

new 명령문으로 인스턴스를 생성하면 해당 클래스에 포함된 필드가 힙 메모리에 할당된다. 만일 어떤 클래스의 인스턴스를 여러 개 생성한다면 각 인스턴스가 독립적으로 힙 메모리에 할당된다. 그런데 때로는 어떤 클래스로 만든 인스턴스가 모두 공유하는 멤버가 필요할 때가 있다.

클래스의 모든 인스턴스가 공유하는 멤버를 정적 멤버(static member)라고 한다. 정적멤버는 클래스의 인스턴스의 특정 인스턴스만 사용하는 게 아니라 **해당 클래스로 생성된 모든 인스턴스가 공유하는 멤버이다.** 자바에서는 필드나 메서드를 공유 멤버로 선언할 수 있으며 이때 **static** 키워드를 사용한다.

정적 멤버와 static

❖정적(static) 멤버란?

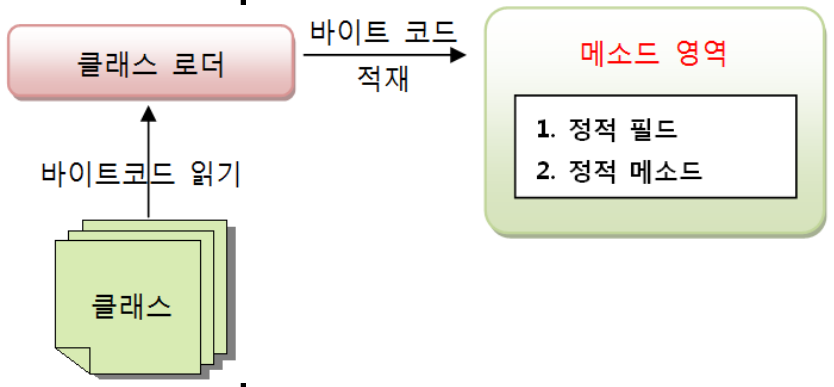
- 클래스에 고정된 필드와 메소드 - 정적 필드, 정적 메소드

- 정적 멤버는 클래스에 소속된 멤버
  - 객체 내부에 존재하지 않고, 메소드 영역에 존재
  - 정적 멤버는 객체를 생성하지 않고 클래스로 바로 접근해 사용

❖정적 멤버 선언

- 필드 또는 메소드 선언할 때 **static** 키워드 붙임

```
public class 클래스 {
    // 정적 필드
    static 자료형 필드;

    // 정적 메서드
    public static 리턴타입 메서드(매개변수
}
```



인스턴스 변수와 정적 변수

인스턴스 변수와 정적 변수를 정리하면 다음과 같다.

- 인스턴스 변수(instance variable): 객체마다 가지고 있어야 할 데이터
- 정적 변수(static variable): 공용적인 데이터

```
public class Calculator{
    String color;                       // 계산기 색상
    static double PI = 3.141592;        // 계산기에서 사용하는 파이값
}
```

구분	메모리 할당 시점	메모리 할당 위치	메모리 해제 시점
인스턴스 필드	인스턴스 생성시	힙 메모리	인스턴스 소멸시
클래스 필드	프로그램 시작시	코드 메모리	프로그램 종료시

정적변수

```
class Car {
        private int speed;
        private int gear;
        private String color;
        // 실체화된 Car 객체의 개수를 위한 정적 변수
        private static int numberOfCars = 0;
        public Car(int s, int g, String c) {
            speed = s;
            gear = g;
            color = c;
            ++numberOfCars;
        }
}
```

정적 변수

정적변수

위의 코드에서 numberOfCars가 바로 정적 변수이다. 정적 변수는 그 클래스의 인스턴스를 만들지 않고서도 사용될 수 있다. 객체가 없으므로 클래스 이름을 객체처럼 사용하여 접근한다.

- 형식

```
클래스이름.정적변수
```

정적 멤버(변수)는 프로그램이 시작할 때 딱 한번 메모리의 메서드 영역에 할당되며 프로그램이 종료될 때가지 유지되는 특징이 있다.

정적 메소드

2) 정적 메소드(static method)

정적 메소드는 static 수식자를 메소드 선언에 붙이며 클래스 이름을 통하여 호출되어야 한다. 정적 메소드는 객체를 생성하지 않고 사용할 수 있는 메소드이다.

정적 메소드는 클래스 메소드(class method)라고도 한다.

