

Computer Language



OOP 2: Method and Inheritance



Agenda

- Method
- Inheritance



Method

Inheritance

Method: Instance Member

- Fields and methods of an object/instance
 - Instance field
 - Instance method
- Instance members **belong to an object/instance**
- Therefore, instance members cannot be used without object instantiation!

```
public class Car {  
    // field  
    int gas;  
  
    // method  
    void setSpeed(int speed) { ... }  
}
```



```
Car myCar = new Car();  
myCar.gas = 10;  
myCar.setSpeed(60);  
  
Car yourCar = new Car();  
yourCar.gas = 20;  
yourCar.setSpeed(80);
```

Method: Static Member

- Fields and methods of a class
 - Static field, static method
 - Sometimes called class members
- Static members **belong to a class**
- Therefore, static members can be used without object instantiation!
- Declaration
 - Use **static** keyword for the members!

```
public class Calculator {  
    static double pi = 3.14159;  
    static int plus(int x, int y) { ... }  
    static int minus(int x, int y) { ... }  
}
```

Method: Static Member (cont'd)

■ Instance vs Static members

	Instance member	Static member
Declaration	<pre>class Sample{ int n; void g(){...} }</pre>	<pre>class Sample{ static int n; static void g(){...} }</pre>
Where?	for each object/instance	for a single Class - class members - static members loaded into method area
When?	Once an object is created After object instantiation, instance members can be used	Once class is loaded Static members can be used without any object instantiation
Sharable?	No Instance members reside in each object	Yes Shared with all objects of the class

Method: Static Member (cont'd)

■ When to use Static members?

➤ Global variable/methods

- Example) Math class (java.lang.Math)
 - All the methods and fields are declared static
 - Without Math object instantiation, we can use all the features of Math class!

```
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();  
    ...  
}
```

```
Math m = new Math(); // Error!  
int n = Math.abs(-5);
```

➤ Sharable members

- All instances of the class can share the static members

Method: Static Member (con

■ Example)

- Use of static field
- Use of static method
- What happens we modify static fields?

```
public class Car { // Car class
```

```
String company;  
String model;  
static int maxSpeed = 150;
```

```
Car(String company, String model) {  
    this.company = company;  
    this.model = model;  
}
```

```
static void bomb(){  
    System.out.println("destroyed");  
}
```

```
public static void main(String[] args) {
```

```
    Car myCar = new Car("my", "my");  
    Car yourCar = new Car("you", "you");  
    System.out.println(myCar.company+": "+myCar.maxSpeed);  
    System.out.println(yourCar.company+": "+yourCar.maxSpeed);
```

```
    Car.maxSpeed = 200;  
    System.out.println(myCar.company+": "+myCar.maxSpeed);  
    System.out.println(yourCar.company+": "+yourCar.maxSpeed);
```

```
    myCar.maxSpeed = 300;  
    System.out.println(myCar.company+": "+myCar.maxSpeed);  
    System.out.println(yourCar.company+": "+yourCar.maxSpeed);
```

```
    Car.bomb();
```

```
}
```


Method: Static Member (cont'd)

■ Restrictions

- **Instance** members cannot be used in a static context
- **this** keyword cannot be used in a static context

```
class StaticMethod{  
    int n;  
    void f1(int x){n=x;}  
    void f2(int x){m=x;}  
  
    static int m;  
    static void s1(int x){n=x;}  
    static void s2(int x){f1(3);}  
  
    static void s3(int x){m=x;}  
    static void s4(int x){s3(3);}  
}
```

```
class StaticAndThis {  
    int n;  
    static int m;  
  
    void f1(int x){this.n=x;}  
    void f2(int x){this.m = x;}  
  
    static void s1(int x){this.n = x;}  
    static void s2(int x){this.m = x;}  
}
```

Method: Static Member (cont'd)

- Example 1) Write three static functions (abs, max, and min)

```
class Calc {  
  
}  
  
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
```

Method: Static Member (cont'd)

- Example 2) Write an exchange rate calculator using static members

```
class CurrencyConverter {  
  
}  
public class StaticMember {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        System.out.print("Exchange rate (1$)>> ");  
        double rate = scanner.nextDouble();  
        CurrencyConverter.setRate(rate); // setting exchange rate  
        System.out.println("1M Won is $" + CurrencyConverter.toDollar(1000000));  
        System.out.println("$100 is " + CurrencyConverter.toKWR(100) + "won.");  
        scanner.close();  
    }  
}
```

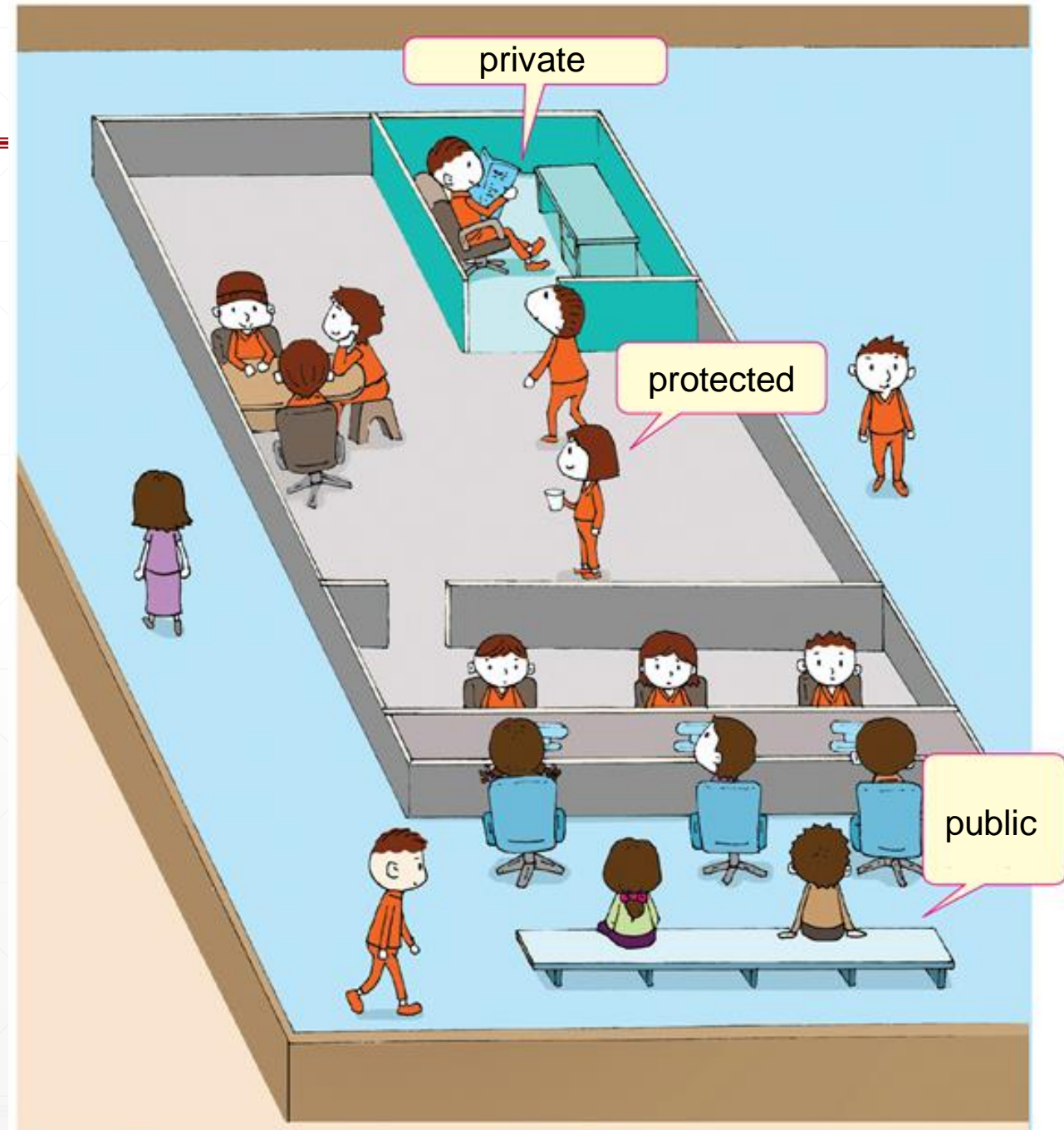
Won to Dollar = Won/rate

$$\text{Dollar to Won} = \text{Dollar} * \text{rate}$$

Exchange rate (1\$)>> 1200
1M Won is \$833.3333333333334
\$100 is 120000.0won

Method: Access Modifier

- Determine who can access!



Method: Access Modifier (cont'd)

