

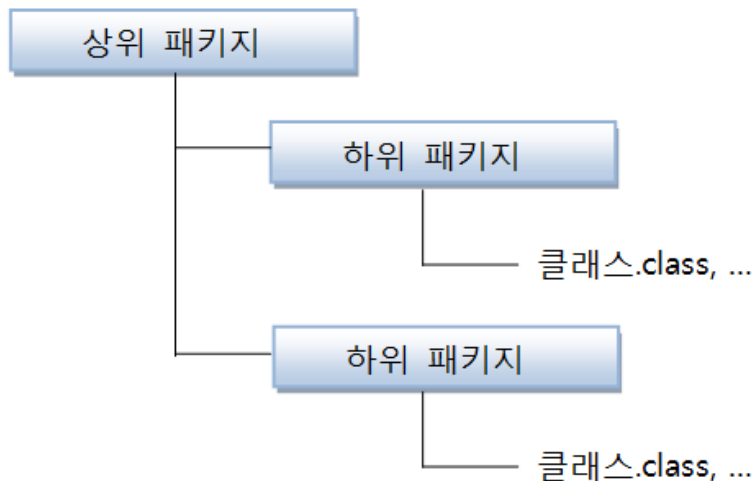
Contents

- ❖ 12절. 패키지(package)
- ❖ 13절. 접근 제한자
- ❖ 14절. Getter와 Setter

12절. 패키지(package)

❖ 패키지란?

- 클래스를 기능별로 묶어서 그룹 이름을 붙여 놓은 것
 - 파일들을 관리하기 위해 사용하는 폴더(디렉토리)와 비슷한 개념
 - 패키지의 물리적인 형태는 파일 시스템의 폴더
- 클래스 이름의 일부
 - 클래스를 유일하게 만들어주는 식별자
 - 전체 클래스 이름 = 상위패키지.하위패키지.클래스
 - 클래스명이 같아도 패키지명이 다르면 다른 클래스로 취급

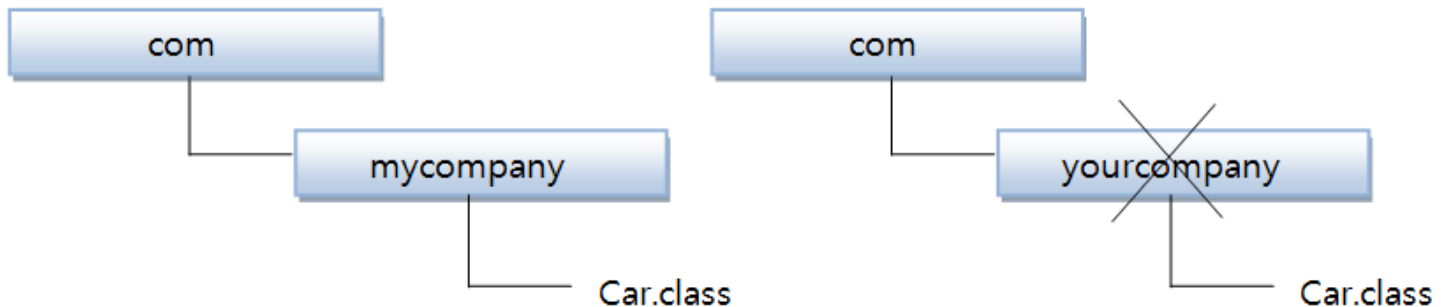


12절. 패키지(package)

❖ 패키지란?

■ 클래스 선언할 때 패키지 결정

- 클래스 선언할 때 포함될 패키지 선언
- 클래스 파일은(~.class) 선언된 패키지와 동일한 폴더 안에서만 동작
- 클래스 파일은(~.class) 다른 폴더 안에 넣으면 동작하지 않음



12절. 패키지(package)

❖ import 문

- 다른 패키지에 있는 클래스를 사용해야 할 경우
 - 패키지 명이 포함된 전체 클래스 이름으로 사용

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

- Import 문으로 패키지를 지정하고 사용

```
package com.mycompany;  
  
import com.hankook.Tire;  
[ 또는 import com.hankook.*; ]  
  
public class Car {  
    Tire tire = new Tire();  
}
```

12절. 패키지(package)

❖ import 문 사용 시 주의 사항

- import 문이 작성되는 위치는 패키지 선언과 클래스 선언 사이
- 패키지에 포함된 다수의 클래스를 사용해야 한다면 클래스 별로 import문을 작성할 필요없이 *를 사용한다.
- import 문으로 지정된 패키지의 하위 패키지는 import 대상이 아니다.
- 다음 두 import 문은 서로 다르다.

```
import com.mycompany.*;  
import com.mycompany.project.*;
```

- 서로 다른 패키지에 동일한 이름의 클래스가 있는 경우에는 import 문을 사용하더라도 패키지 이름 전체를 기술해야 한다.

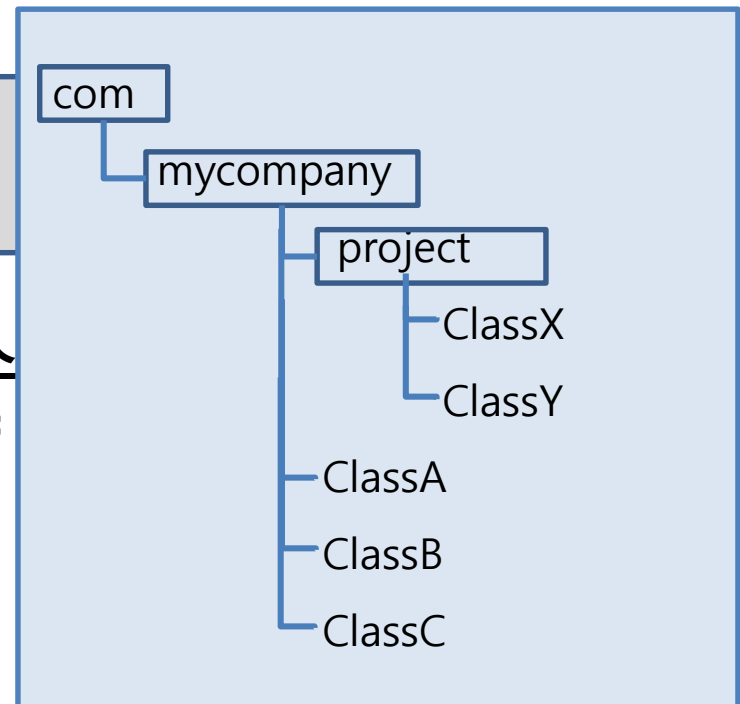
12절. 패키지(package)

❖ import 문 사용 시 주의 사항

- import 문이 작성되는 위치는 패키지 선언과 클래스 선언 사이
- 패키지에 포함된 다수의 클래스를 사용해야 한다면 클래스 별로 import문을 작성할 필요없이 *를 사용한다.
- import 문으로 지정된 패키지의 하위 패키지는 import 대상이 아니다.
- 다음 두 import 문은 서로 다르다.

```
import com.mycompany.*;  
import com.mycompany.project.*;
```