- 형식

```
클래스이름.메서드명()
```

```
double value = Math.random() ;
int result = Math.max(3, 7);
```

정적 멤버와 static

❖정적 멤버 사용

- 클래스 이름과 함께 도트(.) 연산자로 접근

```
public class Calculator{
    public static final double PI = 3.141592;
    public static int plus(int x, int y) { return x + y; }
    public static int minus(int x, int y) { return x - y; }
}
```

[바람직한 사용]

```
double result = 5* 5 * Calculator.PI;
int  result2 = Calculator.plus(3, 9);
int result3 =  Calculator.minus(7, 2);
```

[바람직하지 못한 사용]

```
Calculator c = new Calculator();
double result = 5* 5 * c.PI;
int  result2 = c.plus(3, 9);
int result3 =  c.minus(7, 2);
```

정적 메소드

우리가 많이 사용하였던 main()메소드의 앞에도 static이 붙어 있다. JVM에서 객체를 생성할 필요가 없이 main() 메소드를 호출하여 실행할 수 있도록 하기 위해서이다.

정적 메소드는 객체가 생성되지 않는 상태에서 호출되는 메소드이다.

- 인스턴스 변수와 인스턴스 메소드는 사용할 수 없다.
- 정적 변수와 지역 변수만을 사용할 수 있다.
- 정적 메소드에서 정적 메소드를 호출하는 것은 가능하다.
- 정적 메소드는 this 키워드를 사용할 수 없다.

  왜냐하면 this가 참조할 인스턴스가 없기 때문이다.

정적 메소드

```
public class Test {
	public static void main(String args[]) {
		add(10, 20);
	}

	public int add(int x, int y) {
		return x+y;
	}
}
```

정적 메소드

```
public class Test {
	public static void main(String args[]) {
		add(10, 20);
	}

	public static int add(int x, int y) {
		return x+y;
	}
}
```

static 멤버와 non-static 멤버

non-static 멤버의 특성

- 공간적 특성 – 멤버들은 객체마다 독립적으로 별도 존재
  - 인스턴스 멤버라고도 부름
- 시간적 특성 – 필드와 메소드는 객체 생성 후 비로소 사용 가능
- 비공유 특성 - 멤버들은 다른 객체에 의해 공유되지 않고 배타적

static 멤버란?

- 객체마다 생기는 것이 아님
- 클래스당 하나만 생성됨
  - 클래스 멤버라고도 부름
- 객체를 생성하지 않고 사용가능
- 특성
  - 공간적 특성 - static 멤버들은 클래스 당 하나만 생성
  - 시간적 특성 - static 멤버들은 클래스가 로딩될 때 공간 할당.
  - 공유의 특성 - static 멤버들은 동일한 클래스의 모든 객체에 의해 공유

```
class StaticSample {
   int n;                  // non-static 필드
   void g() {...}          // non-static 메소드

   static int m;           // static 필드
   static void f() {...} // static 메소드
}
```

non-static 멤버와 static 멤버의 차이

	non-static 멤버	static 멤버

선언	class Sample {
	  int n;
	  void g() {…}
	}
		class Sample {
		  static int m;
		  static void g() {…}
		}

공간적 특성	멤버는 객체마다 별도 존재 • 인스턴스 멤버라고 부름	멤버는 클래스당 하나 생성 • 멤버는 객체 내부가 아닌 별도의 공간(클래스 코드가 적재되는 메모리)에 생성 • 클래스 멤버라고 부름
시간적 특성	객체 생성 시에 멤버 생성됨 • 객체가 생길 때 멤버도 생성 • 객체 생성 후 멤버 사용 가능 • 객체가 사라지면 멤버도 사라짐	클래스 로딩 시에 멤버 생성 • 객체가 생기기 전에 이미 생성 • 객체가 생기기 전에도 사용 가능 • 객체가 사라져도 멤버는 사라지지 않음 • 멤버는 프로그램이 종료될 때 사라짐
공유의 특성	공유되지 않음 • 멤버는 객체 내에 각각 공간 유지	동일한 클래스의 모든 객체들에 의해 공유됨

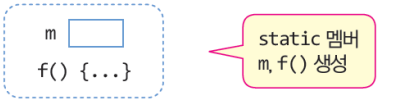
static 멤버를 객체의 멤버로 접근하는 사례

```java
class StaticSample {
    public int n;
    public void g() {
        m = 20;
    }
    public void h() {
        m = 30;
    }
    public static int m;
    public static void f() {
        m = 5;
    }
}

public class Ex {
    public static void main(String[] args) {
        StaticSample s1, s2;
        s1 = new StaticSample();
        s1.n = 5;
        s1.g();
        s1.m = 50; // static
        s2 = new StaticSample();
        s2.n = 8;
        s2.h();
        s2.f(); // static
        System.out.println(s1.m);
    }
}
```