■ Java Package

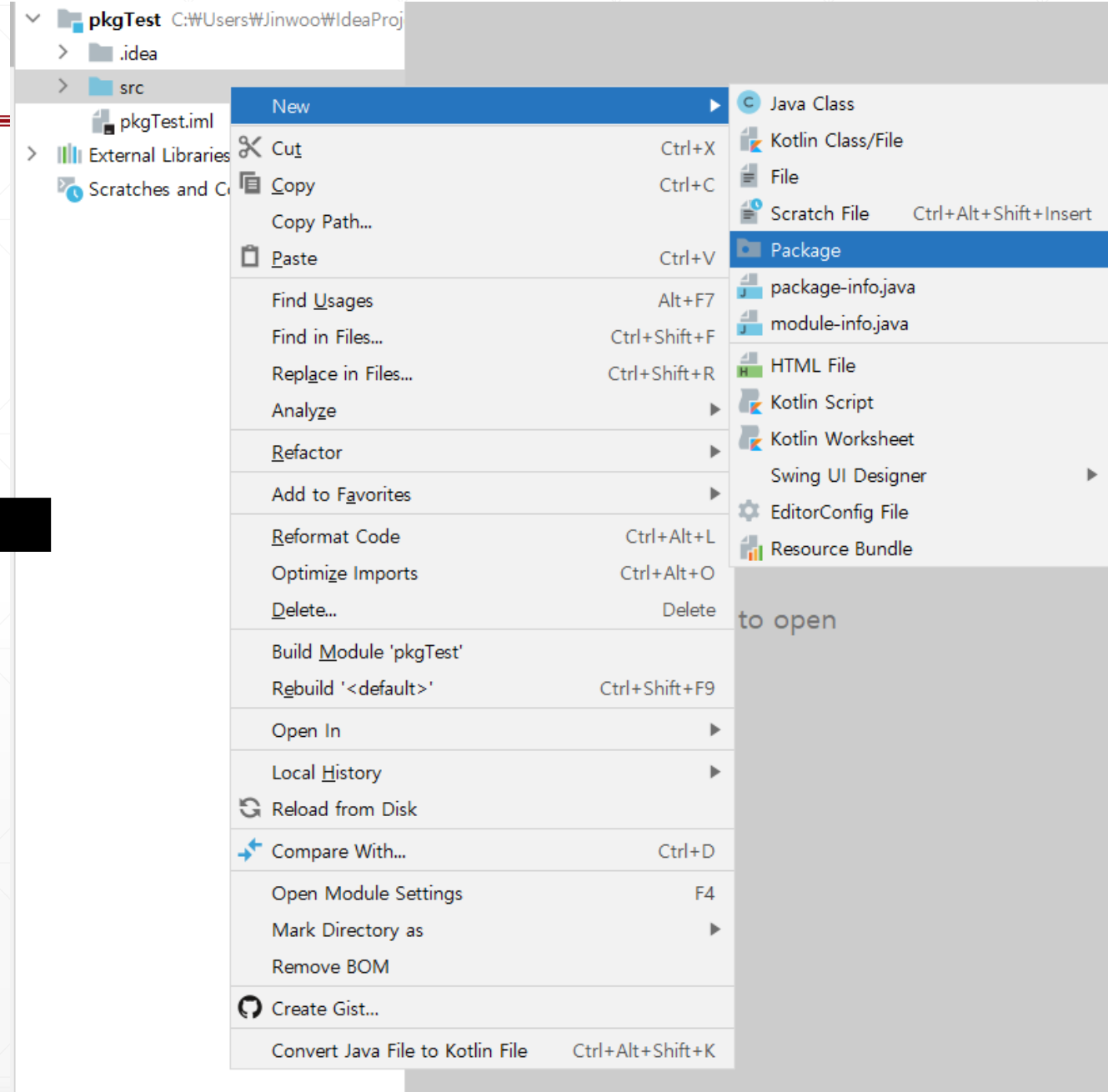
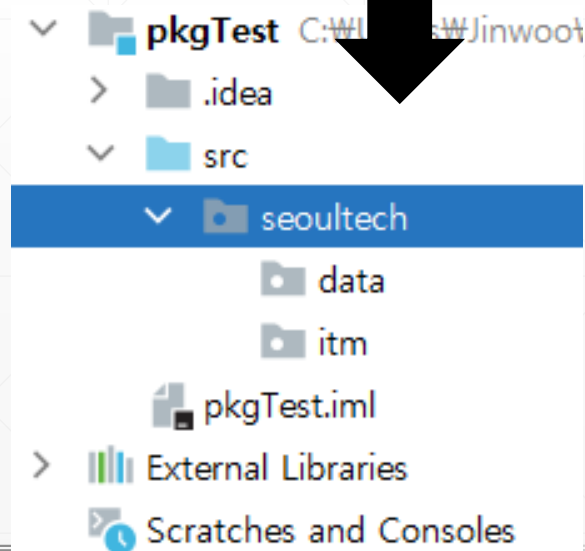
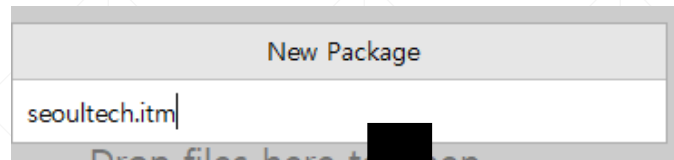
- A group of classes with similar features/categories
 - Similar to “folder” in Filesystem to manage files
 - Class name is composed of
 - Package names (hierarchy)
 - Class name
 - Class name can be uniquely identified by its package names
 - superPkg.subPkg1.myClass
 - superPkg.subPkg2.myClass
- These two classes are different!

Method: Access Modifier

■ Java Package

➤ How to create a package?

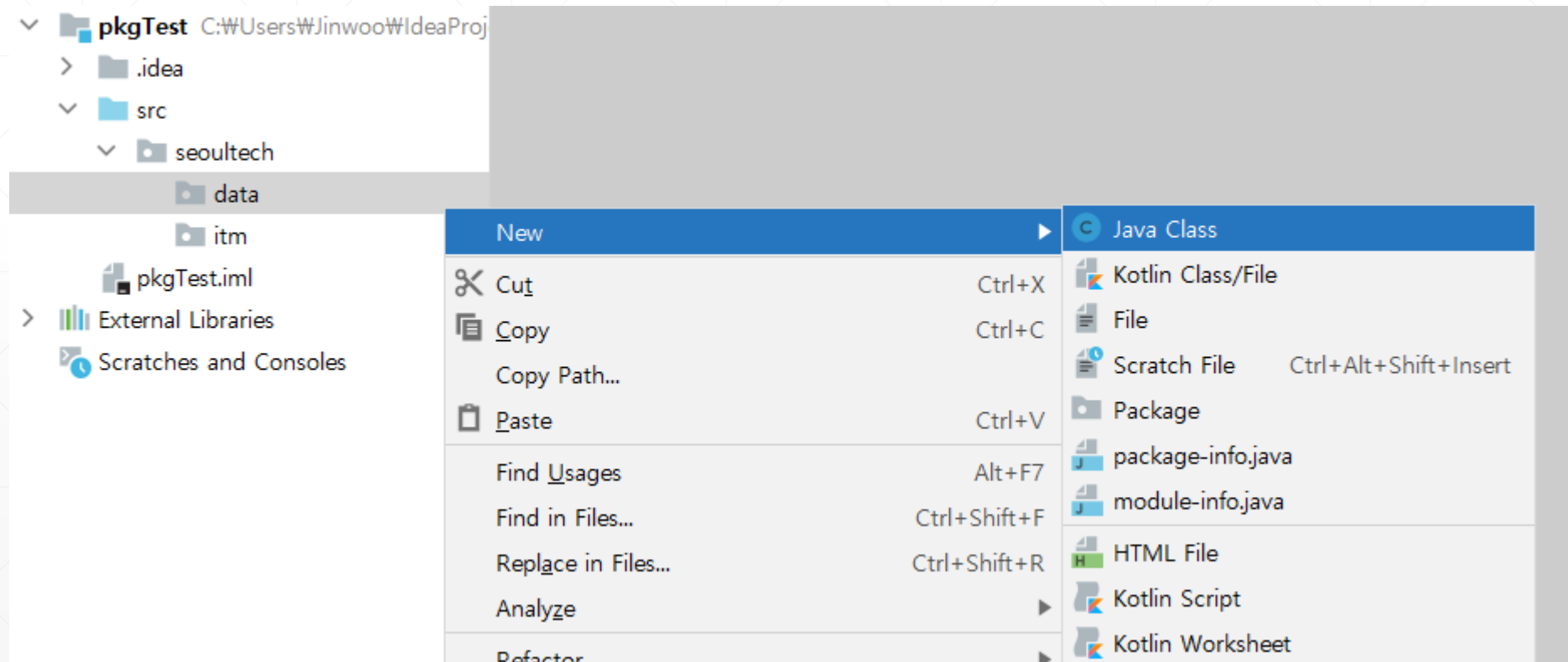
- src → new → package
- Choose your package name



Method: Access Modifier (cont'd)

■ Java Package

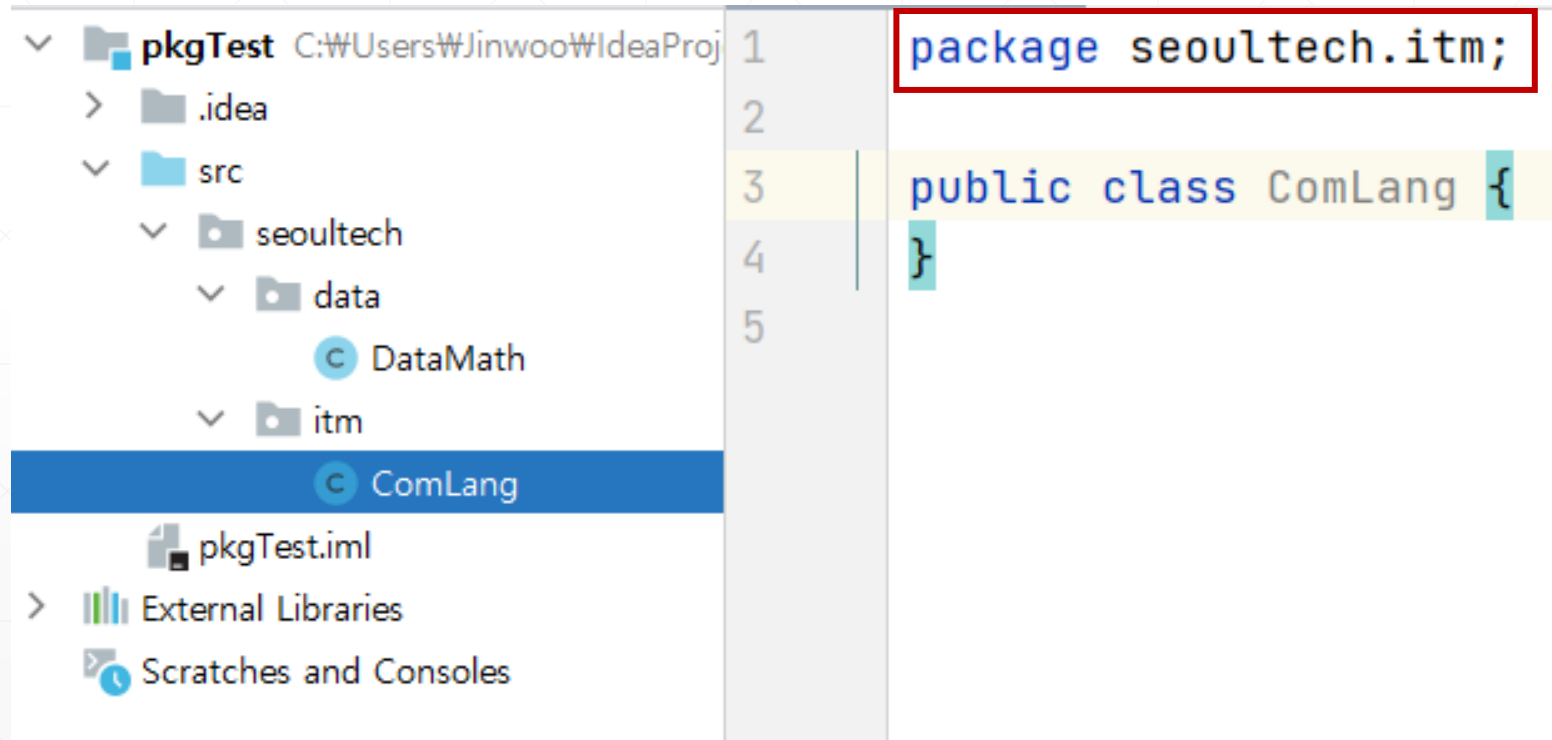
- How to create a class in a package?
 - Create a new class under a specific package



