5.3 Extending And Implementing Interfaces

- Interface
  - declare features but provide no implementation
  - interface에서 declare한 모든 feature를 class는 implement해야함

- 1) Implementation relationship
  - among classes and interfaces
  - a class may implement zero or more interfaces
  - Class는 Interface로부터 어떤 형태의 implementation을 inherit받지 않음

- 2) Extension relationship among interfaces
  - the extended interface does not inherit any implementation from the base interface.
  - An interface can only extend other interface, not classes

5.3 Extending And Implementing Interfaces

- Multiple Inheritance In Java
  - <u>single inheritance</u> for class extension
  - <u>multiple inheritance</u> for interface extension and interface implementation
    - class는 object라는 single root class 존재
    - 그러나, interface는 single root 없음
    - class가 implement 할 때 (여러 multiple interface를...) 모든 abstract를 override해야 함
  - a class may <u>implement multiple interface</u>
  - an interface may <u>extend multiple interfaces</u>

5.3 Extending And Implementing Interfaces

- Implementation : by overriding abstract methods.

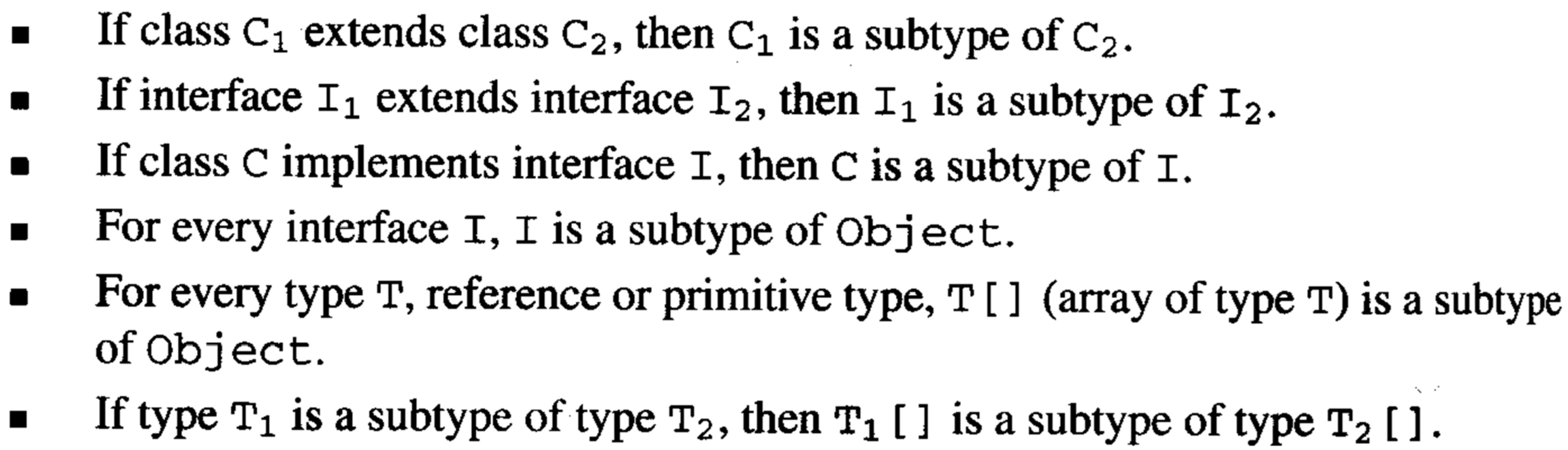
```
Public interface MyInterface {
          void aMethod(int i) ; //an abstract Method
}

Public class MyClass implements MyInterface {
          public void aMethod (int  i) {
                      //implementation
             }
             //...
}
```

5.3.1 Subtypes Revisited

- Interface & subtype relations
  - each interface also defines a type
  - the interface extension and implementation are also <u>subtype relations</u>.
  - <u>A reference type in Java</u> can be either a class type, an array type, or an interface type.