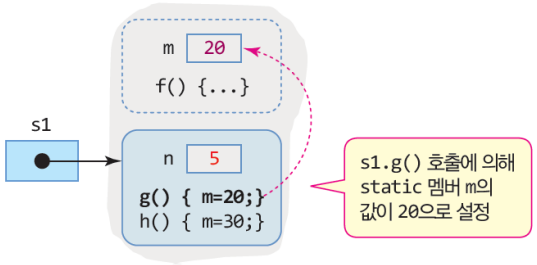
실행 결과

```
5
```

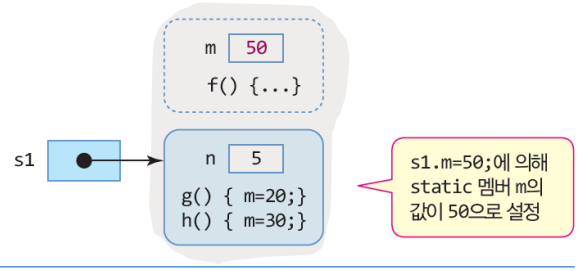
```
StaticSample s1, s2;
```



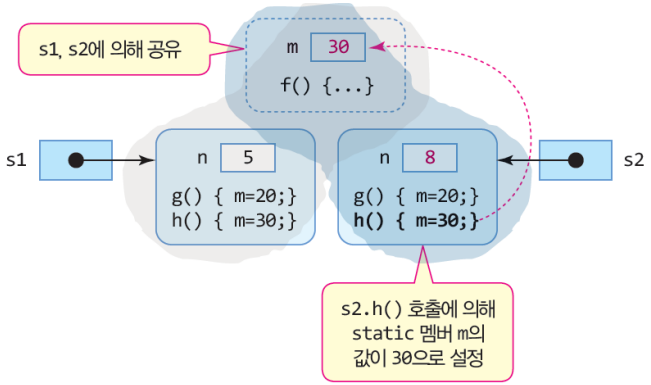

```
s1 = new StaticSample();
s1.n = 5;
s1.g();
```



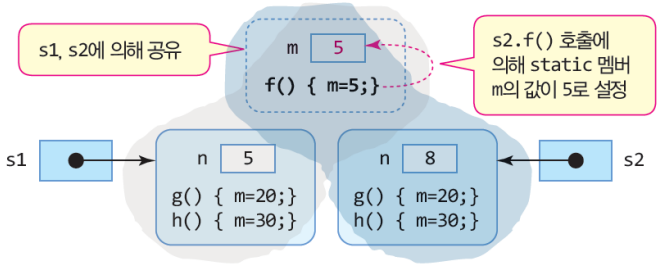

```
s1.m = 50;
```




```
s2 = new StaticSample();
s2.n = 8;
s2.h();
```




```
s2.f();
```




```
System.out.println(s1.m);
```

5 출력

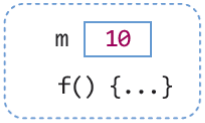
static 멤버를 클래스 이름으로 접근하는 사례

```
class StaticSample {
    public int n;
    public void g() {
        m = 20;
    }
    public void h() {
        m = 30;
    }
    public static int m;
    public static void f() {
        m = 5;
    }
}

public class Ex {
    public static void main(String[] args) {
        StaticSample.m = 10;

        StaticSample s1;
        s1 = new StaticSample();
        System.out.println(s1.m);
        s1.f();
        StaticSample.f();
    }
}
```

실행 결과

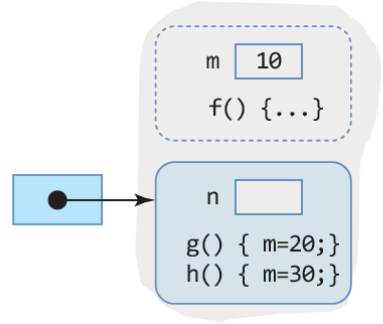
```
10
```

```
StaticSample.m = 10;
```