Method: Access Modifier (cont'd)

■ Java Package

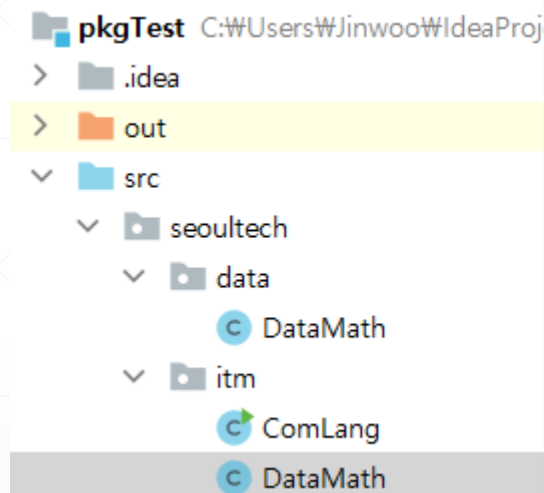
- Package information is added in your class file



Method: Access Modifier (cont'd)

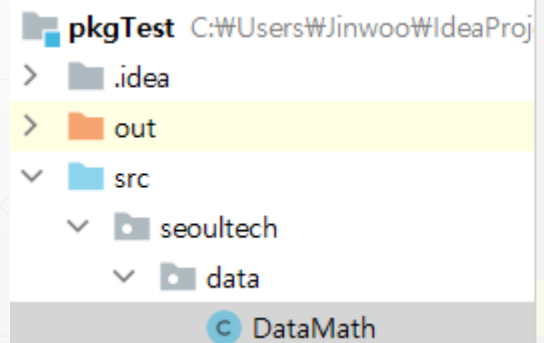
■ Java Package

- We can have multiple packages with the same name, under different packages



```
1 package seoultech.itm;
```

```
2  
3 public class DataMath {  
4     public String msg="ITM's datamath";  
5 }  
6
```



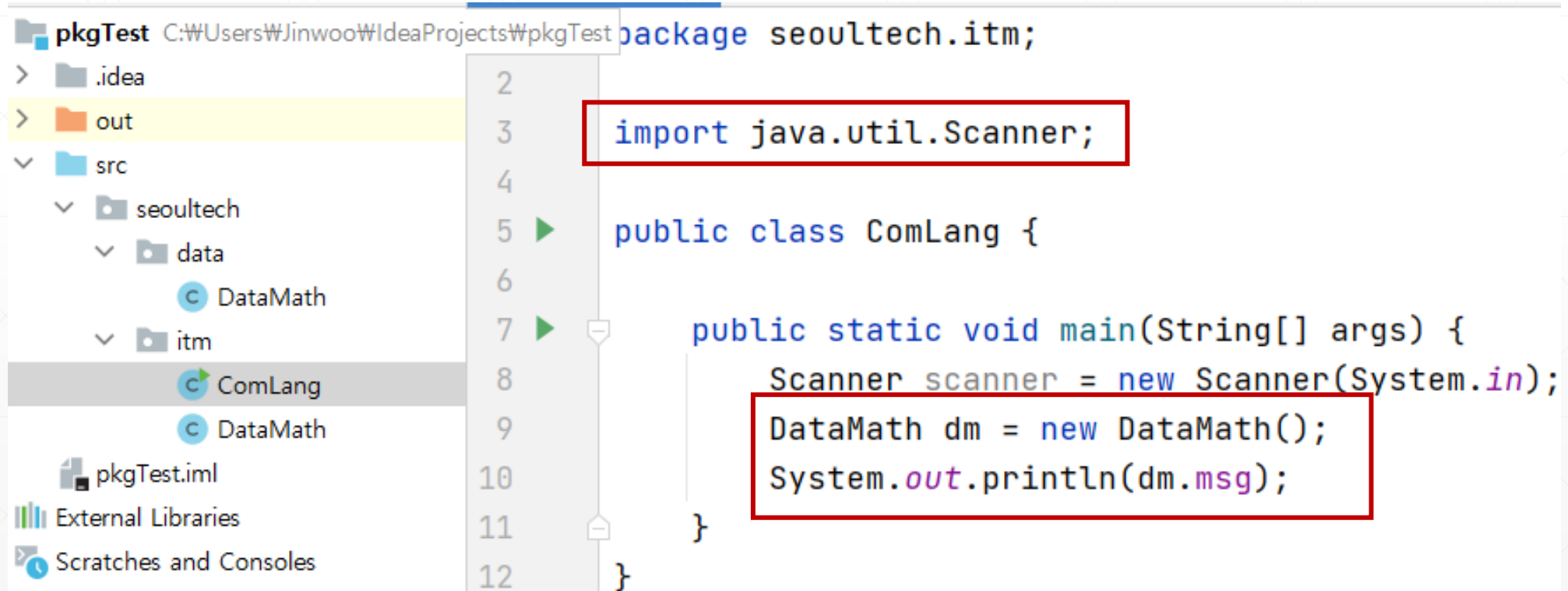
```
1 package seoultech.data;
```

```
2  
3 public class DataMath {  
4     public String msg="Data's datamath";  
5 }  
6
```

Method: Access Modifier (cont'd)

■ Java Package

- To use a class in another package, we need to import it first!
- We can access a certain class in the same package using its class name



```
package seoultech.itm;

import java.util.Scanner;

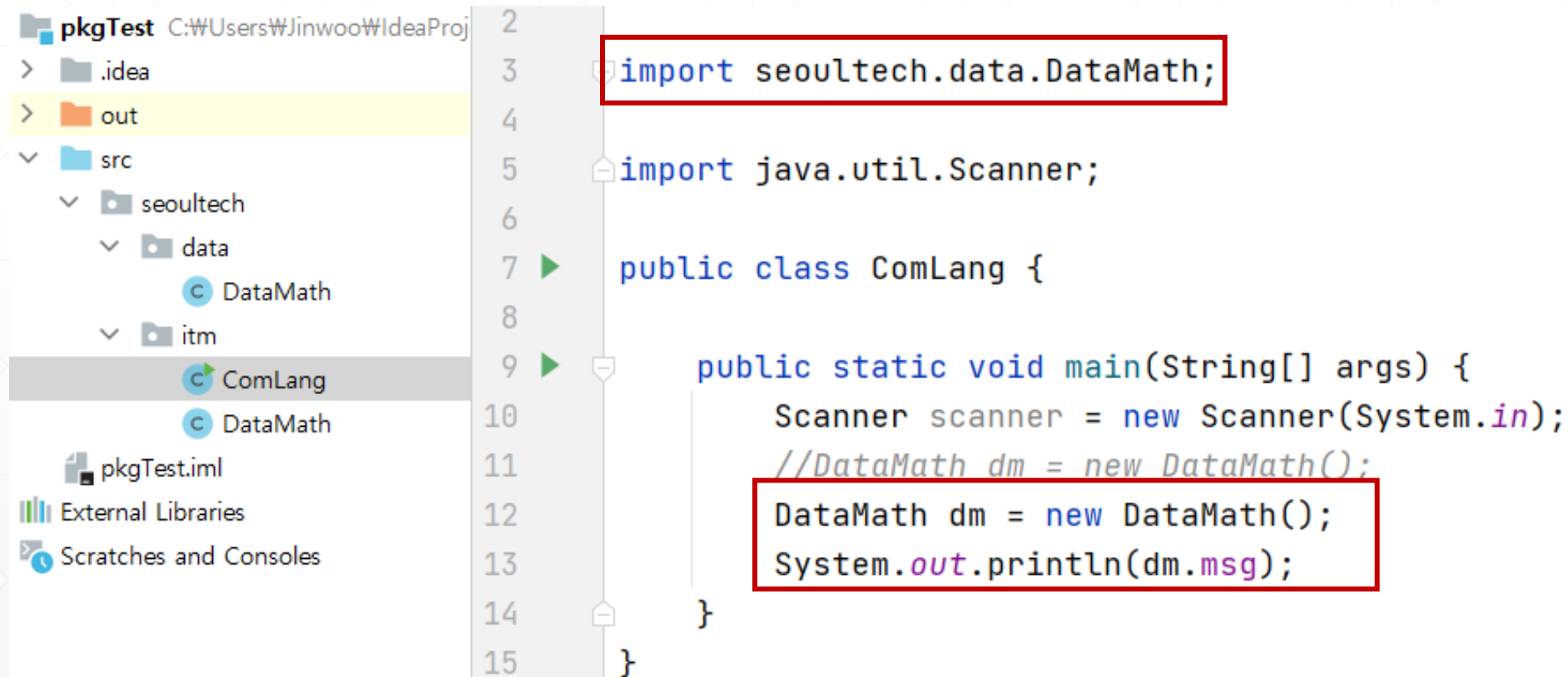
public class ComLang {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        DataMath dm = new DataMath();
        System.out.println(dm.msg);
    }
}
```

Method: Access Modifier (cont'd)