- 서로 다른 패키지에 동일한 이름의 클래스용하더라도 패키지 이름 전체를 기술해야



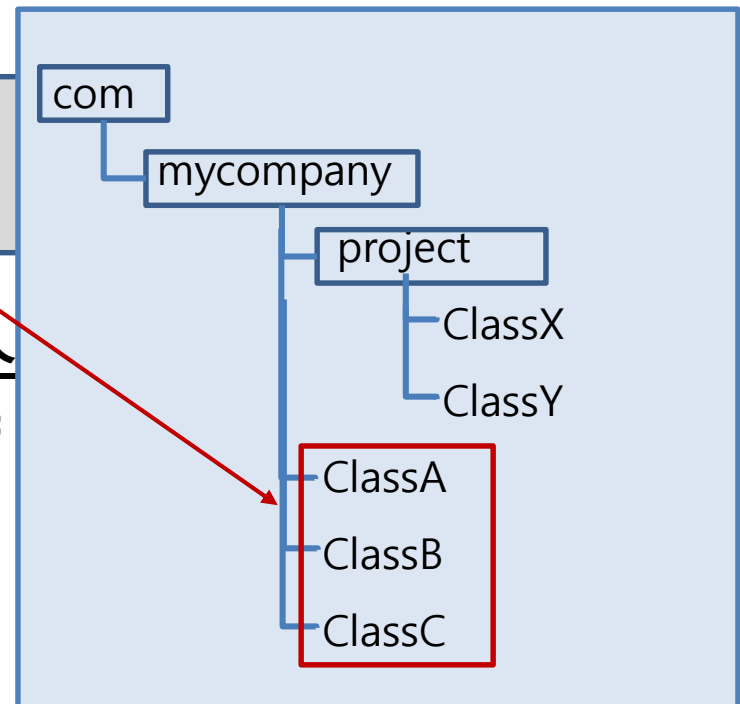
12절. 패키지(package)

❖ import 문 사용 시 주의 사항

- import 문이 작성되는 위치는 패키지 선언과 클래스 선언 사이
- 패키지에 포함된 다수의 클래스를 사용해야 한다면 클래스 별로 import문을 작성할 필요없이 *를 사용한다.
- import 문으로 지정된 패키지의 하위 패키지는 import 대상이 아니다.
- 다음 두 import 문은 서로 다르다.

```
import com.mycompany.*;  
import com.mycompany.project.*;
```

- 서로 다른 패키지에 동일한 이름의 클래스용하더라도 패키지 이름 전체를 기술해야



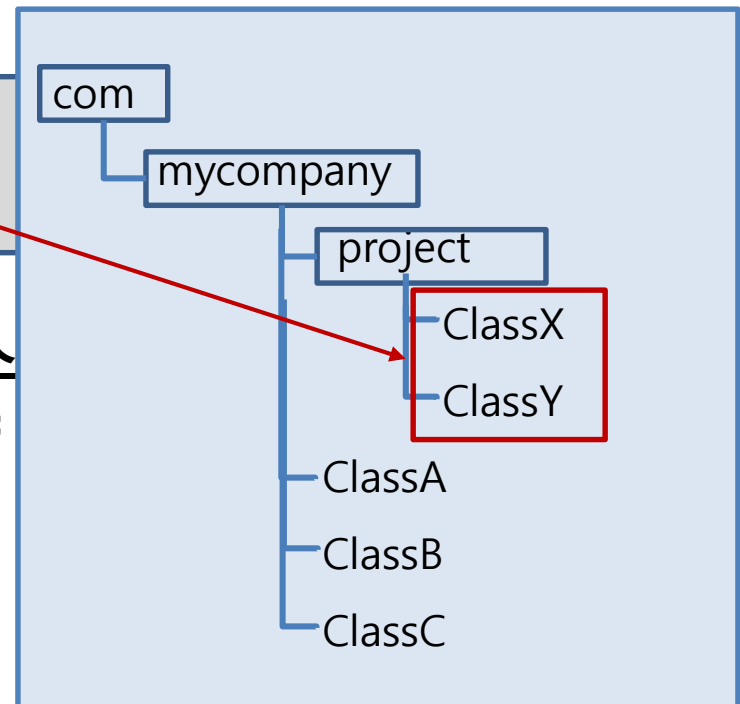
12절. 패키지(package)

❖ import 문 사용 시 주의 사항

- import 문이 작성되는 위치는 패키지 선언과 클래스 선언 사이
- 패키지에 포함된 다수의 클래스를 사용해야 한다면 클래스 별로 import문을 작성할 필요없이 *를 사용한다.
- import 문으로 지정된 패키지의 하위 패키지는 import 대상이 아니다.
- 다음 두 import 문은 서로 다르다.

```
import com.mycompany.*;  
import com.mycompany.project.*;
```

- 서로 다른 패키지에 동일한 이름의 클래스용하더라도 패키지 이름 전체를 기술해야



12절. 패키지(package)

❖ import 문 사용 시 주의 사항

- import 문이 작성되는 위치는 패키지 선언과 클래스 선언 사이
- 패키지에 포함된 다수의 클래스를 사용해야 한다면 클래스 별로 import문을 작성할 필요없이 *를 사용한다.
- import 문으로 지정된 패키지의 하위 패키지는 import 대상이 아니다.
- 다음 두 import 문은 서로 다르다.

```
import com.mycompany.*;  
import com.mycompany.project.*;
```

- 서로 다른 패키지에 동일한 이름의 클래스가 있는 경우에는 import 문을 사용하더라도 패키지 이름 전체를 기술해야 한다.

13절. 접근 제한자

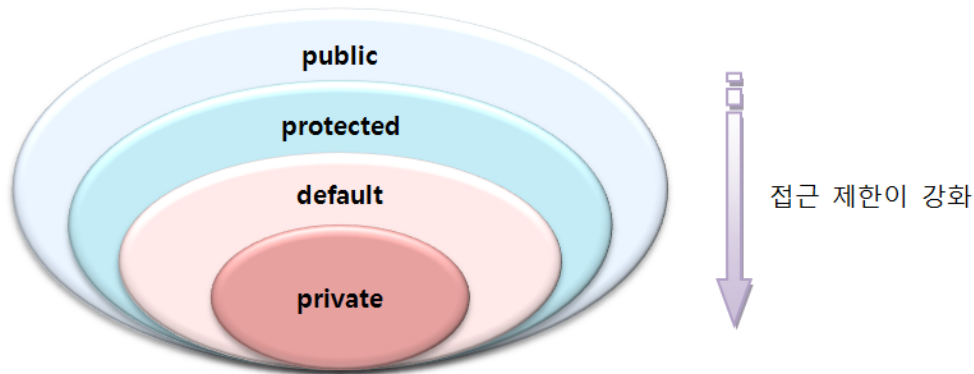
필드, 생성자, 메소드

❖ 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



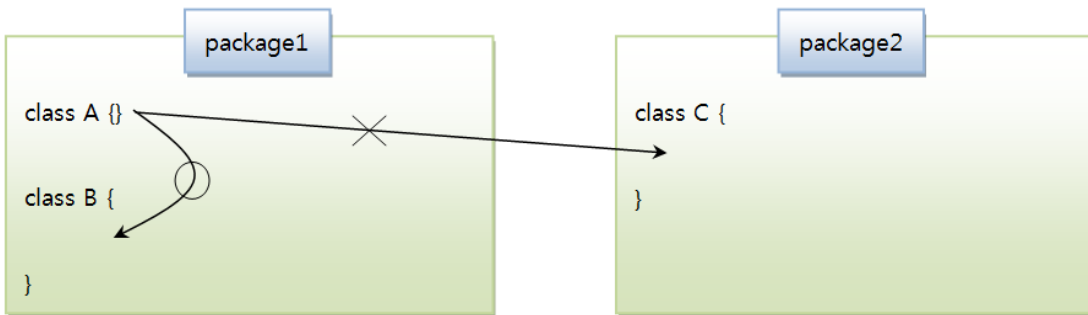
접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스