5.3.1 Subtypes Revisited

- Complete subtype relations in Java



- If class `C1` extends class `C2`, then `C1` is a subtype of `C2`.
- If interface `I1` extends interface `I2`, then `I1` is a subtype of `I2`.
- If class `C` implements interface `I`, then `C` is a subtype of `I`.
- For every interface `I`, `I` is a subtype of `Object`.
- For every type `T`, reference or primitive type, `T[]` (array of type `T`) is a subtype of `Object`.
- If type `T1` is a subtype of type `T2`, then `T1[]` is a subtype of type `T2[]`.
  - for every Class C that is not Object, C is a subtype of Object

- Ex) Implementing multiple interfaces…
  - Class가 <u>different context</u>에서, different role을 하도록 함

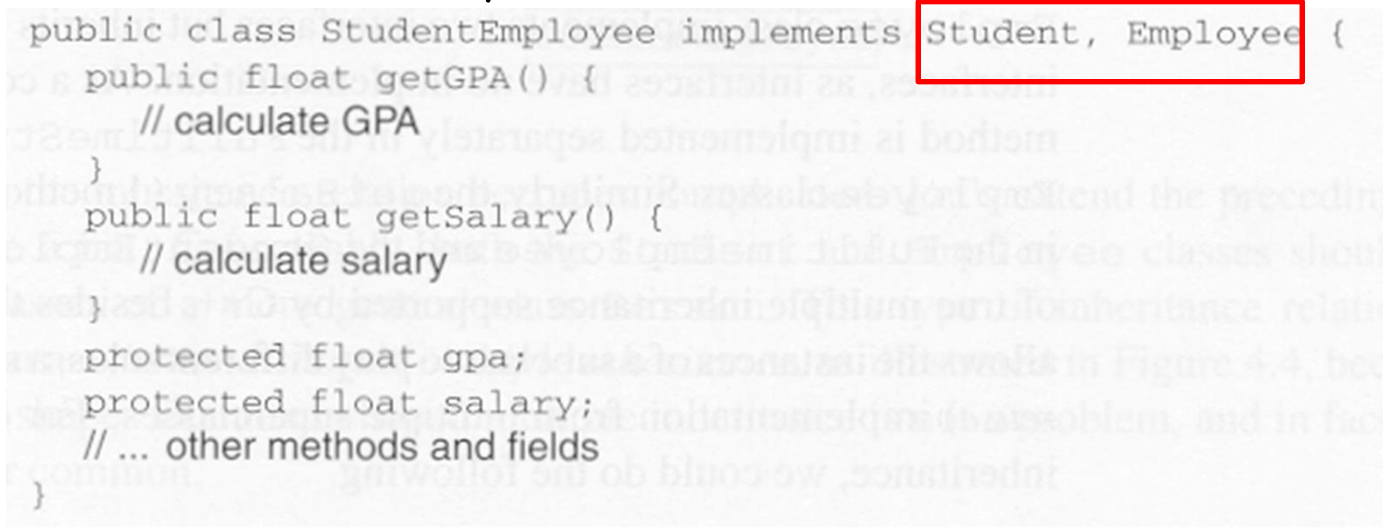
```
interface Student {
   float getGPA();
   // ... other methods
}

interface Employee {
   float getSalary();
   // ... other methods
}
```

- Implement each interface

```
public class FulltimeStudent implements Student {
   public float getGPA() {
      // calculate GPA
   }
   protected float gpa;
   // ... other methods and fields
}

public class FulltimeEmployee implements Employee {
   public float getSalary() {
      // calculate salary
   }
   protected float salary;
   // ... other methods and fields
}
```