■ Java Package

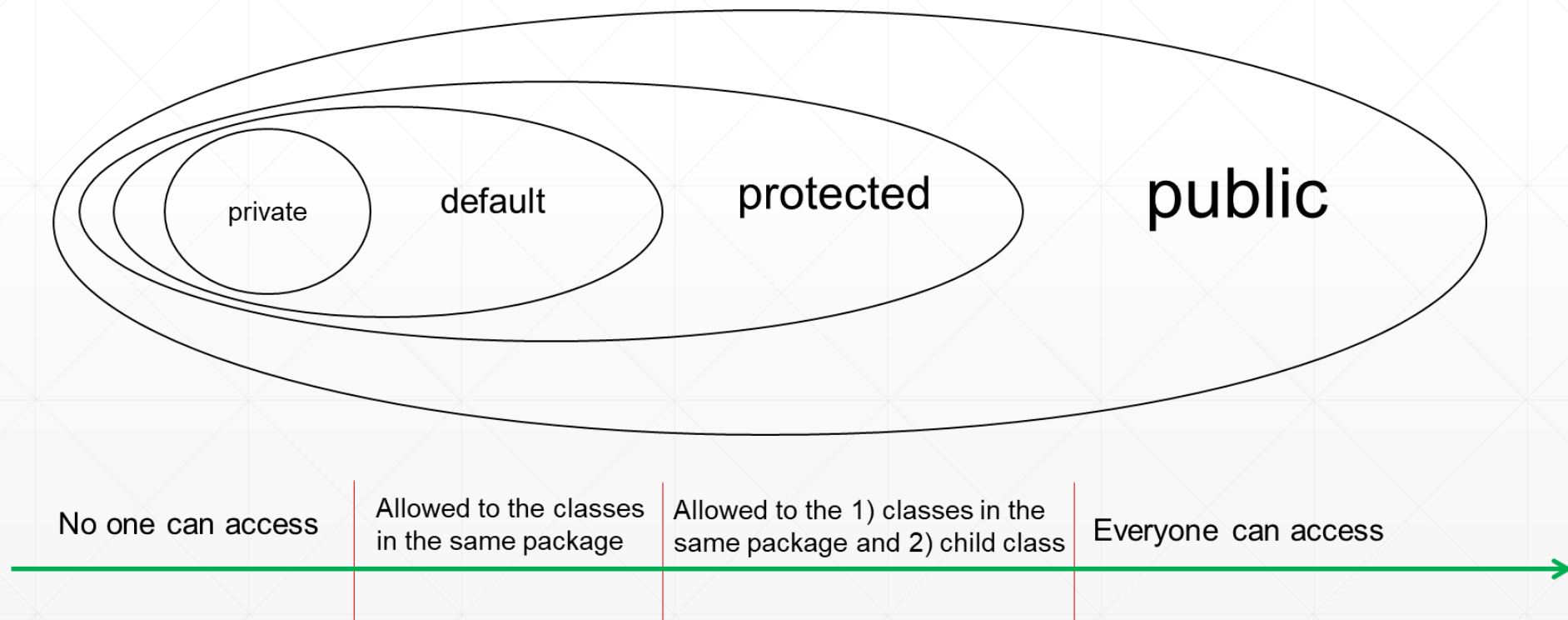
- To use a class in another package, we need to import it first!
- We can access a certain class in the same package using its class name



```
2  
3 import seoultech.data.DataMath;  
4  
5 import java.util.Scanner;  
6  
7 public class ComLang {  
8  
9     public static void main(String[] args) {  
10         Scanner scanner = new Scanner(System.in);  
11         //DataMath dm = new DataMath();  
12         DataMath dm = new DataMath();  
13         System.out.println(dm.msg);  
14     }  
15 }
```

Method: Access Modifier (cont'd)

- Keyword to determine whether other classes can use a particular field or invoke a particular method in the class
 - Related with encapsulation
 - Hide sensitive data, expose publicly available interfaces!



Method: Access Modifier (cont'd)

■ Top-level modifiers

- Public, Default (package-private)
- Determine who can use this **class**

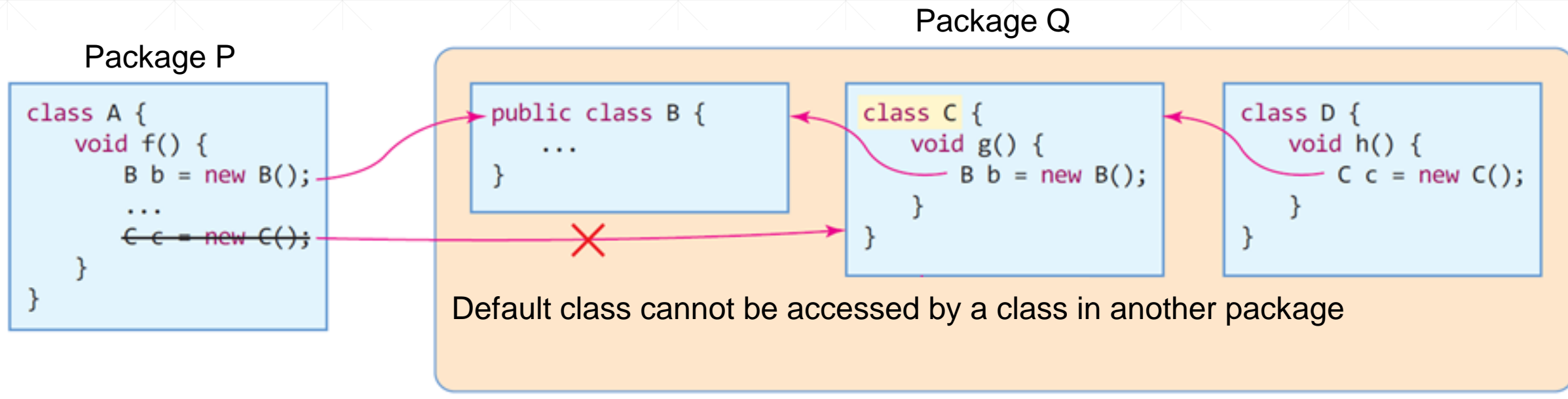
■ Member-level modifiers

- Public, Private, Protected, Default (package-private)
- Determine who can access the **fields and methods** of a class

Method: Access Modifier (cont'd)

■ Top-level access modifier

- Public: all classes can access
- Default (package-private): only the class in the same package can access



Method: Access Modifier (cont'd)

■ Member-level access modifier

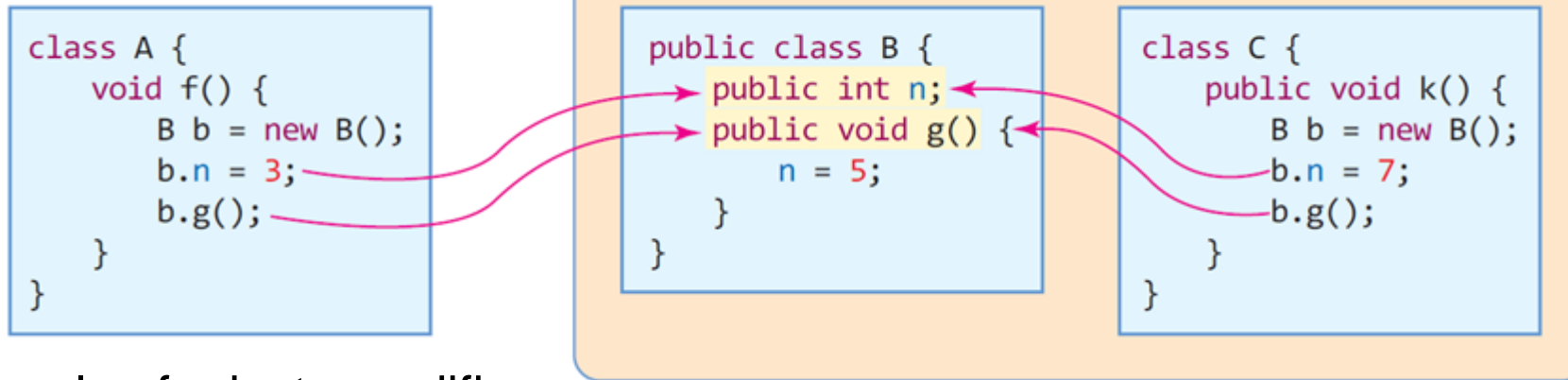
- Public: all classes can access
- Private: no one can access
- Protected:
 - All classes in the same package can access
 - Subclass even in another package can access
- Default (package-private): only the class in the same package can access

Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X O (child)	O

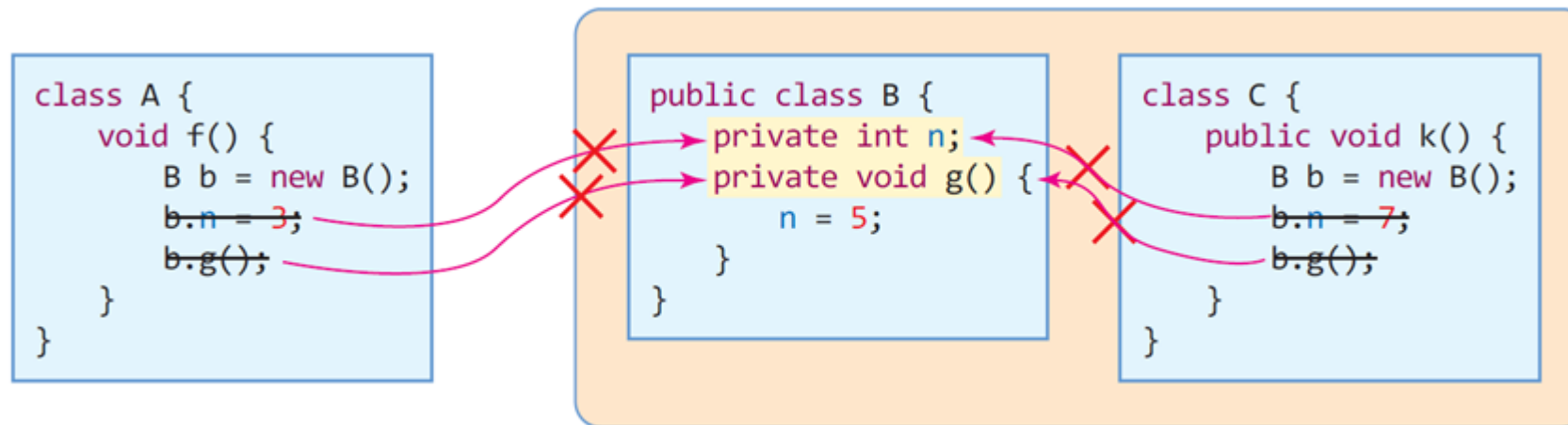
Method: Access Modifier

■ Member-level access modifier

➤ Example of public modifiers



➤ Example of private modifiers



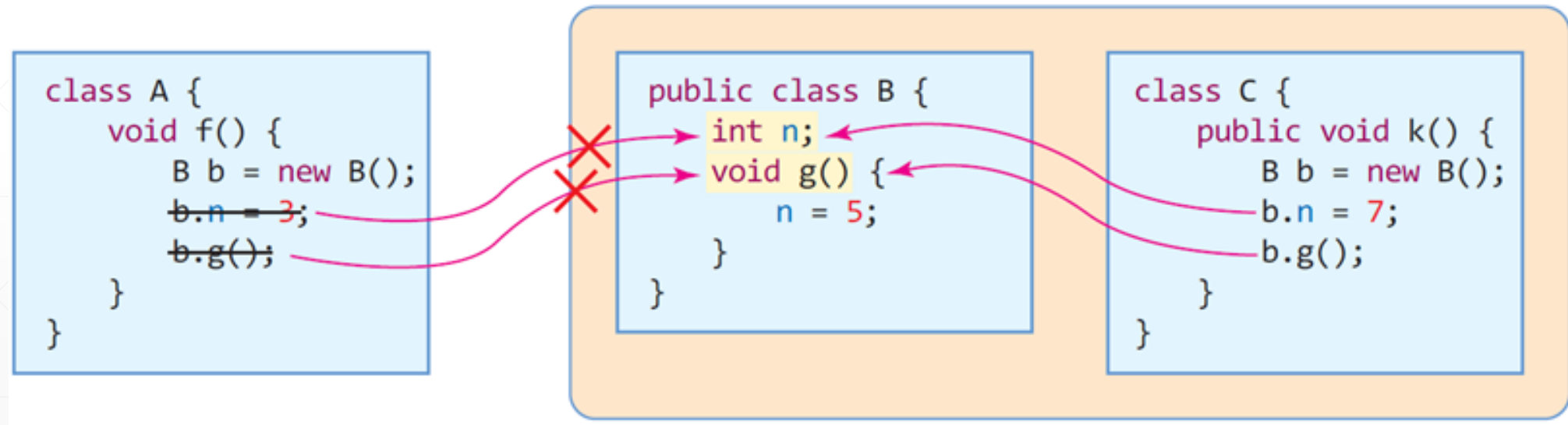
Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X	O

Method: Access Modifier

■ Member-level access modifier

➤ Example of default modifiers

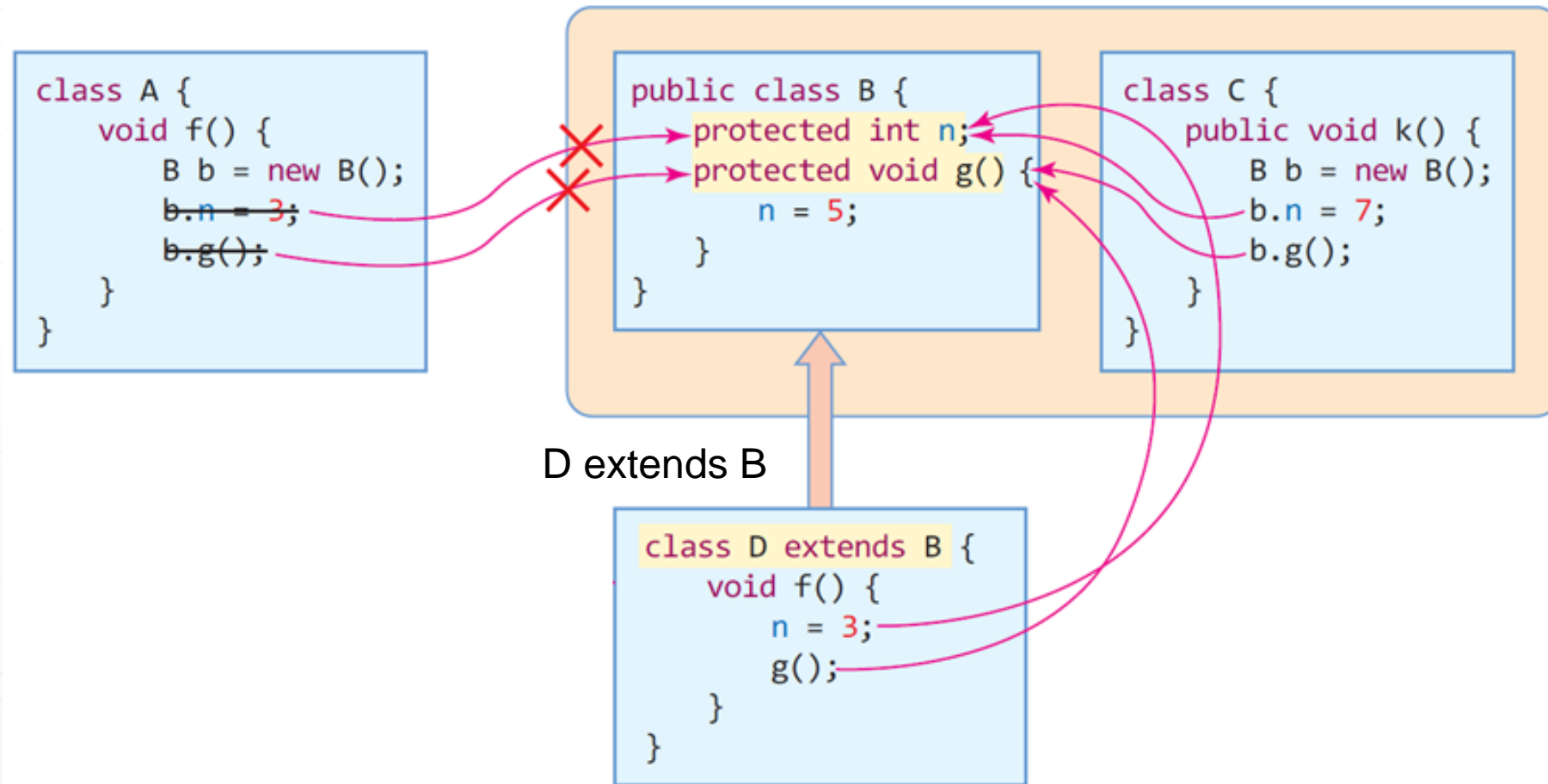
Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X	O



Method: Access Modifier

■ Member-level access modifier

➤ Example of protected modifiers



Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X	O

Method: Access Modifier (cont'd)

■ Example)

- Where a compile error occurs?

■ Hint

- Public: everybody
- Private: no one
- Default: in the same package

```
package seoultech.itm;

class Sample{
    public int publicNum;
    private int privateNum;
    int defaultNum;
}

public class ComLang {

    public static void main(String[] args) {
        Sample mySample = new Sample();
        mySample.publicNum = 10;
        mySample.privateNum = 20;
        mySample.defaultNum = 30;
    }
}
```

Method: Access Modifier (cont'd)

■ One more thing

- Default classes can be included in
 - its own java file, or
 - the java file of the other class
- Public class MUST have its own java file

ComLang.java

```
package seoultech.itm;
```

```
class Sample{  
    public int publicNum;  
    private int privateNum;  
    int defaultNum;  
}
```

```
public class ComLang {  
  
    public static void main(String[] args) {  
        Sample mySample = new Sample();  
        mySample.publicNum = 10;  
        mySample.privateNum = 20;  
        mySample.defaultNum = 30;  
    }  
}
```

Method: Setter and Getter

■ Public member

- Public interface exposed to external accessors
- NEVER (rarely) changed
- Deal with private members for getting/setting values

■ Private member

- Not exposed to external accessors
- Internal use only

```
package seoultech.itm;

class Sample {
    public int publicNum;
    private int privateNum;
    int defaultNum;

    public int getPrivateNum() {
        return privateNum;
    }
    // Getter

    public void setPrivateNum(int privateNum) {
        this.privateNum = privateNum;
    }
    // Setter
}

public class ComLang {
    public static void main(String[] args) {
        Sample mySample = new Sample();
        mySample.publicNum = 10;
        mySample.setPrivateNum(20);
        System.out.println(mySample.getPrivateNum());
        mySample.defaultNum = 30;
    }
}
```



Method

Inheritance

Inheritance: Concept

■ Inheritance in the real world

- Biological nature of parents is inherited to their descendants
- Parent can select who will inherit their wealth

■ Inheritance in the Java world

- Members of a parent class are inherited to their child classes
- Child can select who they wish to inherit!

My Lovely baby~

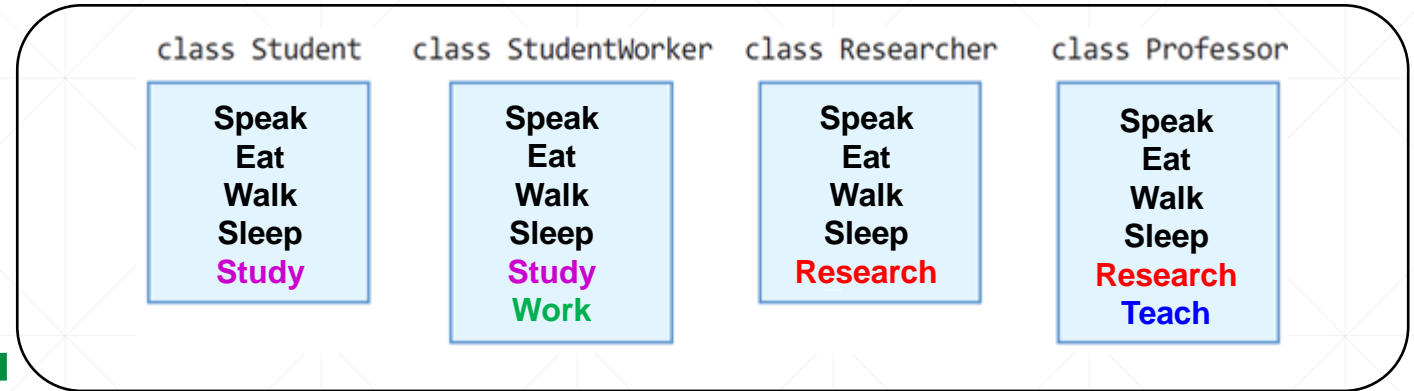


You're right!
They resemble Us!