13절. 접근 제한자

❖ 클래스의 접근 제한

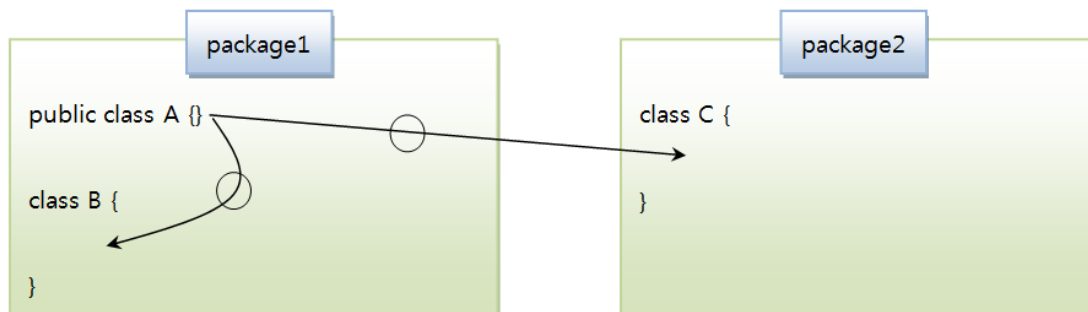
■ default

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



■ public

- 다른 개발자가 사용할 수 있도록 라이브러리 클래스로 만들 때 유용



13절. 접근 제한자

❖ 생성자 접근 제한

- 객체를 생성하기 위해 생성자를 어디에서나 호출할 수 있는 것은 아님
- 생성자가 가지는 접근 제한에 따라 호출 여부 결정
- 생성자 접근 제한자
 - `public` : 모든 패키지에서 생성자 호출 가능
 - `default` : 같은 패키지에서만 생성자 호출 가능
 - `private` : 클래스 내부에서만 생성자 호출 가능

13절. 접근 제한자

❖ 생성자 접근 제한

- 생성자가 가지는 접근 제한에 따라 호출 여부 결정

```
2
3 public class ClassA {
4
5     ClassA c1 = new ClassA(true);
6     ClassA c2 = new ClassA(10);
7     ClassA c3 = new ClassA("문자열");
8
9     ClassA(int data){    }
10    public ClassA(boolean data) {    }
11    private ClassA(String data) {    }
12 }
```

```
public class ClassB {
    ClassA c1 = new ClassA(true);
    ClassA c2 = new ClassA(10);

    //package가 같아도 private은 호출할 수 없다.
    //ClassA c3 = new ClassA("홍길동");
}
```

```
package package1;

import week10.ClassA;

public class ClassB {
    //package가 다른 public만 호출 가능
    ClassA c1 = new ClassA(true);

    //package가 다른 default인 경우도 호출할 수 없다.
    //ClassA c2 = new ClassA(10);

    //private은 자기 클래스 안에서만 호출 가능
    //ClassA c3 = new ClassA("홍길동");
}
```

13절. 접근 제한자

❖ 생성자 접근 제한

■ 생성자가 가지는 접근 제한에 따라 호출 여부 결정

```
2
3 public class ClassA {
4
5     ClassA c1 = new ClassA(true);
6     ClassA c2 = new ClassA(10);
7     ClassA c3 = new ClassA("문자열");
8
9     ClassA(int data){    }
10    public ClassA(boolean data) {    }
11    private ClassA(String data) {    }
12 }
```

default 접근제한자인 경우에는
같은 패키지에서만 사용 가능

```
public class ClassB {
    ClassA c1 = new ClassA(true);
    ClassA c2 = new ClassA(10);

    //package가 같아도 private은 호출할 수 없다.
    //ClassA c3 = new ClassA("홍길동");
}
```

```
package package1;

import week10.ClassA;

public class ClassB {
    //package가 다르면 public만 호출 가능
    ClassA c1 = new ClassA(true);

    //package가 다르면 default인 경우도 호출할 수 없다.
    //ClassA c2 = new ClassA(10);

    //private은 자기 클래스 안에서만 호출 가능
    //ClassA c3 = new ClassA("홍길동");
}
```

13절. 접근 제한자

❖ 필드와 메소드의 접근 제한

- 언제 어디서나 호출할 수 있는 것이 아니라 어떤 접근 제한을 갖느냐에 따라 호출 여부가 결정됨
- 클래스 내부, 패키지 내, 패키지 상호간에 사용할 지 고려해 선언
- 필드와 메소드 접근 제한자
 - **public** : 모든 패키지에서 필드 접근과 메소드 호출 가능
 - **default** : 같은 패키지에서만 필드 접근과 메소드 호출 가능
 - **private** : 클래스 내부에서만 필드 접근과 메소드 호출 가능

13절. 접근 제한자

❖ 필드와 메소드의 접근 제한

```
package week10;

public class ClassA {
    public int field1;    // public 접근제한
    int field2;           // default 접근제한
    private int field3;   // private 접근제한

    public ClassA() {
        field1 = 1;       // 클래스 내부에서는
        field2 = 1;       // 접근 제한자의
        field3 = 1;       // 영향을 받지 않는다

        method1();
        method2();
        method3();
    }

    public void method1() { } // public 접근제한
    void method2() { }       // default 접근제한
    private void method3() { } // private 접근제한
}
```

```
package week10;    // 패키지가 동일

public class ClassB {
    public ClassB() {
        ClassA aa = new ClassA();
        aa.field1 = 1;
        aa.field2 = 1;
        aa.field3 = 1; (X) // private 필드 접근 불가

        aa.method1();
        aa.method2();
        aa.method3(); (X) // private 메소드 접근 불가
    }
}
```


13절. 접근 제한자

❖ 필드와 메소드의 접근 제한

```
package week10;

public class ClassA {
    public int field1;      // public 접근제한
    int field2;             // default 접근제한
    private int field3;     // private 접근제한

    public ClassA() {
        field1 = 1;        // 클래스 내부에서는
        field2 = 1;        // 접근 제한자의
        field3 = 1;        // 영향을 받지 않는다

        method1();
        method2();
        method3();
    }

    public void method1() { } // public 접근제한
    void method2() { }       // default 접근제한
    private void method3() { } // private 접근제한
}
```

```
package project1;          // 패키지가 다름

import week10.*;

public class ClassC {
    public ClassC() {
        ClassA aa = new ClassA();
        aa.field1 = 1;
        aa.field2 = 1; (X)   // default 필드 접근 불가
        aa.field3 = 1; (X)   // private 필드 접근 불가

        aa.method1();
        aa.method2(); (X)    // default 메소드 접근 불가
        aa.method3(); (X)    // private 메소드 접근 불가
    }
}
```

14절. Getter와 Setter

❖ 객체지향 프로그래밍에서는

- 외부에서 직접적인 필드 접근을 막고 대신 메소드를 통해 필드 접근

❖ 클래스 선언할 때 필드는 일반적으로 private 접근 제한

- 읽기 전용 필드가 있을 수 있음 (Getter의 필요성)
- 외부에서 엉뚱한 값으로 변경할 수 없도록 (Setter의 필요성)

❖ Getter

- private 필드의 값을 리턴 하는 역할 - 필요할 경우 필드 값 가공
- **getFieldName()** 또는 **isFieldName()** 메소드
 - 필드 타입이 boolean 일 경우 isFieldName()

❖ Setter

- 외부에서 주어진 값을 필드 값으로 수정 - 필요할 경우 유효성 검사
- **setFieldName(타입 변수)** 메소드
 - 매개 변수 타입은 필드의 타입과 동일