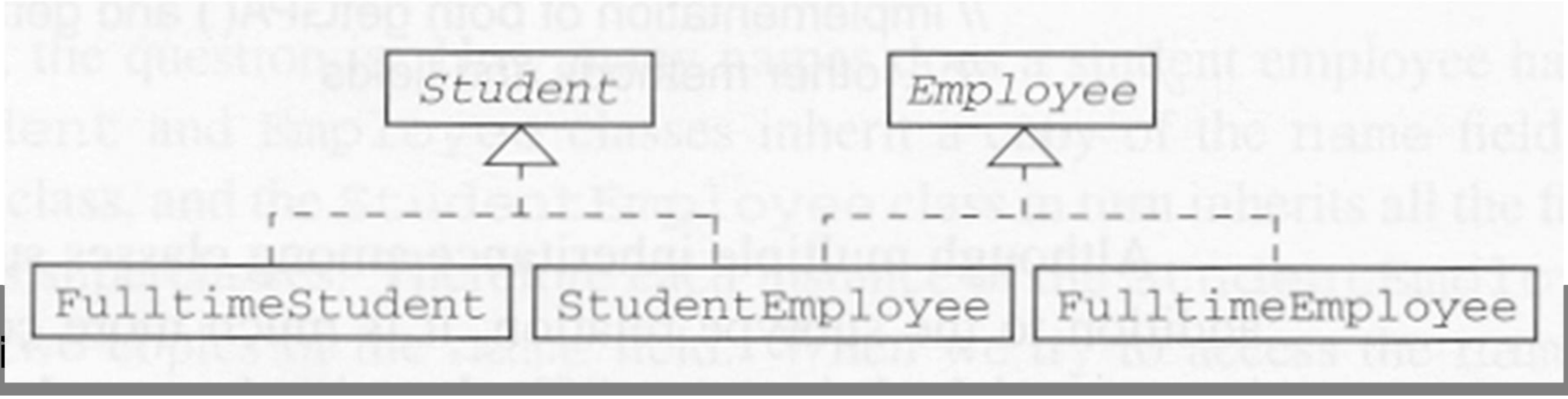
5.3.1 Subtypes Revisited

- A class can also implement both interfaces.



```
public class StudentEmployee implements Student, Employee {
   public float getGPA() {
      // calculate GPA
   }
   public float getSalary()  {
      // calculate salary
   }
   protected float gpa;
   protected float salary;
   // ... other methods and fields
}
```

- Figure 5.3 Implementation of interfaces



5.3.1 Subtypes Revisited

- A student employee can be viewed <u>as a student</u>.

```
Student[] students = new Student[...];
students[0] = new FulltimeStudent();
students[1] = new StudentEmployee(); // a student employee as a student
// ...
for (int i = 0; i < students.length; i++) {
  ... students[i].getGPA() ...
}
```

- A student employee can be viewed <u>as a employee</u>

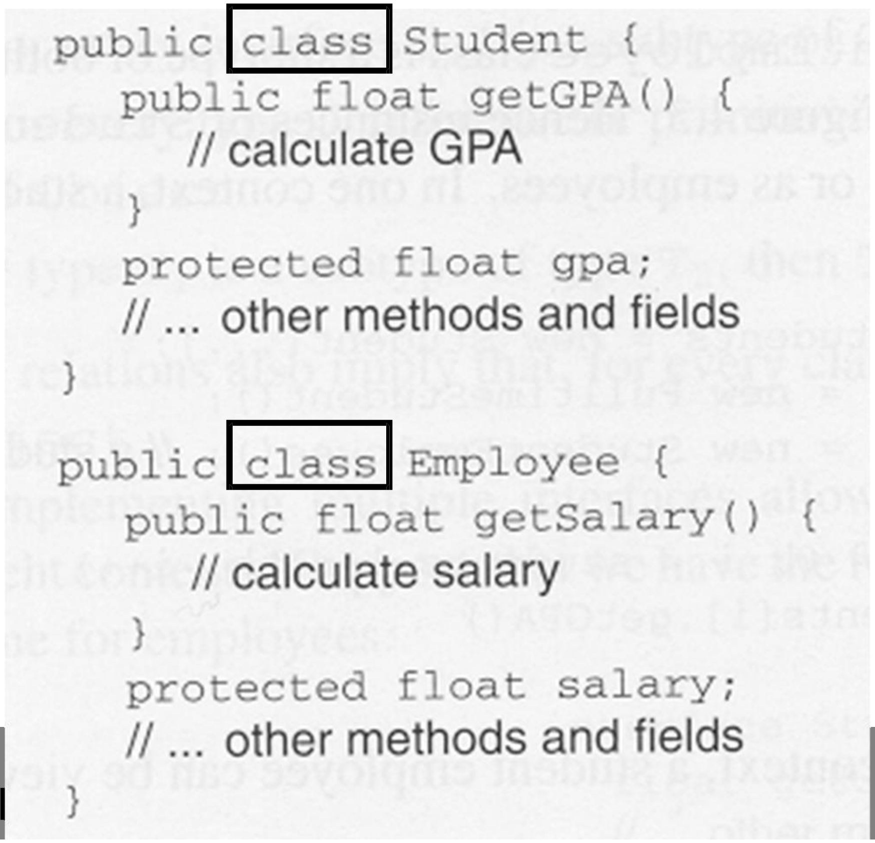
```
Employee[] employees = new Employee[...];
employees[0] = new FulltimeEmployee();
employees[1] = new StudentEmployee(); // a student employee as an employee
// ...
for (int i = 0; i < employees.length; i++) {
  ... employees[i].getSalary() ...
}
```