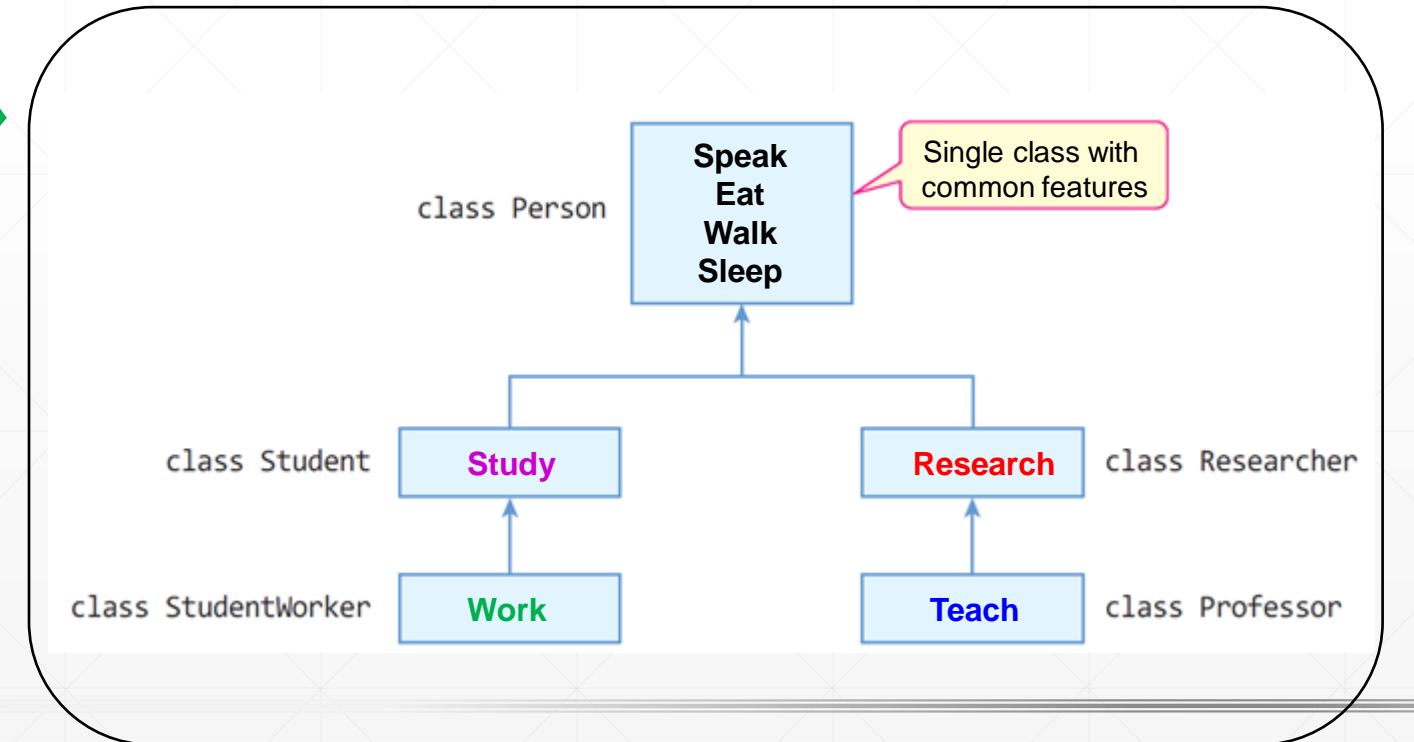
Inheritance: Concept (cont'd)

■ Example of Inheritance

4 Classes with duplicated members/implementations



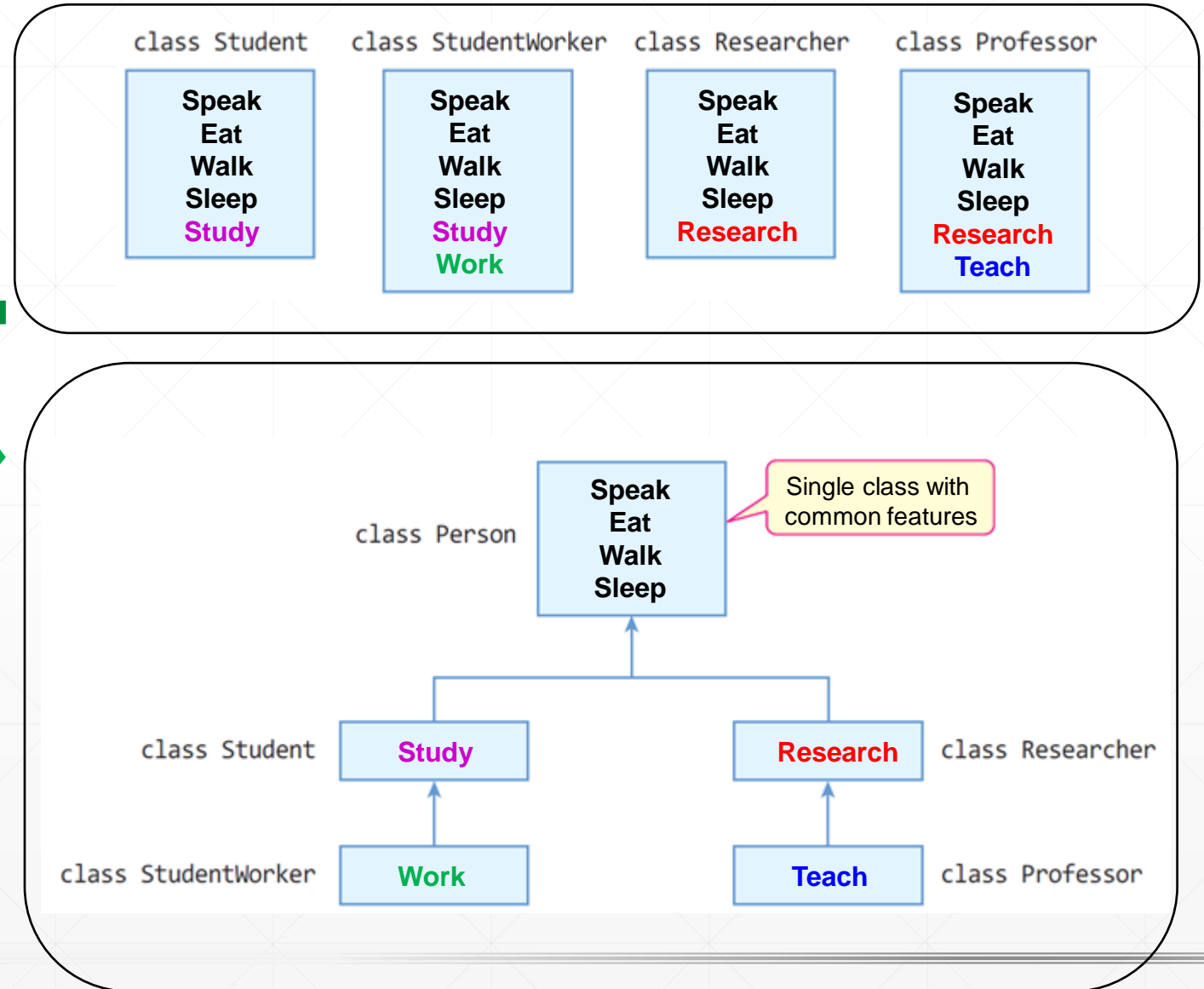
One class with common features
+
Specialized sub-classes without duplicated members



Inheritance: Concept (cont'd)

■ Advantages of Inheritance

- Reduced duplicated codes
- Better maintenance of classes
 - Hierarchical relationships
- Improved productivity
 - Class reuse and extension



Inheritance: Concept (cont'd)

■ Declaration of inheritance: “extends” keyword

```
public class Person {  
    ...  
}  
public class Student extends Person { // declares that Student inherits Person class  
    ...  
}  
public class StudentWorker extends Student { // declares that StudentWorker inherits Student  
    ...  
}
```

■ Sub (child) class

- A class that is derived from another class

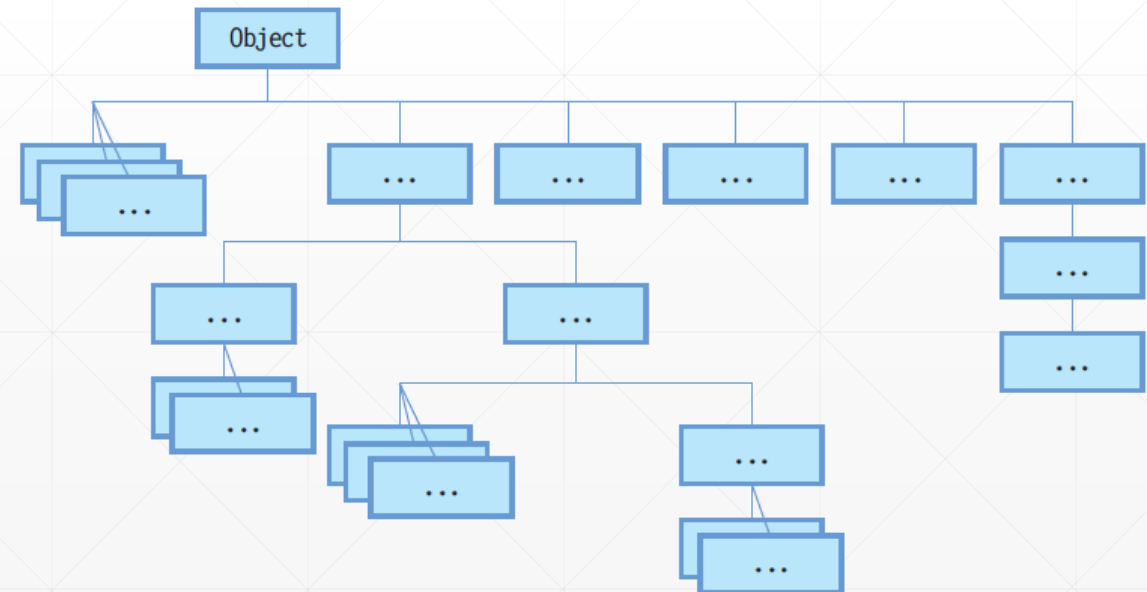
■ Super (parent) class

- A class from which the subclass is derived

Inheritance: Concept (cont'd)