static 멤버 생성

```
StaticSample s1;
s1 = new StaticSample();
```

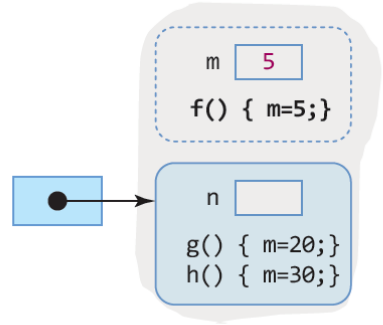


객체 s1 생성

```
System.out.println(s1.m);
```

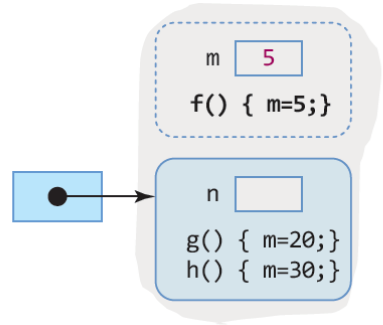
10 출력

```
s1.f();
```



s1.f() 호출에 의해 static 멤버 m의 값이 5로 변경

```
StaticSample.f();
```



StaticSample.f() 호출에 의해 static 멤버 m의 값이 5로 변경

static의 활용

1. 전역 변수와 전역 함수를 만들 때 활용

- 전역변수나 전역 함수는 static으로 클래스에 작성
- static 멤버를 가진 클래스 사례
  - Math 클래스 : java.lang.Math
    - 모든 필드와 메소드가 public static으로 선언
    - 다른 모든 클래스에서 사용할 수 있음

```
public class Math {
   public static int abs(int a);
   public static double cos(double a);
   public static int max(int a, int b);
   public static double random();
   ...
}
```

```
// 잘못된 사용법
Math m = new Math(); // Math() 생성자는 private
int n = m.abs(-5);

// 바른 사용법
int n = Math.abs(-5);
```

2. 공유 멤버를 작성할 때

- static 필드나 메소드는 하나만 생성. 클래스의 객체들 공유

static 멤버를 가진 Calc 클래스 작성

전역 함수로 작성하고자 하는 abs, max, min의 3개 함수를 static 메소드로 작성하고 호출하는 사례를 보여라.

```
class Calc {
   public static int abs(int a) { return a>0?a:-a; }
   public static int max(int a, int b) { return (a>b)?a:b; }
   public static int min(int a, int b) { return (a>b)?b:a; }
}

public class CalcEx {
   public static void main(String[] args) {
      System.out.println(Calc.abs(-5));
      System.out.println(Calc.max(10, 8));
      System.out.println(Calc.min(-3, -8));
   }
}
```

```
5
10
-8
```

static 메소드의 제약 조건 1

- static 메소드는 non-static 멤버 접근할 수 없음
  - 객체가 생성되지 않은 상황에서도 static 메소드는 실행될 수 있기 때문에, non-static 메소드와 필드 사용 불가
  - 반대로, non-static 메소드는 static 멤버 사용 가능

```
class StaticMethod {
    int n;
    void f1(int x) {n = x;} // 정상
    void f2(int x) {m = x;} // 정상

    static int m;
    static void s1(int x) {n = x;} // 컴파일 오류. static 메소드는 non-static 필드 사용 불가
    static void s2(int x) {f1(3);} // 컴파일 오류. static 메소드는 non-static 메소드 사용 불가

    static void s3(int x) {m = x;} // 정상. static 메소드는 static 필드 사용 가능
    static void s4(int x) {s3(3);} // 정상. static 메소드는 static 메소드 호출 가능
}
```

오류

오류

static 메소드의 제약 조건 2

- static 메소드는 this 사용불가
  - static 메소드는 객체가 생성되지 않은 상황에서도 호출이 가능하므로, 현재 객체를 가리키는 this 레퍼런스 사용할 수 없음

```
class StaticAndThis {
    int n;
    static int m;
    void f1(int x) {this.n = x;}
    void f2(int x) {this.m = x;} // non-static 메소드에서는 static 멤버 접근 가능
    static void s1(int x) {this.n = x;} // 컴파일 오류. static 메소드는 this 사용 불가
    static void s2(int x) {this.m = x;} // 컴파일 오류. static 메소드는 this 사용 불가
}
```

오류

오류

초기화의 순서

메모리를 할당하고 나면 반드시 초기화를 해야 한다. 자바에서는 변수를 다양한 방법으로 초기화를 할 수 있다.

1) 변수의 초기화

변수를 선언하고 할당된 메모리에 처음으로 값을 저장하는 것을 변수의 초기화라고 한다.

멤버변수의 초기화 = 생략가능
지역변수의 초기화 = 필수

2) 멤버변수의 초기화

멤버변수는 지역 변수와 달리 다음과 같이 다양한 방법으로 초기화 할 수 있다.