14절. Getter와 Setter

❖ 예제

- Source > Generate Getters and Setters > Select All > Generate

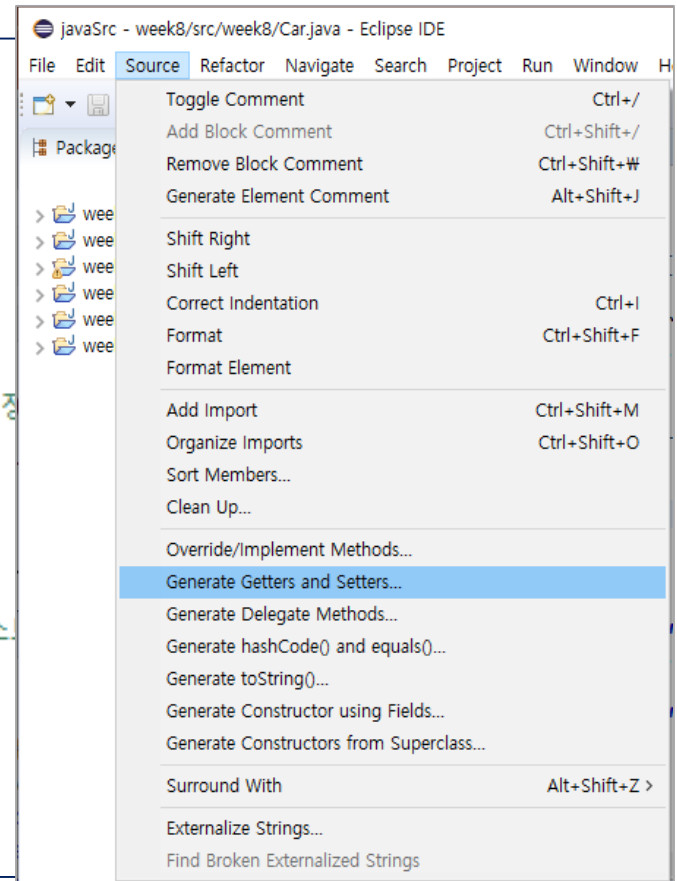
```
2
3 public class CarBasic {
4     // 클래스의 각 필드를 private으로 선언해서 외부 접근을 제한한다
5     private String company;
6     private String model;
7     private String color;
8     private int    maxSpeed;
9
10    //외부에서 값이 필요한 경우 값을 제공하기 위한 getter() 메소드 정의
11    public String getCompany() {
12        return company;
13    }
14
15    //외부로부터 주어진 값을 필드값으로 적용하기 위해 setter() 메소드 정의
16    //필요한 경우 외부값의 유효성 검사가 가능하다
17    public void setCompany(String company) {
18        this.company = company;
19    }
20
```

14절. Getter와 Setter

❖ 예제

- Source > Generate Getters and Setters > Select All > Generate

```
2
3 public class CarBasic {
4     // 클래스의 각 필드를 private으로 선언해서 외부 접근을 제한한다
5     private String company;
6     private String model;
7     private String color;
8     private int    maxSpeed;
9
10    //외부에서 값이 필요한 경우 값을 제공하기 위한 getter() 메소드 정
11    public String getCompany() {
12        return company;
13    }
14
15    //외부로부터 주어진 값을 필드값으로 적용하기 위해 setter() 메소
16    //필요한 경우 외부값의 유효성 검사가 가능하다
17    public void setCompany(String company) {
18        this.company = company;
19    }
20
```



setMaxspeed() 유효성 검사 추가

14절. Getter와 Setter

❖ 예제

```
2
3 public class CarBasicEx {
4     public static void main(String[] args) {
5         CarBasic car1 = new CarBasic();
6         car1.setCompany("현대자동차");
7         car1.setModel("그랜저");
8         car1.setColor("오션블루");
9         car1.setMaxSpeed(350);
10
11         System.out.println(car1.getCompany());
12         System.out.println(car1.getModel());
13         System.out.println(car1.getColor());
14         System.out.println(car1.getMaxSpeed());
15         System.out.println("-----");
16     }
```



7장. 상속

Contents

- ❖ 1절. 상속 개념
- ❖ 2절. 클래스 상속(extends)
- ❖ 3절. 부모 생성자 호출(super(...))
- ❖ 4절. 메소드 재정의(Override)
- ❖ 5절. final 클래스와 final 메소드
- ❖ 6절. protected 접근 제한자

1절. 상속 개념

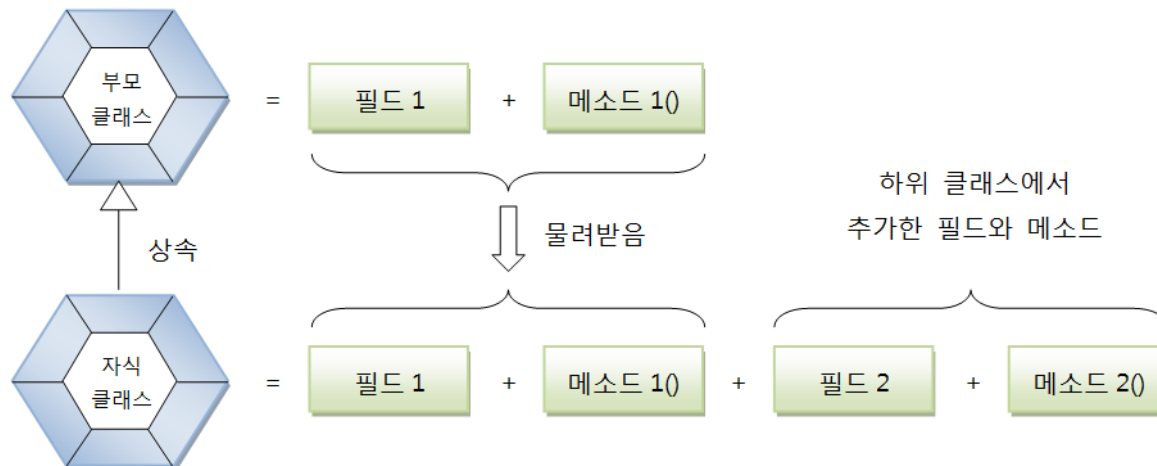
❖ 상속(Inheritance)이란?

■ 현실 세계:

- 부모가 자식에게 물려주는 행위
- 부모가 자식을 선택해서 물려줌

■ 객체 지향 프로그램:

