final 필드(제한자)와 상수

> 인스턴스 필드와 정적 필드는 언제든지 값을 변경할 수 있지만 final은 값을 변경하지 않고 읽기만 허용된다.

- final 필드 선언

```
final 타입 필드 = [=초기값];
```

*final에 초기값을 주는 방법

1. 필드 선언 시에 초기값 대입
2. 생성자에서 초기값 대입

*final 필드의 주요 특징

1. 초기화 후 값을 변경할 수 없으며 상수 역할을 한다.
2. 선언 시점이나 생성자 내부에서 반드시 초기화되어야 하며 초기화되지 않을 경우 컴파일 에러가 난다.
3. 컴파일 타임에 상수로 평가되기 때문에 실행 중 값을 변경할 수 없다. 하지만 그렇기 때문에 컴파일러가 이미 값을 알고 있어 빠른 성능을 제공한다.
4. 다른 스레드로부터 변경이 없기 때문에 동기화 없이 안정성을 보장할 수 있다.
5. 주로 상수로 사용한다. public static final MAX_VALUE = 100;
6. fianl 필드는 선언 시 또는 생성자에서 초기화된다. 생성자를 통해 final 필드에 값을 할당할 때, 한 번 할당된 값은 변경할 수 없다.

인스턴스 필드에 final 적용

```
public class MyClass {
    // 인스턴스 필드에 final 적용
    private final int instanceField;

    public MyClass(int value) {
        this.instanceField = value; // 생성자에서 초기화
    }

    public int getInstanceField() {
        return instanceField;
    }
}
```

클래스(static) 필드에 final 적용

```
public class MyConstants {
    // 클래스(static) 필드에 final 적용
    public static final double PI = 3.14159265359;
```

```
    public static final String GREETING = "Hello, World!";
}
```

```
// Korea.java
public class Korea {
    // 인스턴스 final 필드 선언
    final String nation = "대한민국";
    final String ssn;

    // 인스턴스 필드 선언
    String name;

    // 생성자 선언
    public Korea(String ssn, String name) {
        this.ssn = ssn;
        this.name = name;
    }
}

// KoreaExample.java
public class KoreaExample {
    public static void main(String[] args) {
        Korea k1 = new Korea("2323545-345345", "아무개");

        System.out.println(k1.nation);
        System.out.println(k1.ssn);
        System.out.println(k1.name);

        //
        k1.name = "김자바";

        System.out.println(k1.name);
    }
}
```

상수 선언

초기값은 선언 시에 주는 것이 일반적이지만, 복잡한 초기화가 필요한 경우 정적 블록에서 초기화할 수도 있다.

- 상수 이름은 대문자로 작성한다.
- 혼합된 이름은 언더바(_)로 연결한다.
- 선언시에 초기화한다.

```
static final 타입 상수 [= 초기값];
```

```

```

```
static final 타입 상수;
static {
    상수 = 초기값;
}

static final double PI = 3.14;
static final double EARTH_SURFACE_AREA = 5.1471;
```

```
클래스명.상수
```

6.12 패키지

- package 선언은 Java 클래스가 속하는 패키지를 명시하는데 필수이다.
- 패키지 이름은 소문자로 작성하는 것이 관례이다.
- 같은 패키지의 클래스만 불러올 수 있다.

자바의 패키지는 디렉토리만 의미하지 않는다. 패키지는 클래스를 식별하는 용도로 사용되기 때문에 클래스의 전체 이름에 포함된다. 예를 들어 Car 클래스가 com.company 패키지에 속해 있다면 전체 이름은 com.company.Car 가 된다.

소스 파일이(~.java) 저장되면 이클립스는 bin 디렉토리에 패키지 디렉토리와 함께 바이트코드 파일(~.class)을 생성한다.

패키지 선언

패키지 디렉토리는 클래스를 컴파일하는 과정에서 자동으로 생성된다.

```
package 상위패키지.하위패키지;

public class 클래스명 {...}
```

import 문

다른 패키지의 클래스를 사용하려면 import 문을 이용해 클래스를 명시한다.

- import 문은 하위 패키지를 포함하지 않는다.
- package 선언과 class 선언 사이에 import 가 작성된다.
- 동일한 이름의 클래스를 import 할 경우 패키지 이름 전체를 작성해야 하며 이 경우 import 는 필요 없다.

```
package com.company;

import com.depart.Tire;

public class Car {
    // import 선언으로 불러옴
```

```
    Tire tire = new Tire();
}
```

```
// 상위 패키지 클래스와 하위 패키지 클래스 모두 사용하기 위해선 상위/하위 패키지 모두 선언해야 한다.
// 하위 패키지를 포함하지 않기 때문에
import com.hankook.*;
import com.hankook.projcet.*
```

```
// 동일한 클래스를 사용할 경우
com.hankook.Tire tire = new com.hankook.Tire();
```

6.13 접근 제한자

객체의 무결성을 유지하기 위해 사용된다. 이것을 캡슐화라고한다.