명시적 초기화
생성자를 이용한 초기화
초기화 블록을 이용한 초기화

```
class Test{
        static { /* 클래스 변수의 초기화 */ }
        { /* 인스턴스 변수의 초기화 */ }
}
```

정적 멤버와 static

❖초기화 블록

초기화 블록은 클래스 초기화 블록과 인스턴스 초기화 블록 두 가지가 있다. 각각 클래스 변수와 인스턴스 변수의 초기화에 사용한다. 초기화 블록은 다음과 같이 작성한다.

```
class Test{
        static {/* 클래스 변수의 초기화 */}
        {/* 인스턴스 변수의 초기화 */}
}
```

초기화 블록{}내에는 메서드의 몸체처럼 다양한 명령문을 함께 사용할 수 있다. 그래서 초기화 블록은 복잡한 초기화에 사용한다.

❖클래스 초기화 블록

클래스 초기화 블록에서는 클래스 변수를 초기화한다. 클래스가 한번 메모리에 적재될 때 클래스 초기화 블록도 한번만 초기화 된다.

정적 멤버와 static

- 정적 메소드와 정적 블록 작성시 주의할 점
  - 객체가 없어도 실행 가능
  - 블록 내부에 인스턴스 필드나 인스턴스 메소드 사용 불가
  - 객체 자신의 참조인 this 사용 불가
    - EX) main()

```
ClassName obj = new ClassName();
obj.field1 = 10;
obj.method1();
```

```
// 인스턴스 필드와 메서드
int field1;
void method1() { ... }
```

```
// 정적 필드와 메서드
static int field2;
static void method2() { ... }
```

```
// 정적 블록
static {
    field1 = 10;     (x)
    method1();       (x)
    field2 = 20;     (0)
    method2();       (0)
}
```

컴파일 에러

```
// 정적 메서드
static void method3() {
    this.field1 = 10;     (x)
    this.method1();       (x)
    field2 = 10;          (0)
    method2();            (0)
}
```

컴파일 에러

정적 멤버와 static

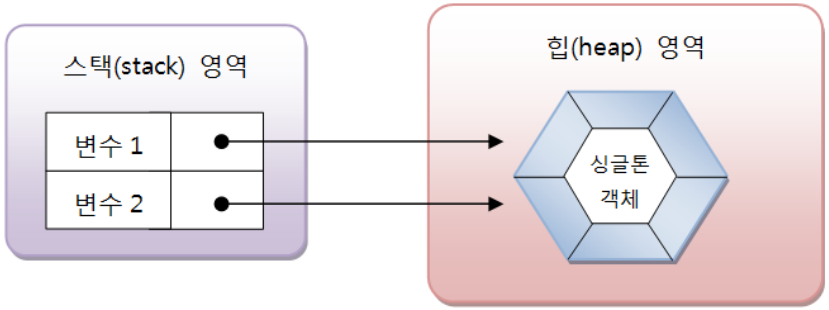
- 싱글톤(Singleton)
  - 하나의 애플리케이션 내에서 단 하나만 생성되는 객체

- 싱글톤을 만드는 방법
  - 외부에서 new 연산자로 생성자를 호출할 수 없도록 막기
    - private 접근 제한자를 생성자 앞에 붙임
  - 클래스 자신의 타입으로 정적 필드 선언
    - 자신의 객체를 생성해 초기화
    - private 접근 제한자 붙여 외부에서 필드 값 변경 불가하도록
  - 외부에서 호출할 수 있는 정적 메소드인 getInstance() 선언
    - 정적 필드에서 참조하고 있는 자신의 객체 리턴

정적 멤버와 static

❖싱글톤 얻는 방법

```
클래스 변수1 = 클래스.getInstane();

클래스 변수2 = 클래스.getInstane();
```




```
Singleton obj1 = new Singleton();
Singleton obj2 = new Singleton();

Singleton obj1 = Singleton.getInstane();
Singleton obj2 = Singleton.getInstane();

if(obj1 == obj2)
      System.out.println("같은 Singleton 객체입니다");
else
     System.out.println("다른 Singleton 객체입니다");
```

final 필드와 상수(static final)

❖final 필드

- 최종적인 값을 갖고 있는 필드 = 값을 변경할 수 없는 필드
- final 필드의 딱 한번의 초기값 지정 방법
  - 필드 선언 시
  - 생성자

```
public class Person{
    final String nation = "Korea";
    final String ssn;
    String name;

    public Person(){
        this.ssn = ssn;
        this.name = name;
    }
}
```

final 필드와 상수(static final)