5.3.2 Single Versus Multiple Inheritance

- Single Inheritance // Multiple Inheritance
  - one of the most hotly debated issues
  - Java : support only limited multiple inheritance (interface extension and implementation)
    - ex) the StudentEmployee .. Inherits no implementation from the interface
    - the getGPA() method is implemented separately in the FullTimeStudent, and the StudentEmployee Classes.
  - C++ : support true multiple inheritance
    - subclass can also inherit (I.e.) reuse implementation from multiple superclasses.

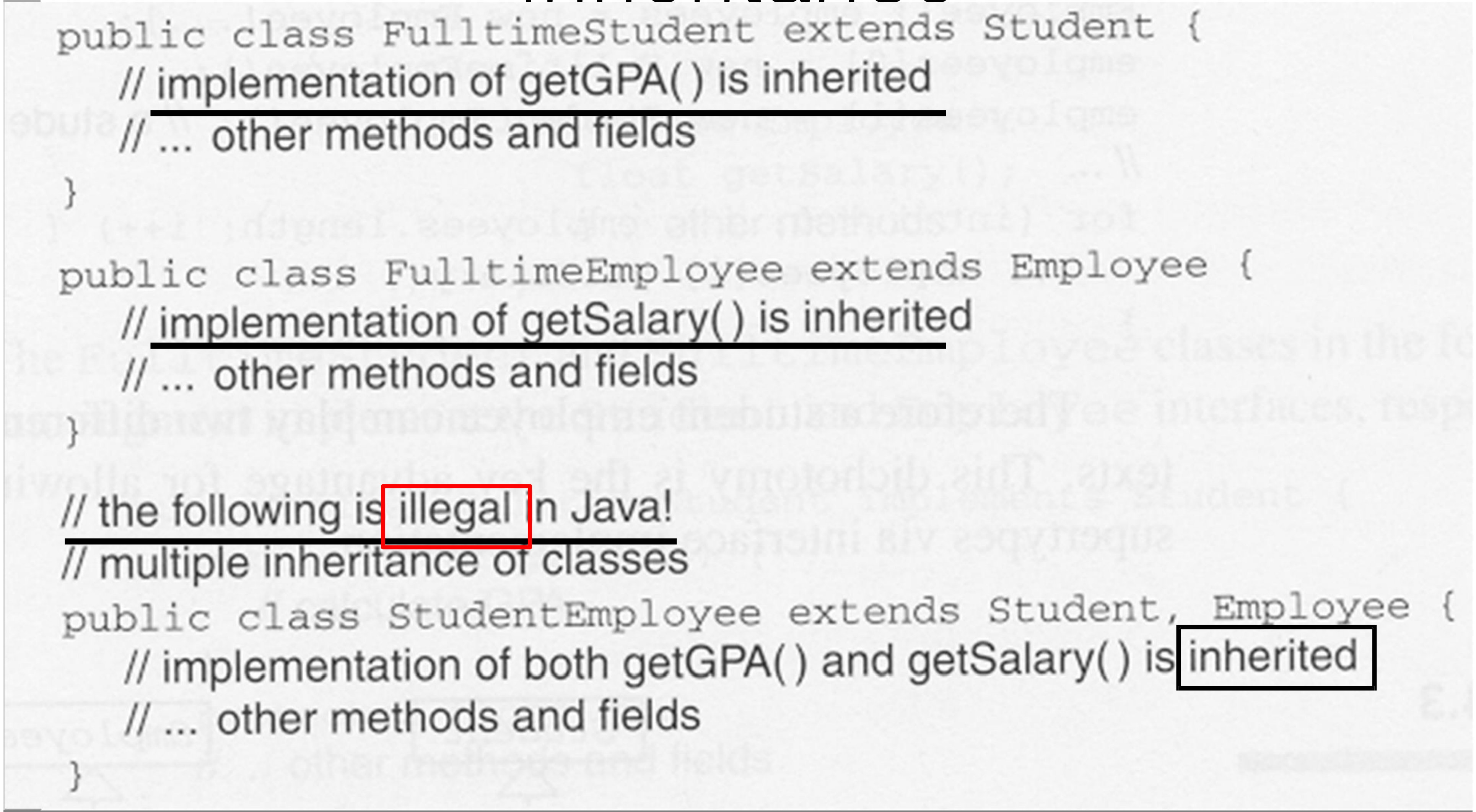
5.3.2 Single Versus Multiple Inheritance