■ Characteristics of Java inheritance

- Does not support multiple inheritance
- No limitation on the number of inheritance
 - E.g) Classes can be derived from classes that are derived from classes that are ..., and so on
- Every class is implicitly a subclass of Object class
 - The root: java.lang.Object
 - Automatically made by Java compiler



Inheritance: Concept (cont'd)

■ Example)

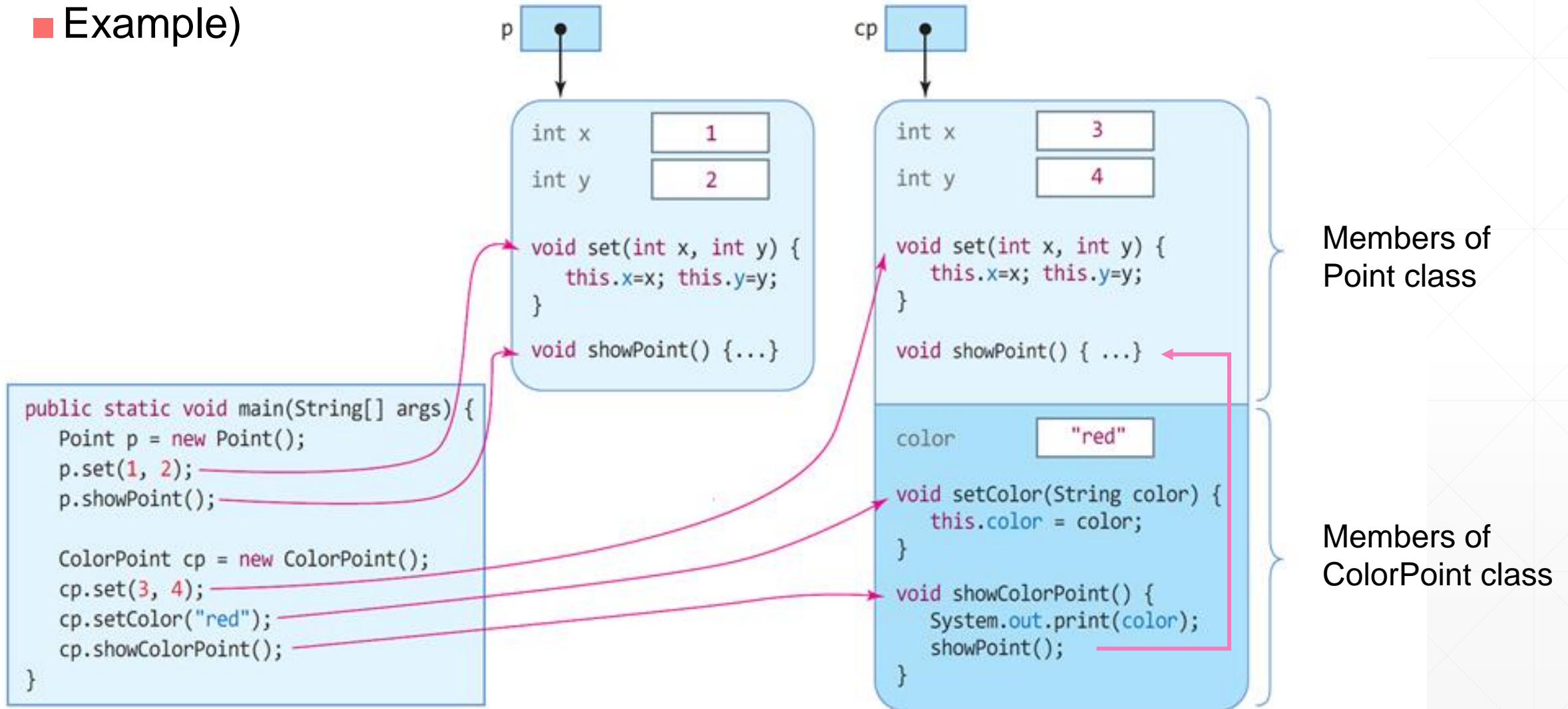
```
class Point {  
    private int x, y;  
    public void set(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void showPoint() {  
        System.out.println("(" + x + ", " + y + ")");  
    }  
}
```

```
// define class ColorPoint that inherits Point class  
class ColorPoint extends Point {  
    private String color;  
    public void setColor(String color) {  
        this.color = color;  
    }  
    public void showColorPoint() {  
        System.out.print(color);  
        showPoint(); // call showPoint() of Point class  
    }  
}
```

```
public class ColorPointEx {  
    public static void main(String [] args) {  
        Point p = new Point(); // instantiate Point object  
        p.set(1, 2); // call set() of Point class  
        p.showPoint();  
  
        ColorPoint cp = new ColorPoint(); // instantiate ColorPoint object  
        cp.set(3, 4); // call set() of Point class  
        cp.setColor("red"); // call setColor() of ColorPoint class  
        cp.showColorPoint();  
    }  
}
```

Inheritance: Concept (cont'd)

■ Example)