- 상수(static final)
  - 상수 = 정적 final 필드
    - final 필드:
      - 객체마다 가지는 불변의 인스턴스 필드
    - 상수(static final):
      - 객체마다 가지고 있지 않음
      - 공용 데이터로서 사용
  - 상수 이름은 전부 대문자로 작성
  - 다른 단어가 결합되면 _ 로 연결

상수

정적 변수는 모든 객체가 공유하는 정보를 나타내는 데 주로 사용되는데 대표적인 것이 상수이다. 상수를 인스턴스 변수로 선언하면 각 객체마다 하나씩 만들어지므로 저장 공간이 낭비된다. 자바에서 상수는 final 키워드를 사용하여 만든다. 대문자만를 사용한다.

```
public class Car {
        …
        static final int MAX_SPEED = 350;
…
}
```

중간점검

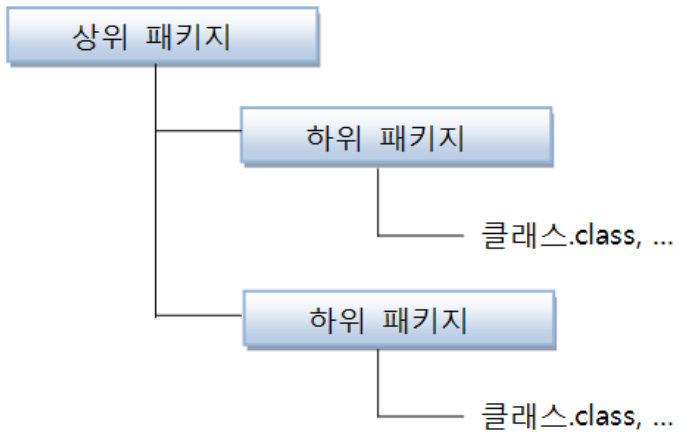
1. 정적 변수는 어떤 경우에 사용하면 좋은가?
2. 정적 변수나 정적 메소드를 사용할 때, 클래스 이름을 통하여 접근하는 이유는 무엇인가?
3. main() 안에서 인스턴스 메소드를 호출할 수 없는 이유는 무엇인가?

패키지(package)

- **패키지란? 클래스를 관리하는 방법**
  - 클래스를 기능별로 묶어서 그룹 이름을 붙여 놓은 것
    - 파일들을 관리하기 위해 사용하는 폴더(디렉토리)와 비슷한 개념
    - 패키지의 물리적인 형태는 파일 시스템의 폴더(디렉토리)

```
package 패키지명
```

  - 클래스 이름의 일부
    - 클래스를 유일하게 만들어주는 식별자
    - 전체 클래스 이름 = 상위패키지.하위패키지.클래스
    - 클래스명이 같아도 패키지명이 다르면 다른 클래스로 취급



패키지(package)

❖ **import 문** : 사용하려는 객체가 속한 패키지 정보를 나타내는 것

- 패키지 내에 같이 포함된 클래스간 클래스 이름으로 사용 가능
- 다른 패키지에 포함된 클래스를 사용해야 할 경우
  - 패키지 명 포함된 전체 클래스 이름으로 사용

```
package com.usedcar;
public class Car{
        com.newcar.Tire tire = new com.newcar.Tire();
}
```

  - import 문으로 패키지를 지정하고 사용

```
package com.usedcar;
import com.newcar.Tire;  [또는 import com.newcar.*;]

public class Car{
        Tire tire = new Tire();
}
```