- GetGPA() is implemented separately.
- With true multiple inheritance....



```
public class Student {
    public float getGPA() {
        // calculate GPA
    }
    protected float gpa;
    // ... other methods and fields
}

public class Employee {
    public float getSalary() {
        // calculate salary
    }
    protected float salary;
    // ... other methods and fields
}
```

5.3.2 Single Versus Multiple Inheritance



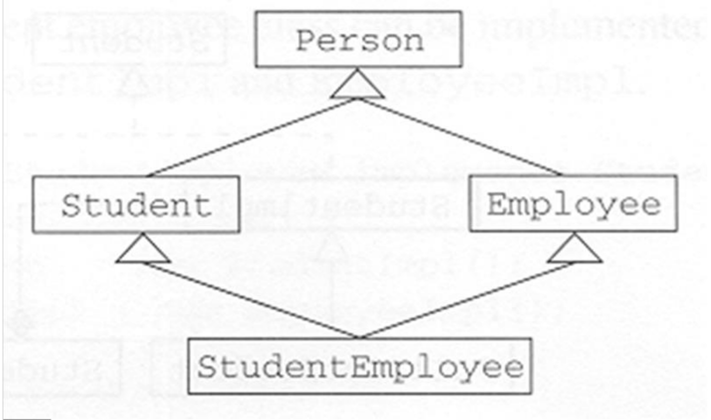
```
public class FulltimeStudent extends Student {
    // implementation of getGPA( ) is inherited
    // ... other methods and fields
}

public class FulltimeEmployee extends Employee {
    // implementation of getSalary( ) is inherited
    // ... other methods and fields
}

// the following is illegal in Java!
// multiple inheritance of classes
public class StudentEmployee extends Student, Employee {
    // implementation of both getGPA( ) and getSalary( ) is inherited
    // ... other methods and fields
}
```

5.3.2 Single Versus Multiple Inheritance

- Multiple inheritance
  - advantages : support implementation reuse
  - disadvantages
    - more complicated
    - more difficult to implement
    - less efficient
    - difficult to use
- Diamond-shape multiple inheritance
  - More general Class Person.