- public
- protected
- private

접근 제한자	제한 대상	제한 범위
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	같은 패키지거나, 자식 객체만 사용 가능
default	클래스, 필드, 생성자, 메소드	같은 패키지
private	필드, 생성자, 메소드	객체 내부

클래스의 접근 제한

클래스 선언 시 public 제한자를 생략했다면 default 접근 제한을 가진다. 이 경우 클래스는 같은 패키지에서는 제한 없이 사용할 수 있지만 다른 패키지에서는 사용할 수 없다.

- public
- default

```
src.package01;
// default
class A {}
```

```
src.package02;
src.package01.A;
```

```
public class B {
    A a; // 다른 패키지의 클래스이므로 접근불가
}
```

생성자의 접근 제한

객체를 생성하기 위해 생성자를 어디서나 호출할 수 있는 것이 아니다. 어떤 접근제한을 갖느냐에 따라 호출 가능 여부가 결정된다. 생성자는 public, default, private 접근 제한을 가진다.

```
public class ClassName {
    // 생성자 선언
    [public | private] className {

    }
}
```

접근 제한자	생성자	설명
public	클래스()	모든 패키지에서 생성자를 호출할 수 있다. = 모든 패키지에서 객체를 생성할 수 있다.
	클래스()	같은 패키지에서만 생성자를 호출할 수 있다. = 같은 패키지에서만 객체를 생성할 수 있다.
private	클래스()	클래스 내부에서만 생성자를 호출할 수 있다. = 클래스 내부에서만 객체를 생성할 수 있다.

```
package src.package01;

public class A {
    public A();
    A(boolean value);
    private A(int value);
}
```

```
package src.package02;

import src.package01.A;

public class B {
    A a = new A();
    A a = new A(true); // 에러
    A a = new A(1); // 에러
}
```

필드와 메소드의 접근 제한

필드와 메소드도 접근 제한에 따라 호출 여부가 결정된다. public, default, private 접근 제한을 가진다.

접근 제한자	생성자	설명
public	필드, 메소드()	모든 패키지에서 필드를 읽고 변경할 수 있다. 모든 패키지에서 메소드를 호출할 수 있다.
	필드, 메소드()	같은 패키지에서 필드를 읽고 변경할 수 있다. 같은 패키지에서 메소드를 호출할 수 있다.
private	필드, 메소드()	클래스 내부에서만 필드를 읽고 변경할 수 있다. 클래스 내부에서만 메소드를 호출할 수 있다.

```
package org.example.controller;

public class Common {
    public String name;
    public int age;

    public Common(){

    }

    public int getNumber(int age){
        return this.age = age;
    }
}
```

```
package org.example.service;

import org.example.controller.Common;

public class Service {
    Common comm = new Common();

    int number = comm.getNumber(1);

}
```

Getter와 Setter

객체의 필드(데이터)를 외부에서 읽고 변경할 경우 객체의 무결성(결점이 없음)이 깨질 수 있다.

- 메서드 호출은 메서드나 생성자 내부에서 이루어져야 합니다.

```
Car myCar = new Car();
// 속력은 음수가 올 수 없다. 고로 무결성이 깨진다.
myCar.speed = -100;
```

```
package org.example.controller;

public class Common {
    public String name;
    public int age;

    public Common(){

    }

    public void setNumber(int age){
        if(age < 0){
            this.age = 0;
        }else {
            this.age = age;
        }
    }

    public int getNumber(){
        return age;
    }
}
```

```
package org.example.service;

import org.example.controller.Common;

public class Service {
    public static void Service(){
        Common comm = new Common();

        comm.setNumber(1);
        int getNum = comm.getNumber();
    }
}
```

6.15 싱글톤 패턴

애플리케이션 전체에서 단 한 개의 객체만 생성해서 사용하고 싶다면 싱글톤 패턴을 적용할 수 있다. 싱글톤 패턴의 핵심은 private 접근 제한에서 외부에서 new 연산자로 생성자를 호출할 수 없도록 막는 것이다.

```java
package org.example.controller;

public class SingleTone {
    // 1. private 접근 권한을 가지는 정적 필드 선언과 초기화
    private static SingleTone singletone = new SingleTone();

    // 2. private 접근 권한을 갖는 생성자 선언
    private SingleTone(){};

    // 3. public 접근 권한을 갖는 정적 메소드 선언
    public static SingleTone getInstance() {
        return singletone;
    }

}
```

```java
package org.example.controller;

public class SingleToneExample {
    public static void main(String[] args){
        SingleTone obj1 = SingleTone.getInstance();
        SingleTone obj2 = SingleTone.getInstance();

        if(obj1 == obj2){
            System.out.println("같은 싱글톤 객체입니다.");
        }
    }

}
```