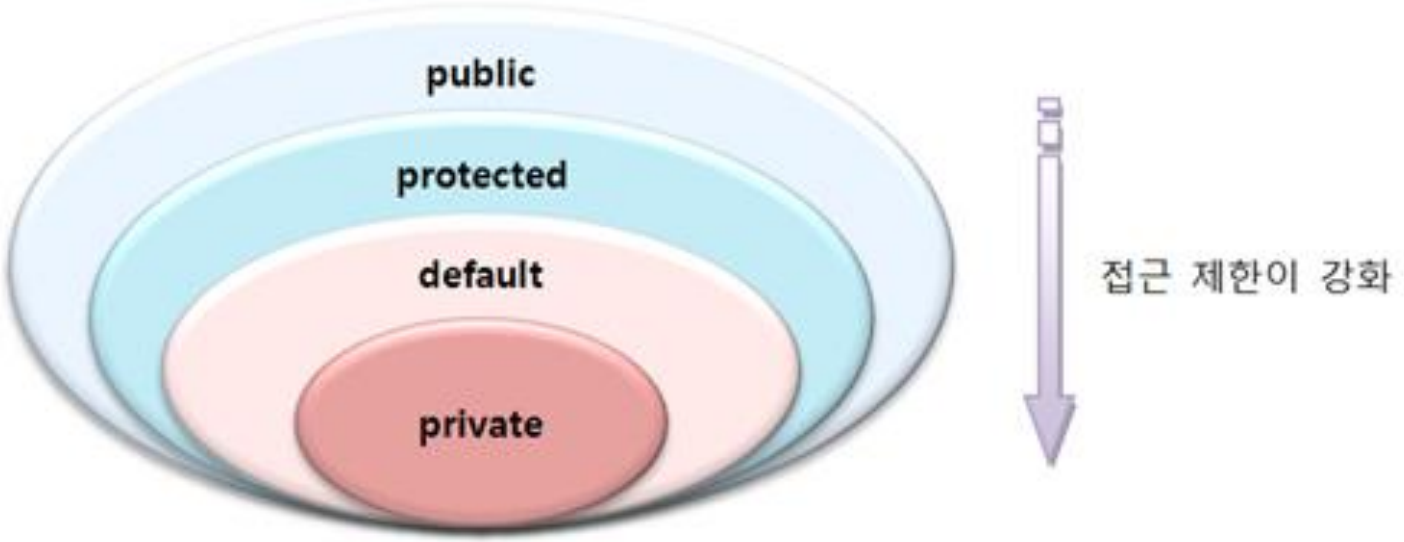
Source>Organize imports (단축키: Ctrl+Shift+O)

접근 제한자

❖접근 제한자(Access Modifier)

- 클래스 및 클래스의 구성 멤버에 대한 접근을 제한하는 역할
  - 다른 패키지에서 클래스를 사용하지 못하도록 (클래스 제한)
  - 클래스로부터 객체를 생성하지 못하도록 (생성자 제한)
  - 특정 필드와 메소드를 숨김 처리 (필드와 메소드 제한)

- 접근 제한자의 종류



접근 제어

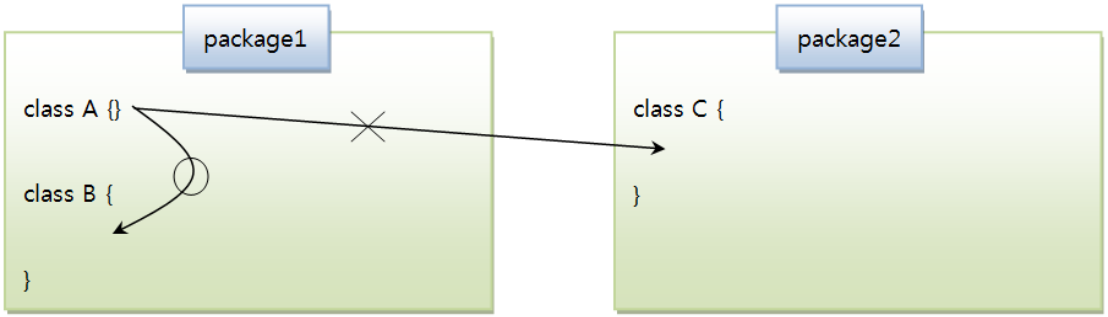
접근 제어(access control)란 다른 클래스가 특정한 필드나 메소드에 접근하는 것을 제어하는 것이다. private이나 public 등의 수식어를 필드나 메소드 앞에 붙여서 접근을 제한하게 된다.

접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메서드	없음.
protected	필드, 생성자, 메서드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메서드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메서드	모든 외부 클래스