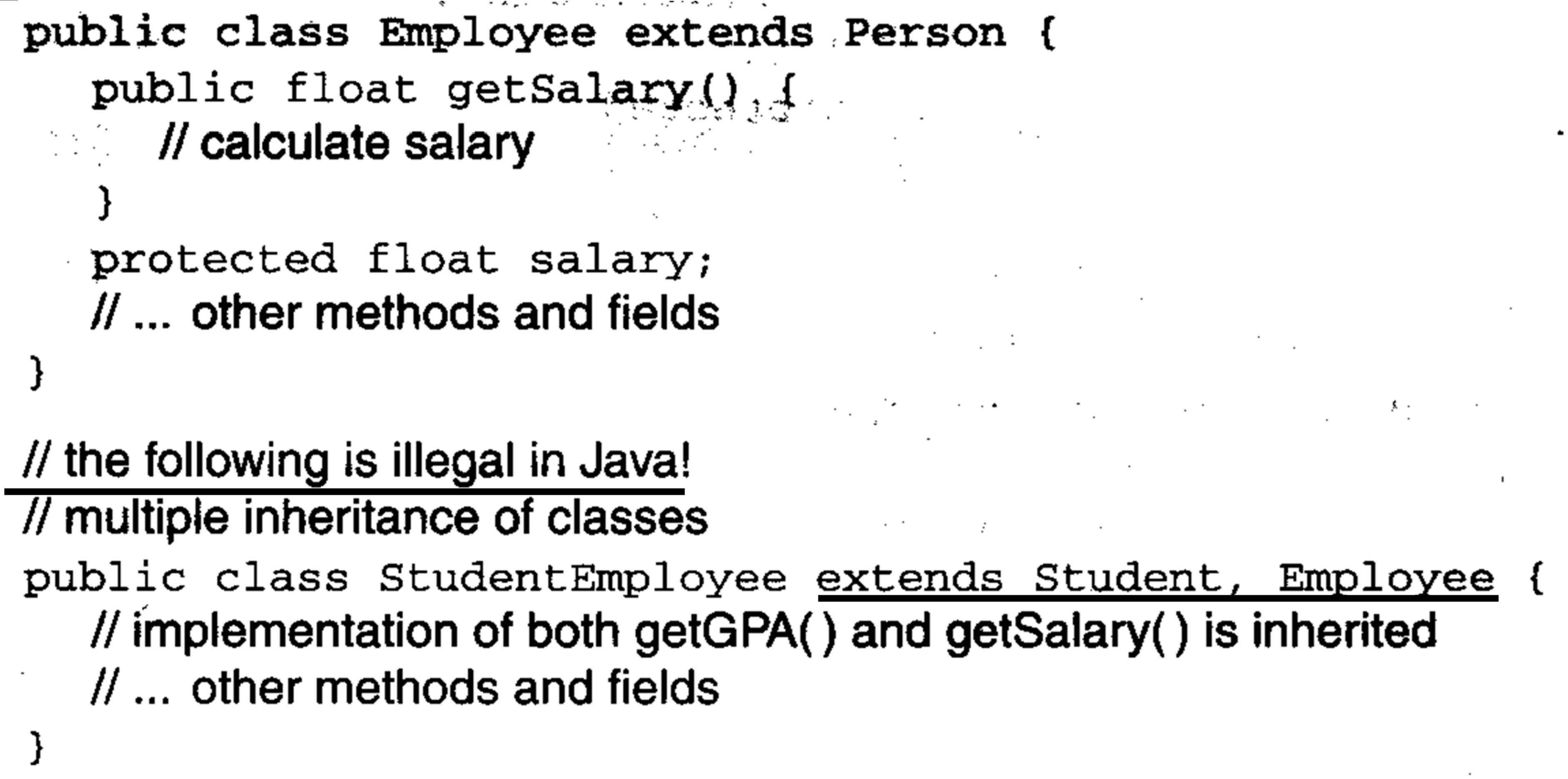
5.3.2 Single Versus Multiple Inheritance

- Diamond-shaped multiple inheritance



```
public class Person {
   public String getName() {
      // ...
   }
   protected String name;
}

public class Student extends Person {
   public float getGPA() {
      // calculate GPA
   }
   protected float gpa;
   // ... other methods and fields
}
```

5.3.2 Single Versus Multiple Inheritance



```
public class Employee extends Person {
   public float getSalary() {
      // calculate salary
   }
   protected float salary;
   // ... other methods and fields
}

// the following is illegal in Java!
// multiple inheritance of classes
public class StudentEmployee extends Student, Employee {
   // implementation of both getGPA() and getSalary() is inherited
   // ... other methods and fields
}
```