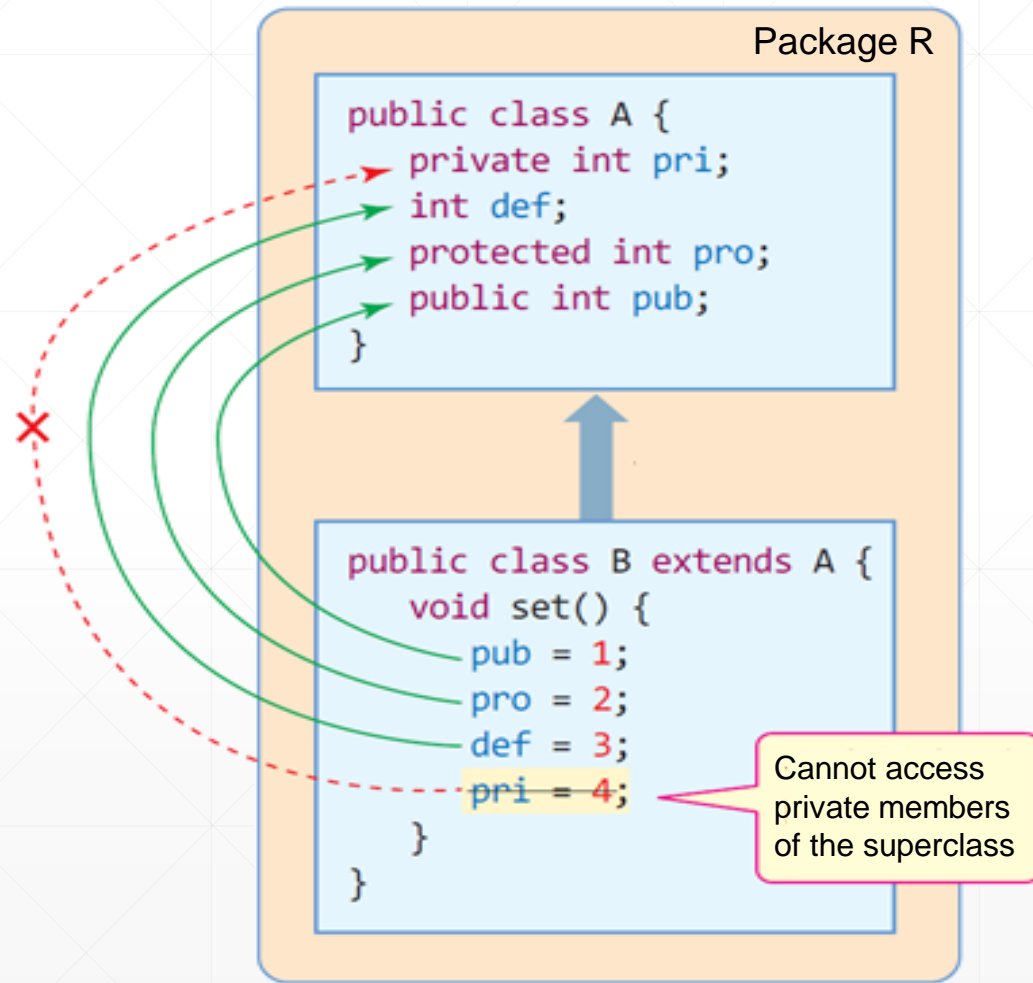


Inheritance: Access Modifier

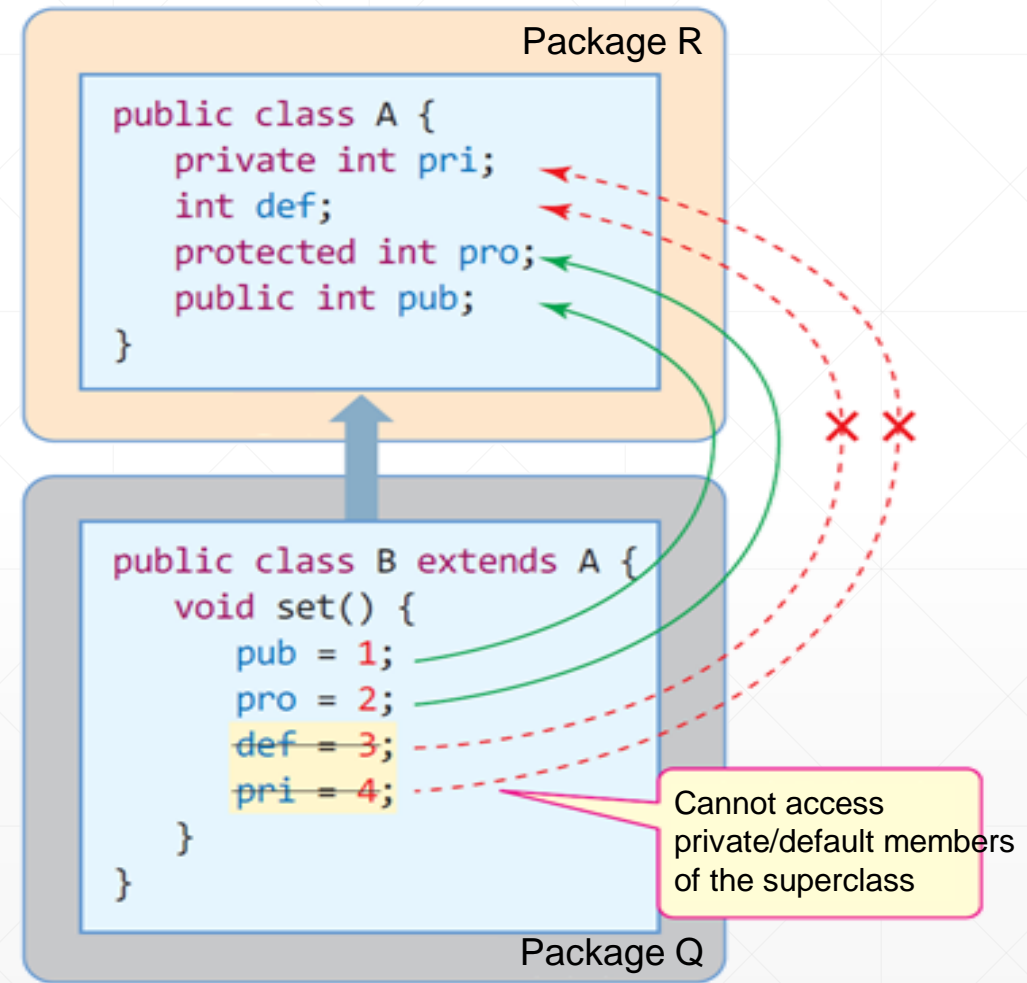
■ Access modifiers in the super class

- Public: all other classes can access these members
- Private
 - No one can access these members
 - Only other members inside the same class allowed
- Protected
 - All classes in the same package can access these members
 - **Child class** outside the package can access these members
- Default
 - All classes in the same package can access these members

Inheritance: Access Modifier (cont'd)



In the same package



Child is outside the package

Inheritance: Access Modifier (cont'd)

■ Example)

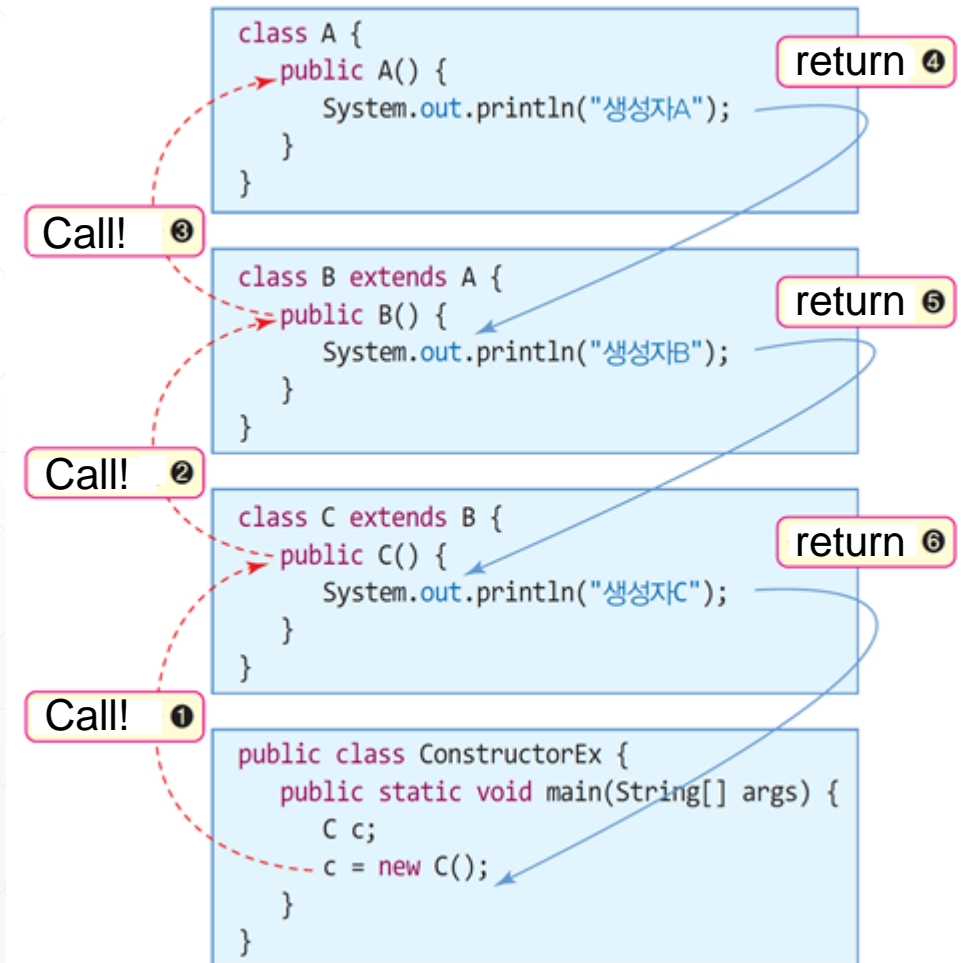
```
class Person {  
    private int weight;  
    int age;  
    protected int height;  
    public String name;  
  
    public void setWeight(int weight) {  
        this.weight = weight;  
    }  
    public int getWeight() {  
        return weight;  
    }  
}
```

```
class Student extends Person {  
    public void set() {  
        age = 30; // OK  
        name = "Jinwoo"; // OK  
        height = 175; // OK  
        // weight = 99; // Error. Private member of superclass  
        setWeight(99); // OK  
    }  
}
```

```
public class InheritanceEx {  
    public static void main(String[] args) {  
        Student s = new Student();  
        s.set();  
    }  
}
```


Inheritance: Constructor

- What happens when a new object of a subclass is instantiated?
 - Constructor() of Superclass is invoked first!
 - Constructor() of Subclass is then invoked
- What will be printed out from this example?



Inheritance: Constructor (cont'd)

- Which constructor of a superclass will be invoked?
 - Multiple constructors can be defined in both super/sub classes
- Constructor of a subclass **MUST** choose the super-constructor to invoke
 - Implicit invocation
 - Explicit invocation

Inheritance: Constructor (cont'd)

- Implicit invocation of a super-constructor
 - When a subclass constructor does not specify which one to invoke
 - Java compiler automatically inserts a call to the no-argument constructor (i.e., default constructor) of the superclass

```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        .....  
    }  
}
```

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
}
```

```
public class ConstructorEx2 {  
    public static void main(String[] args) {  
        B b;  
        b = new B();  
    }  
}
```

Inheritance: Constructor (cont'd)

■ Implicit invocation of a super-constructor

- Exception) What if no default constructor defined in the superclass?
 - If a superclass has constructors with parameters, then no default constructor provided by Compiler
- There will be a compile error!

“There is no default constructor available in [Superclass]”

- Default constructor must be defined!

```
class A {  
    public A(int x) {  
        System.out.println("생성자A");  
    }  
}
```

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
}
```

```
public class ConstructorEx2 {  
    public static void main(String[] args) {  
        B b;  
        b = new B();  
    }  
}
```



Inheritance: Constructor (cont'd)

■ Implicit invocation of a super-constructor

- Exception) What if a constructor with parameters of a subclass invoked?
 - Which super-constructor in this case will be selected?
- **Only default constructor** of a super-class is invoked!
 - This is also a case of “Implicit Invocation”

```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        System.out.println("매개변수생성자A");  
    }  
}
```

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
    public B(int x) {  
        System.out.println("매개변수생성자B");  
    }  
}
```

```
public class ConstructorEx3 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(5);  
    }  
}
```

Inheritance: Constructor (cont'd)

■ Explicit invocation of a super-constructor

- When a subclass constructor invokes a specific super-constructor using “**super()**” method call
- Any of superclass constructors can be selected
 - Super(): the default super constructor
 - Super(params): the super constructor with parameters
- Explicit super-constructor invocation **must be in the first line** of the sub-constructor
 - Same to that of this() invocation

Inheritance: Constructor (cont'd)

■ Explicit invocation of a super-constructor

- Any of superclass constructors can be selected
 - Super(): the default super constructor
 - Super(params): the super constructor with parameters
- Explicit super-constructor invocation must be in the first line of the sub-constructor
 - Same to that of this() invocation

```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        System.out.println("매개변수생성자A" + x);  
    }  
}
```

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
    public B(int x) {  
        super(x);  
        System.out.println("매개변수생성자B" + x);  
    }  
}
```

x=5

```
public class ConstructorEx4 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(5);  
    }  
}
```


Inheritance: Constructor (cont'd)

■ Explicit invocation of a super-constructor

```
class Point {  
    private int x, y;  
    public Point() {  
        this.x = this.y = 0;  
    }  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void showPoint() {  
        System.out.println("(" + x + "," + y + ")");  
    }  
}
```

```
class ColorPoint extends Point {  
    private String color;  
    public ColorPoint(int x, int y, String color) {  
        super(x, y);  
        this.color = color;  
    }  
    public void showColorPoint() {  
        System.out.print(color);  
        showPoint();  
    }  
}
```

```
public class SuperEx {  
    public static void main(String[] args) {  
        ColorPoint cp = new ColorPoint(5, 6, "blue");  
        cp.showColorPoint();  
    }  
}
```


Q&A

■ Next week

- Up/Downcasting
- Method Overriding