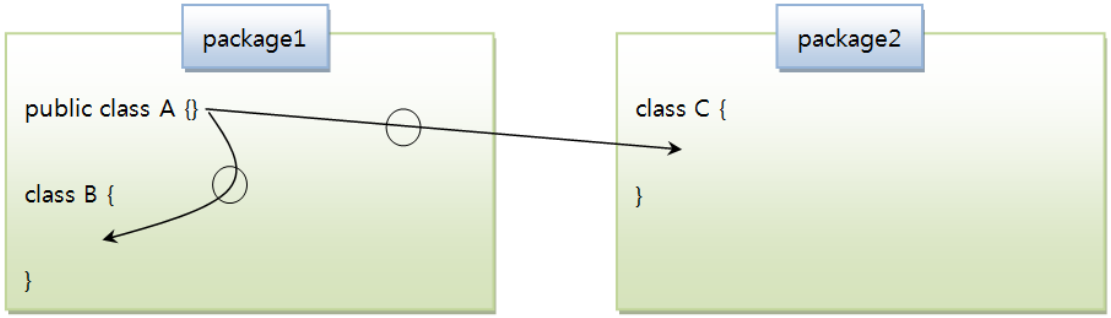
접근 제한자

❖클래스의 접근 제한

- default
  - 클래스 선언할 때 public 생략한 경우
  - 다른 패키지에서는 사용 불가



- public
  - 다른 개발자가 사용할 수 있도록 라이브러리 클래스로 만들 때 유용



접근 제어

- 클래스 수준에서의 접근 제어 - 클래스를 다른 클래스가 사용하게 하거나 못하게 하는 것.
- 멤버 수준에서의 접근 제어 - 필드나 메소드를 다른 클래스가 사용하게 하거나 못하게 하는 것.

클래스 수준에서의 접근 제어

클래스 앞에 붙일 수 있는 수식어에서 접근 제어와 관련되는 것이 public이다. 수식자 public으로 선언된 클래스는 다른 모든 클래스가 사용할 수 있는 공용 클래스가 된다. 만약 수식자가 없으면 자동적으로 패키지 수준의 클래스가 되고, 같은 패키지 안에 있는 클래스만이 사용할 수 있다.

public: 다른 모든 클래스가 사용할 수 있는 공용 클래스
default: 수식자가 없으면, 같은 패키지 안에 있는 클래스들만이 사용

패키지(package)는 관련된 클래스를 모아둔 것

```
public class myClass
{
        ...
}
```

```
class myClass {
        ...
}
```

멤버 수준에서의 접근 제어

전용멤버, 디폴트 멤버, 공용멤버로 나눌 수 있다.

분류	접근 제어자	클래스 내부	같은 패키지 내의 클래스	다른 모든 클래스
전용 멤버	private	0	X	X
디폴트 멤버	없음	0	0	X
공용 멤버	public	0	0	0

멤버 수준에서의 접근 제어

- 전용 멤버 : 클래스 내부에서만 접근이 허용된다. private를 이름 앞에 붙인다.
- 디폴트 멤버 : 멤버를 정의할 때 아무런 접근 지정자를 붙이지 않으면 자동적으로 패키지 멤버로 된다. 디폴트 멤버는 같은 패키지 안에 있는 모든 클래스가 접근할 수 있다.
- 공용 멤버 : 공용멤버는 다른 모든 클래스들이 사용할 수 있다. 멤버이름 앞에 public을 붙인다.

접근 제어

(접근 제어를 선택하는 방법)

- 일반적으로 멤버에 대해서는 가장 엄격한 접근 제어를 선택하라. 만약 특별한 이유가 없으면 private를 선택하라.
- 상수를 제외하고는 필드에 public을 사용하면 안된다. 외부 클래스들이 필드를 직접 사용하면 코드를 변경하기가 힘들어진다.
- 일반적으로 메소드에 대해서는 외부에서 접근 가능한 접 public을 선택하라.

클래스와 클래스 간의 관계

자바 프로그램은 여러 개의 클래스로 이루어진다. 클래스와 클래스 간에는 어떤 관계가 있을까? 일반적으로 3가지의 관계를 생각할 수 있다.

- 사용(use) : 하나의 클래스가 다른 클래스를 사용한다
- 집합(has-a) : 하나의 클래스가 다른 클래스를 포함한다
- 상속(is-a) : 하나의 클래스가 다른 클래스를 상속한다.

사용관계(use)

이 관계에서는 클래스 A의 메소드에서 클래스 B의 메소드들을 호출한다. 클래스 B의 메소드를 호출하려면 클래스 B의 객체에 대한 참조를 가져야 한다. CarTest 클래스에서 Car 클래스 객체를 생성하여서 사용하는 경우가 바로 사용관계이다. 사용관계에서 특히 흥미로운 형태는 하나의 클래스가 자기 자신을 사용하는 경우이다. 이것은 주로 객체 참조를 매개변수로 받는 형태로 나타난다.

집합관계(has-a)

이 관계에서는 하나의 객체 안에 다른 객체들이 포함된다. 하나의 객체 안에 객체들에 대한 참조를 포함하고 있다.

클래스는 다른 클래스의 객체에 대한 참조를 멤버로서 가질 수 있다. 이것은 흔히 has-a 관계로 불린다.


Thank You