- 자식(하위, 파생) 클래스가 부모(상위) 클래스의 멤버를 물려받는 것
- 자식이 부모를 선택해 물려받음
- 상속 대상: 부모의 필드와 메소드



1절. 상속 개념

❖ 상속(Inheritance) 개념의 활용

■ 상속의 효과

- 부모 클래스를 재사용해 자식 클래스를 빨리 개발 가능
- 반복된 코드 중복 줄임
- 유지 보수 편리성 제공
- 객체 다형성 구현 가능

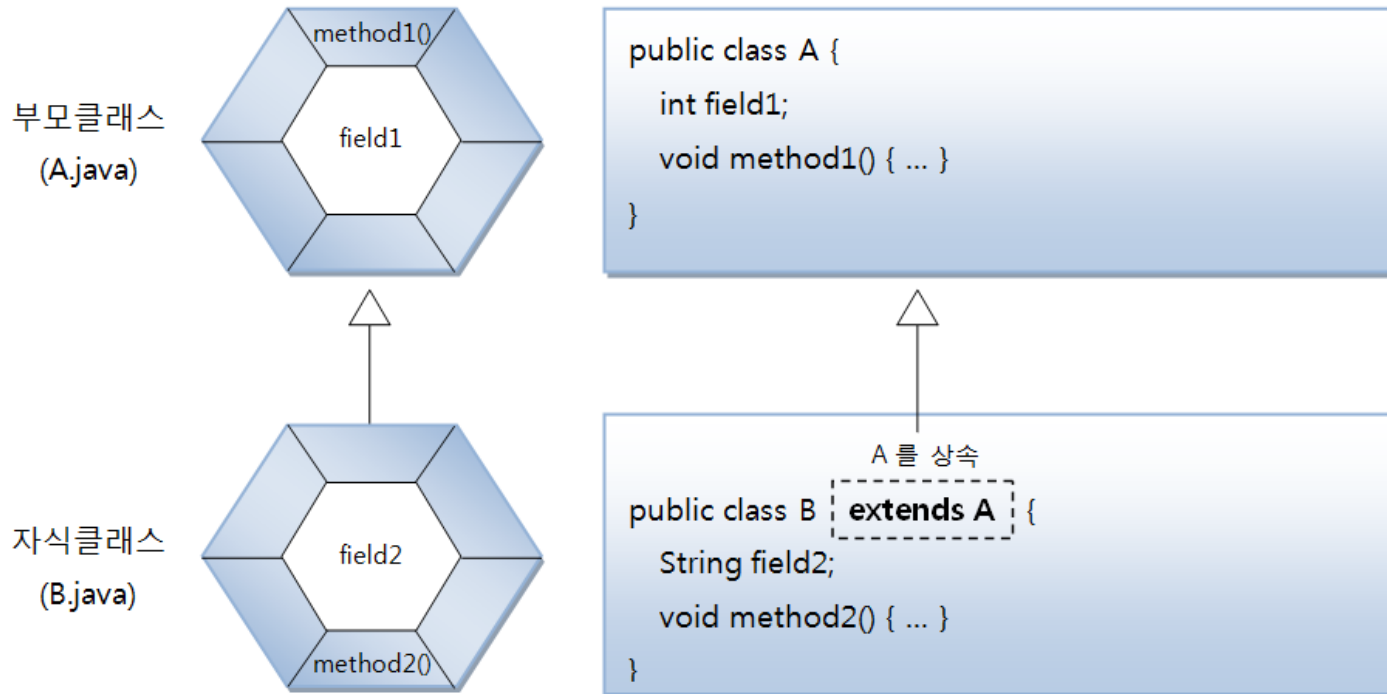
■ 상속 대상 제한

- 부모 클래스의 private 접근을 갖는 필드와 메소드 제외
- 부모 클래스가 다른 패키지에 있을 경우, default 접근 갖는 필드와 메소드도 제외

2절. 클래스 상속(extends)

❖ extends 키워드

- 자식 클래스가 상속할 부모 클래스를 지정하는 키워드



- 자바는 단일 상속 - 부모 클래스 나열 불가

```
class 자식클래스 extends 부모클래스 1, 부모클래스 2 {  
}
```

2절. 클래스 상속(extends)

❖ 상속 예제

CordedPhone

- model, color
- bell()
- sendVoice()
- receiveVoice()
- hangUp()
- lineInfo()

SmartPhone

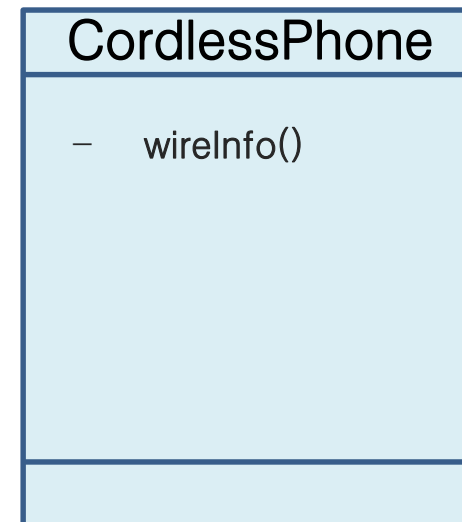
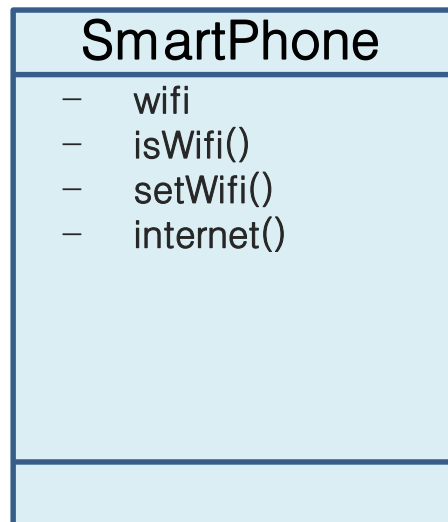
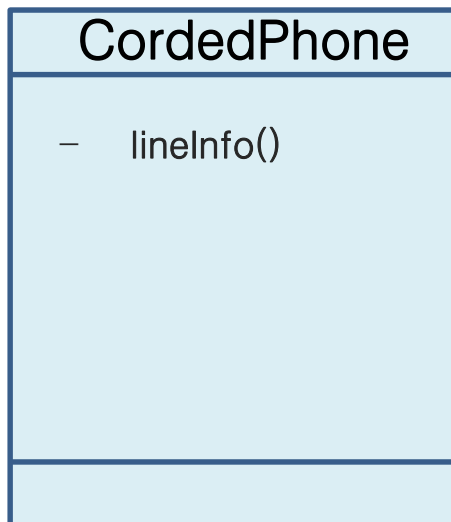
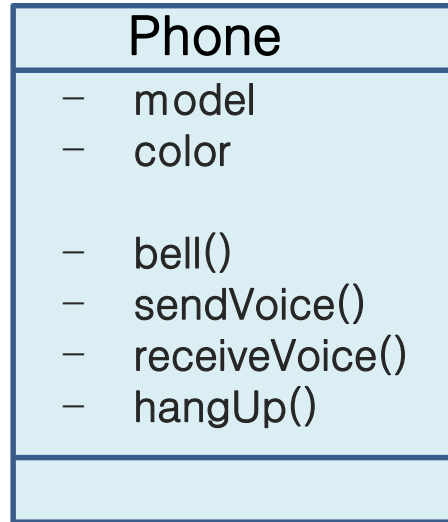
- model, color
- bell()
- sendVoice()
- receiveVoice()
- hangUp()
- wifi
- isWifi()
- setWifi()
- internet()

CordlessPhone

- model, color
- bell()
- sendVoice()
- receiveVoice()
- hangUp()
- wireInfo()

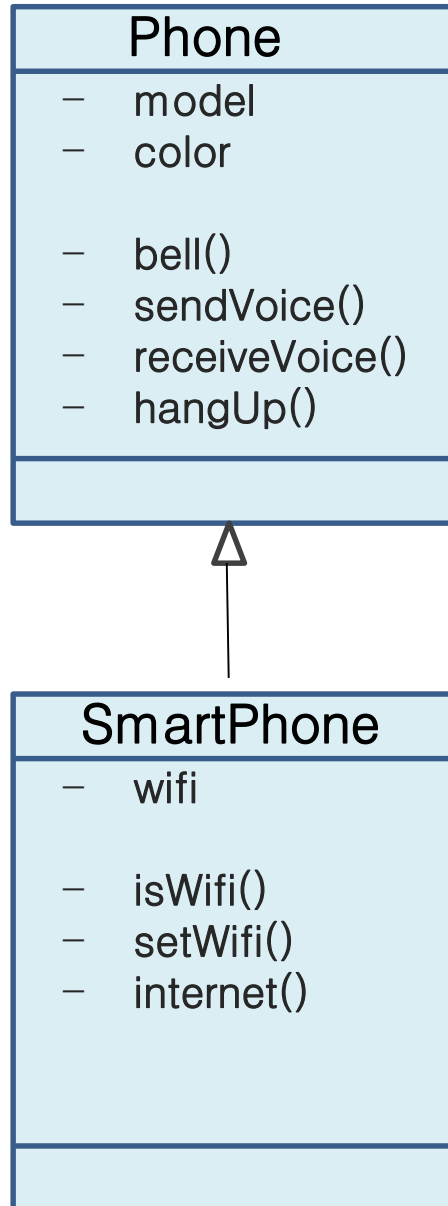
2절. 클래스 상속(extends)

❖ 상속 예제



2절. 클래스 상속(extends)

❖ 상속 예제



2절. 클래스 상속(extends)

❖ 상속 예제

```
public class Phone {  
    //private인 경우에는 상속 불가  
    public String model;  
    public String color;  
  
    public void bell() {  
        System.out.println("벨이 울립니다.");  
    }  
    public void sendVoice(String msg) {  
        System.out.println("나 : " + msg);  
    }  
    public void receiveVoice(String msg) {  
        System.out.println("너 : " + msg);  
    }  
    public void hangUp() {  
        System.out.println("전화를 끊습니다.");  
    }  
}
```

2절. 클래스 상속(extends)

❖ 상속 예제

```
public class SmartPhone extends Phone {  
    private boolean wifi;  
    public SmartPhone(String model, String color) {  
        this.model = model;  
        this.color = color;  
    }  
  
    public boolean isWifi() {  
        return wifi;  
    }  
    public void setWifi(boolean wifi) {  
        this.wifi = wifi;  
        System.out.println("wifi 상태 변경");  
    }  
    public void internet() {  
        System.out.println("인터넷 연결");  
    }  
}
```

2절. 클래스 상속(extends)

❖ 상속 예제

```
public class SmartPhoneEx {  
    public static void main(String[] args) {  
        SmartPhone mine = new SmartPhone("갤럭시", "white");  
  
        System.out.println("모델 : " + mine.model);  
        System.out.println("색상 : " + mine.color);  
  
        System.out.println("wifi : " + mine.isWifi());  
  
        mine.bell();  
        mine.sendVoice("Hello");  
        mine.receiveVoice("홍길동입니다");  
        mine.hangUp();  
  
        mine.setWifi(true);  
        mine.internet();  
    }  
}
```

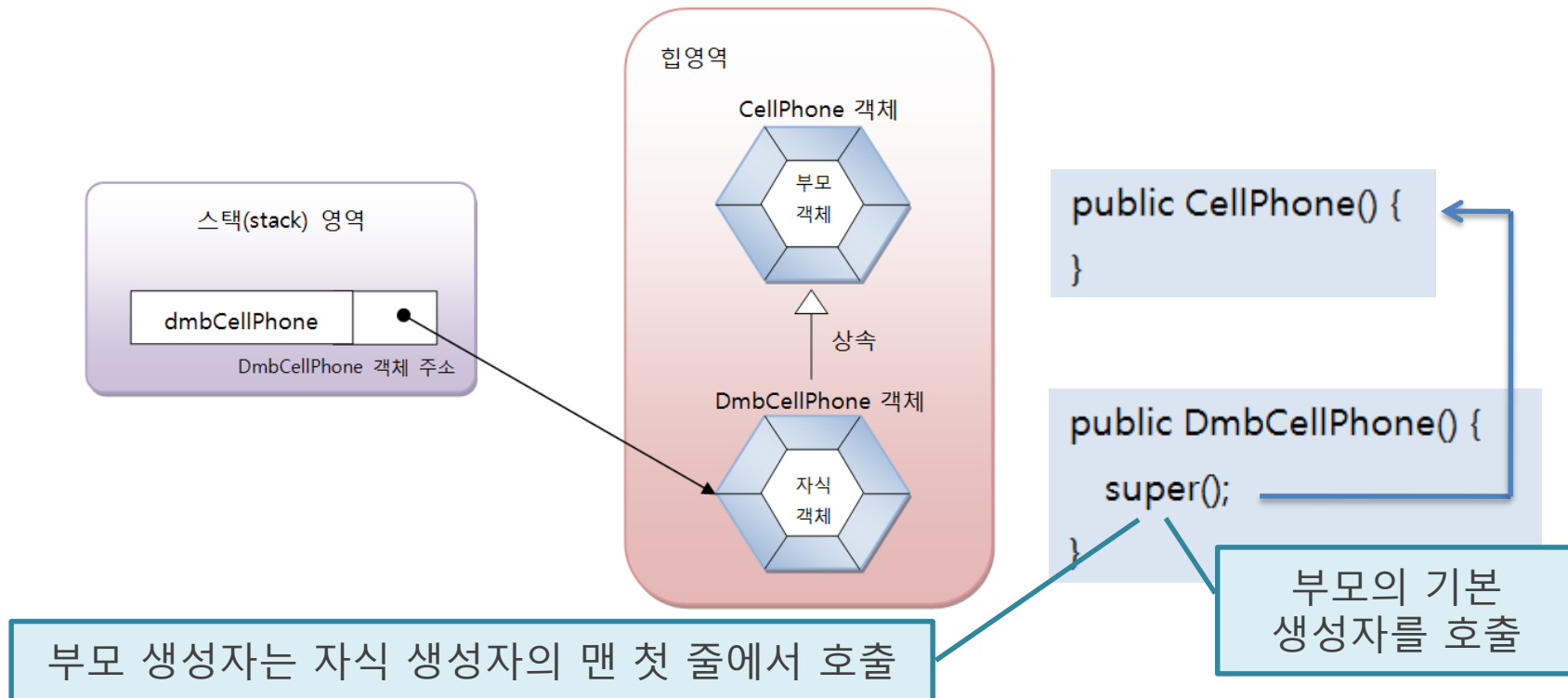

3절. 부모 생성자 호출(super(...))

❖ 자식 객체 생성하면 부모 객체도 생성되는가?

■ 부모 없는 자식 없음

- 자식 객체 생성할 때는 부모 객체부터 생성 후 자식 객체 생성
- 부모 생성자 호출 완료 후 자식 생성자 호출 완료

```
DmbCellPhone dmbCellPhone = new DmbCellPhone();
```



3절. 부모 생성자 호출(super(...))

❖ 부모 생성자 & 자식 생성자 호출 순서 예제

```
public class Phone {  
    //private인 경우에는 상속 불가  
    public String model;  
    public String color;  
  
    public Phone() {  
        System.out.println("Phone() 생성자 호출");  
    }  
}
```

```
public class SmartPhone extends Phone {  
    private boolean wifi;  
    public SmartPhone(String model, String color) {  
        this.model = model;  
        this.color = color;  
        System.out.println("SmartPhone(String model, String color) 생성  
    }  
}
```

3절. 부모 생성자 호출(super(...))

❖ 명시적인 부모 생성자 호출

- 부모 객체 생성할 때, 부모 생성자 선택해 호출

```
자식클래스( 매개변수선언, ... ) {  
    super( 매개값, ... );  
    ...  
}
```

- **super(매개값,...)**
 - 매개값과 동일한 타입, 개수, 순서 맞는 부모 생성자 호출
- 부모 생성자 없다면 컴파일 오류 발생
- 반드시 자식 생성자의 첫 줄에 위치
- 부모 클래스에 기본(매개변수 없는) 생성자가 없다면 필수 작성

3절. 부모 생성자 호출(super(...))

❖ 명시적인 부모 생성자 호출 예제

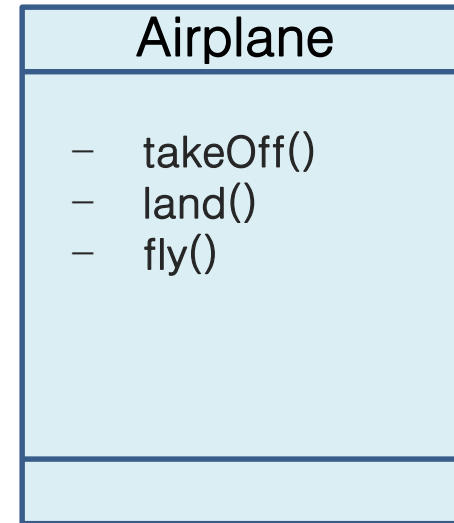
```
public class Phone {  
    //private인 경우에는 상속 불가  
    public String model;  
    public String color;  
  
    public Phone(String model, String color) {  
        System.out.println("Phone(String model, String color) 생성자 호출");  
    }  
}
```

```
public class SmartPhone extends Phone {  
    private boolean wifi;  
    public SmartPhone(String model, String color) {  
        super(model, color);  
        // this.model = model;  
        // this.color = color;  
        System.out.println("SmartPhone(String model, String color) 생성자 호출");  
    }  
}
```

3절. 클래스 상속 예제

❖ 부모 클래스 – Airplane

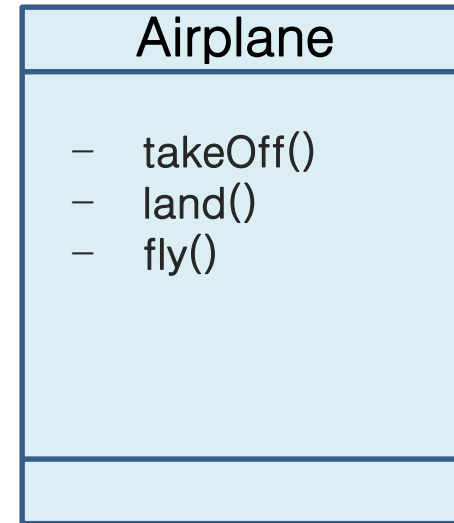
- takeOff()
- land()
- fly()



3절. 클래스 상속 예제

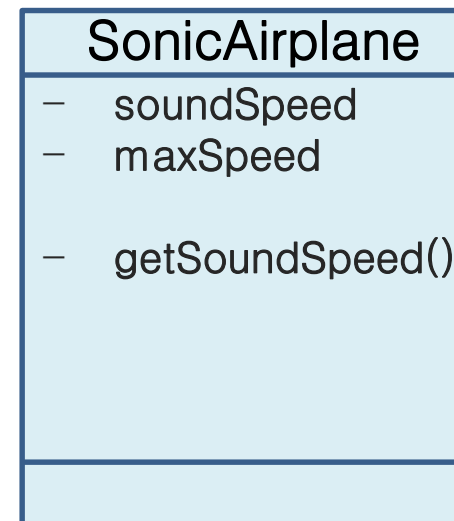
❖ 부모 클래스 – Airplane

- takeOff()
- land()
- fly()



❖ 자식 클래스 – SonicAirplane

- soundSpeed
- maxSpeed
- getSoundSpeed()



3절. 클래스 상속 예제

❖ 부모 클래스 작성(Airplane.java)

```
Airplane.java  SonicAirplane.java  AirplaneEx.java
1  package week10;
2
3  public class Airplane {
4
5
6
7  public void land() {
8      System.out.println("착륙합니다");
9  }
10
11 public void fly() {
12     System.out.println("일반비행합니다");
13 }
14
15 public void takeOff() {
16     System.out.println("이륙합니다");
17 }
18 }
```

3절. 클래스 상속 예제

- ❖ 자식 클래스 작성(SonicAirplane.java)
- ❖ 실행 클래스 작성(AirplaneEx.java)

```
package week10;

public class SonicAirplane extends Airplane {
    int soundSpeed;
    int maxSpeed;

    int getSoundSpeed() {
        return soundSpeed;
    }
}
```

```
Airplane.java  SonicAirplane.java  AirplaneEx.java ✕
1  package week10;
2
3  public class AirplaneEx {
4      public static void main(String[] args) {
5          SonicAirplane sPlane = new SonicAirplane();
6          sPlane.takeOff();
7          sPlane.fly();
8          sPlane.land();
9
10     }
11 }
```


3절. 클래스 상속 예제

❖ 자식 클래스 작성(SonicAirplane.java)

❖ 실행 클래스 작성(AirplaneEx.java)

```
package week10;

public class SonicAirplane extends Airplane {
    int soundSpeed;
    int maxSpeed;

    int getSoundSpeed() {
        return soundSpeed;
    }
}
```

Console

<terminated> AirplaneEx [Java

이륙합니다

일반비행합니다

착륙합니다

```
Airplane.java  SonicAirplane.java  AirplaneEx.java
1 package week10;
2
3 public class AirplaneEx {
4     public static void main(String[] args) {
5         SonicAirplane sPlane = new SonicAirplane();
6         sPlane.takeOff();
7         sPlane.fly();
8         sPlane.land();
9     }
10 }
11 }
```

4절. 메소드 재정의(Override)

❖ 메소드 재정의(@Override)

- 부모 클래스의 상속 메소드 수정해 자식 클래스에서 재정의하는 것
- 메소드 재정의 조건
 - 부모 클래스의 메소드와 동일한 시그니처를 가져야 한다.
 - 리턴 타입, 메소드 이름, 매개변수 리스트
 - 접근 제한을 더 강하게 오버라이딩 불가
 - public을 default나 private으로 수정 불가
 - 반대로 default는 public 으로 수정 가능
 - 새로운 예외(Exception) throws 불가

4절. 메소드 재정의(Override)

❖ @Override 어노테이션

- 컴파일러에게 부모 클래스의 메소드 선언부와 동일한지 검사 지시
- 정확한 메소드 재정의의 위해 붙여주면 OK

❖ 메소드 재정의 효과

- 부모 메소드는 숨겨지는 효과 발생
 - 재정의된 자식 메소드 실행

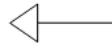
4절. 메소드 재정의(Override)

❖ 부모 메소드 사용(super)

- 메소드 재정의는 부모 메소드 숨기는 효과 !!
 - 자식 클래스에서는 재정의된 메소드만 호출
- 자식 클래스에서 수정되기 전 부모 메소드 호출 - super 사용
 - super는 부모 객체 참조(참고: this는 자신 객체 참조)

```
super.부모메소드();
```

```
class Parent {  
    void method1() { ... }  
    void method2() { ... }  
}
```



상속

부모 메소드 호출

```
class Child extends Parent {  
    void method2() { ... } //Overriding  
    void method3() {  
        method2();  
        super.method2();  
    }  
}
```

재정의된 호출

4절. 메소드 재정의(Override)

❖ 부모 메소드 사용(super)

```
SonicAirplane.java ✖
1 package week10;
2
3 public class SonicAirplane extends Airplane {
4     public static final int NORMAL = 1;
5     public static final int SUPERSONIC = 2;
6
7     int flyMode = NORMAL;
8
9     @Override
10    public void fly() {
11        if (flyMode == SUPERSONIC)
12            System.out.println("음속비행합니다");
13        else
14            super.fly();
15    }
16 }
```

4절. 메소드 재정의(Override)

❖ 부모 메소드 사용(super)

```
AirplaneEx2.java ✖
1 package week10;
2
3 public class AirplaneEx2 {
4     public static void main(String[] args) {
5         SonicAirplane sPlane = new SonicAirplane();
6         sPlane.takeOff();
7         sPlane.fly();
8         sPlane.land();
9
10        sPlane.flyMode = SonicAirplane.SUPERSONIC;
11        sPlane.fly();
12        sPlane.flyMode = SonicAirplane.NORMAL;
13        sPlane.fly();
14    }
15 }
```

4절. 메소드 재정의(Override)

❖ 부모 메소드 사용(super)

```
AirplaneEx2.java ✖
1 package week10;
2
3 public class AirplaneEx2 {
4     public static void main(String[] args) {
5         SonicAirplane sPlane = new SonicAirplane();
6         sPlane.takeOff();
7         sPlane.fly();
8         sPlane.land();
9
10        sPlane.flyMode = SonicAirplane.SUPERSONIC;
11        sPlane.fly();
12        sPlane.flyMode = SonicAirplane.NORMAL;
13        sPlane.fly();
14    }
15 }
```

Console ✖

<terminated> AirplaneEx2 [Java]

이륙합니다
일반비행합니다
착륙합니다
음속비행합니다
일반비행합니다

5절. final 클래스와 final 메소드

❖ final 키워드의 용도

- final 필드: 수정 불가 필드
- final 클래스: 부모로 사용 불가한 클래스
- final 메소드: 자식이 재정의할 수 없는 메소드

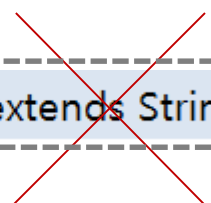
❖ 상속할 수 없는 final 클래스

- 자식 클래스 만들지 못하도록 final 클래스로 생성

```
public final class 클래스 { ... }
```

```
public final class String { .. }
```

```
public class NewString extends String { ... }
```



❖ 오버라이딩 불가한 final 메소드

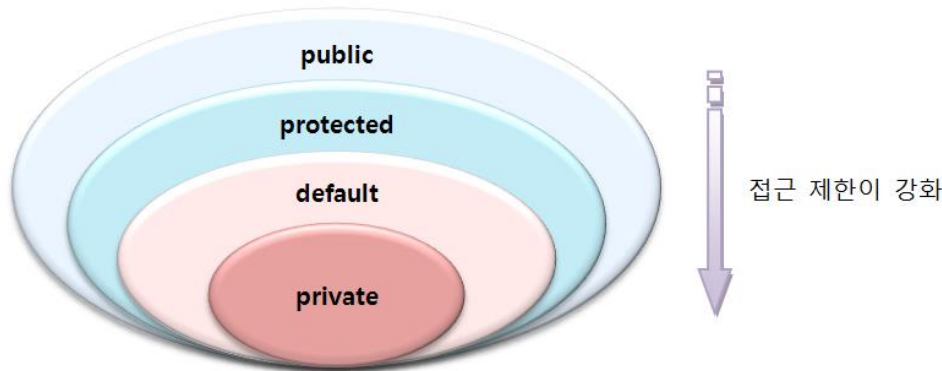
- 자식 클래스가 재정의 못하도록 부모 클래스의 메소드를 final로 생성

6절. protected 접근 제한자

❖ protected 접근 제한자

■ 상속과 관련된 접근 제한자

- 같은 패키지: default와 동일
- 다른 패키지: 자식 클래스만 접근 허용



접근 제한	적용할 내용	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스

6절. protected 접근 제한자

❖ protected 접근 제한자

■ 상속과 관련된 접근 제한자

- 같은 패키지: default와 동일

```
package package1;

public class A {
    protected String field;

    protected A() {
    }

    protected void method() {
    }
}
```

가능

```
package package1;

public class B {
    public void method() {
        A a = new A();           // (o)
        a.field = "value";       // (o)
        a.method();              // (o)
    }
}
```

불가능

```
package package2;
import package1.A;

public class C {
    public void method() {
        A a = new A();           // (x)
        a.field = "value";       // (x)
        a.method();              // (x)
    }
}
```

6절. protected 접근 제한자

❖ protected 접근 제한자

- 상속과 관련된 접근 제한자
 - 다른 패키지: 자식 클래스만 접근 허용

```
package package1;

public class A {
    protected String field;

    protected A() {
    }

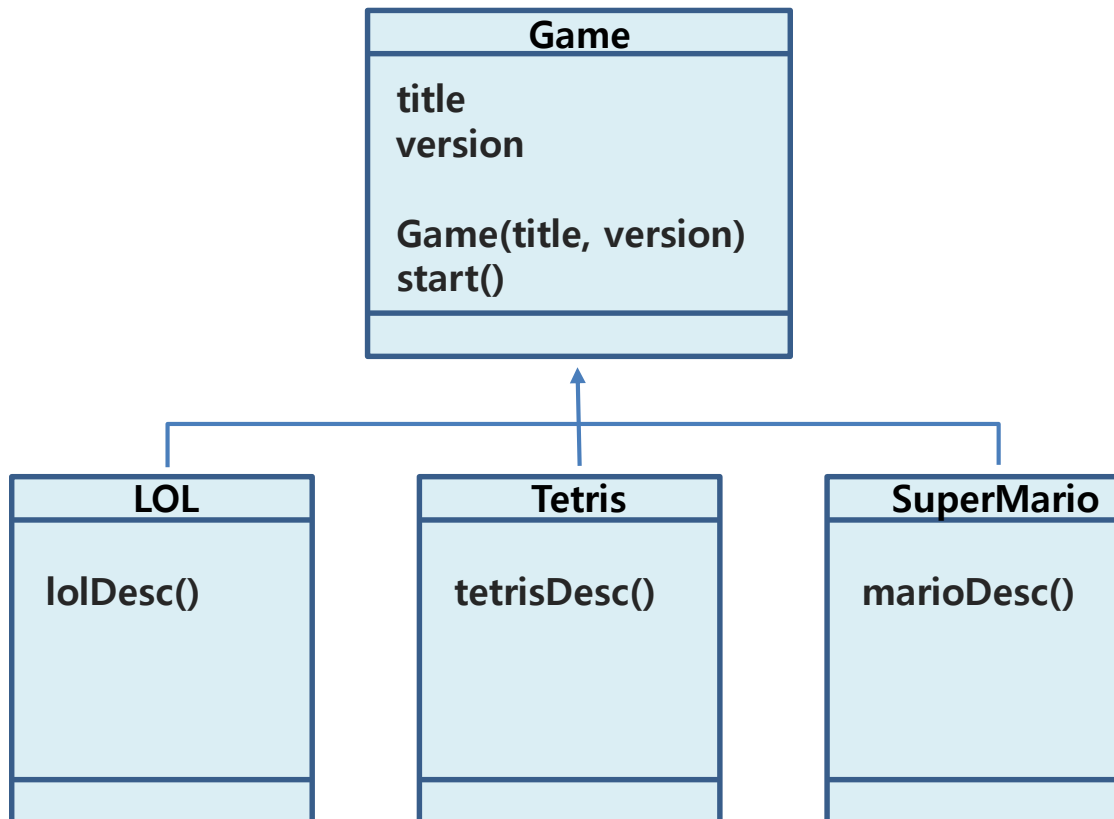
    protected void method() {
    }
}
```

```
package package2;
import package1.A;

public class D extends A {
    public D() {
        super();                // (o)
        this.field = "value";   // (o)
        this.method();          // (o)
    }
}
```

❖ 게임 클래스 만들기

- 다음 3개의 게임 클래스를 만들어 보자.
- 공통된 특징을 부모 클래스(Game)로 만들어서 상속받도록 한다.
- Game 클래스는 매개변수가 있는 생성자를 가진다.
- <결과화면>을 참조해서 나머지 코드를 완성하시오 (GameEx.java).



❖ 게임 클래스 만들기

- 다음 3개의 게임 클래스를 만들어 보자.
- 공통된 특징을 부모 클래스(Game)로 만들어서 상속받도록 한다.
- Game 클래스는 매개변수가 있는 생성자를 가진다.
- <결과화면>을 참조해서 나머지 코드를 완성하시오(GameEx.java).

제목 : 롤

버전 : 13.0

롤게임을 시작합니다.

리그 오브 레전드는 세계 최고의 MOBA(Multiplayer Online Battle Arena) 게임입니다.

제목 : 테트리스

버전 : 12.5

테트리스게임을 시작합니다.

테트리스(Tetris)는 퍼즐 게임으로, 소련의 프로그래머 알렉세이 파지트노프가 처음 디자인하고 프로그래밍 한 게임이다.

제목 : 슈퍼마리오

버전 : 15.1

슈퍼마리오게임을 시작합니다.

닌텐도의 대표 비디오 게임 시리즈인 마리오 시리즈의 핵심이 되는 본가 시리즈.