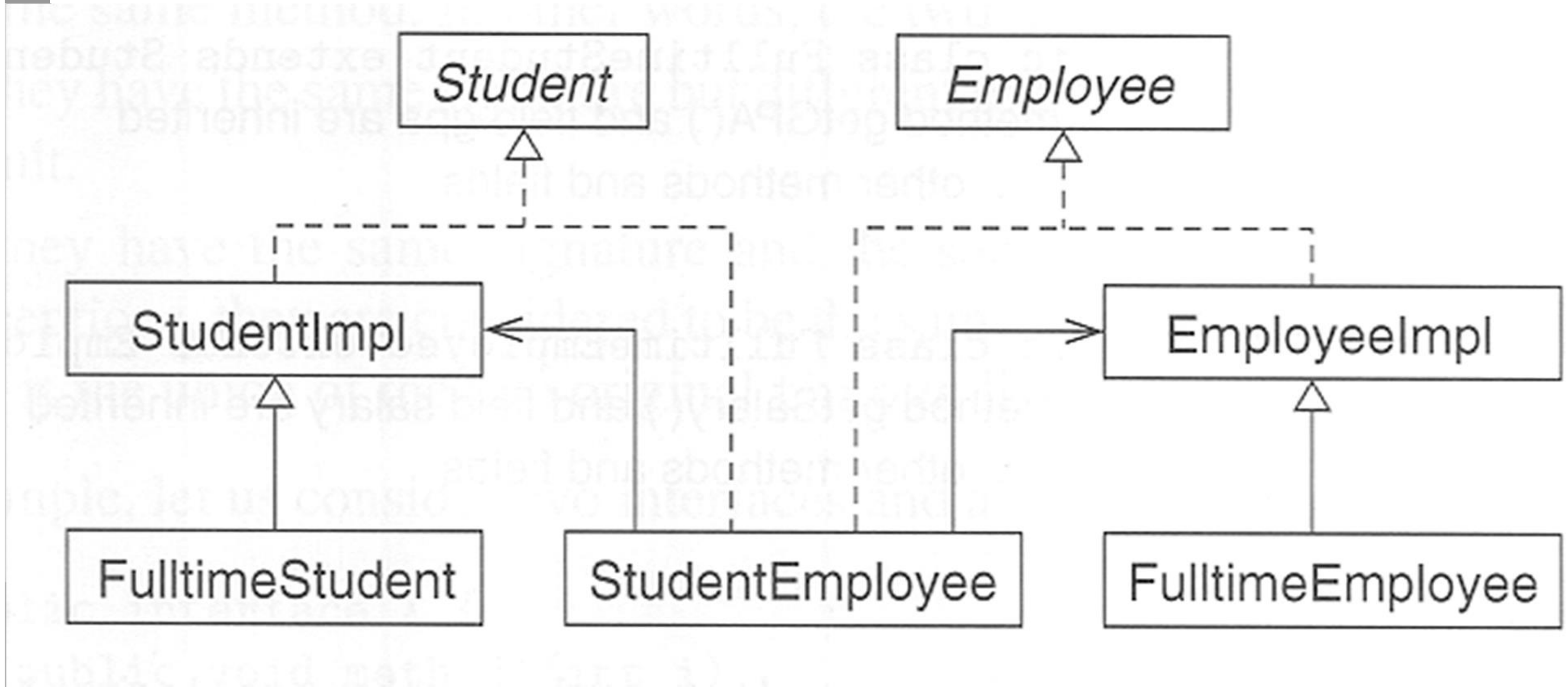
5.3.2 Single Versus Multiple Inheritance

- Questions
  - How many names does a student employee have ?
    - two copies of the name field? When we try to access the name field of a student employee, which copy is access ed ?
    - is it possible to access both copies?
    - ⓪ Common sense, only one copy
  - ⓪ ⓪ C++(multiple inheritance)
    - Name resolution features
    - Virtual base classes

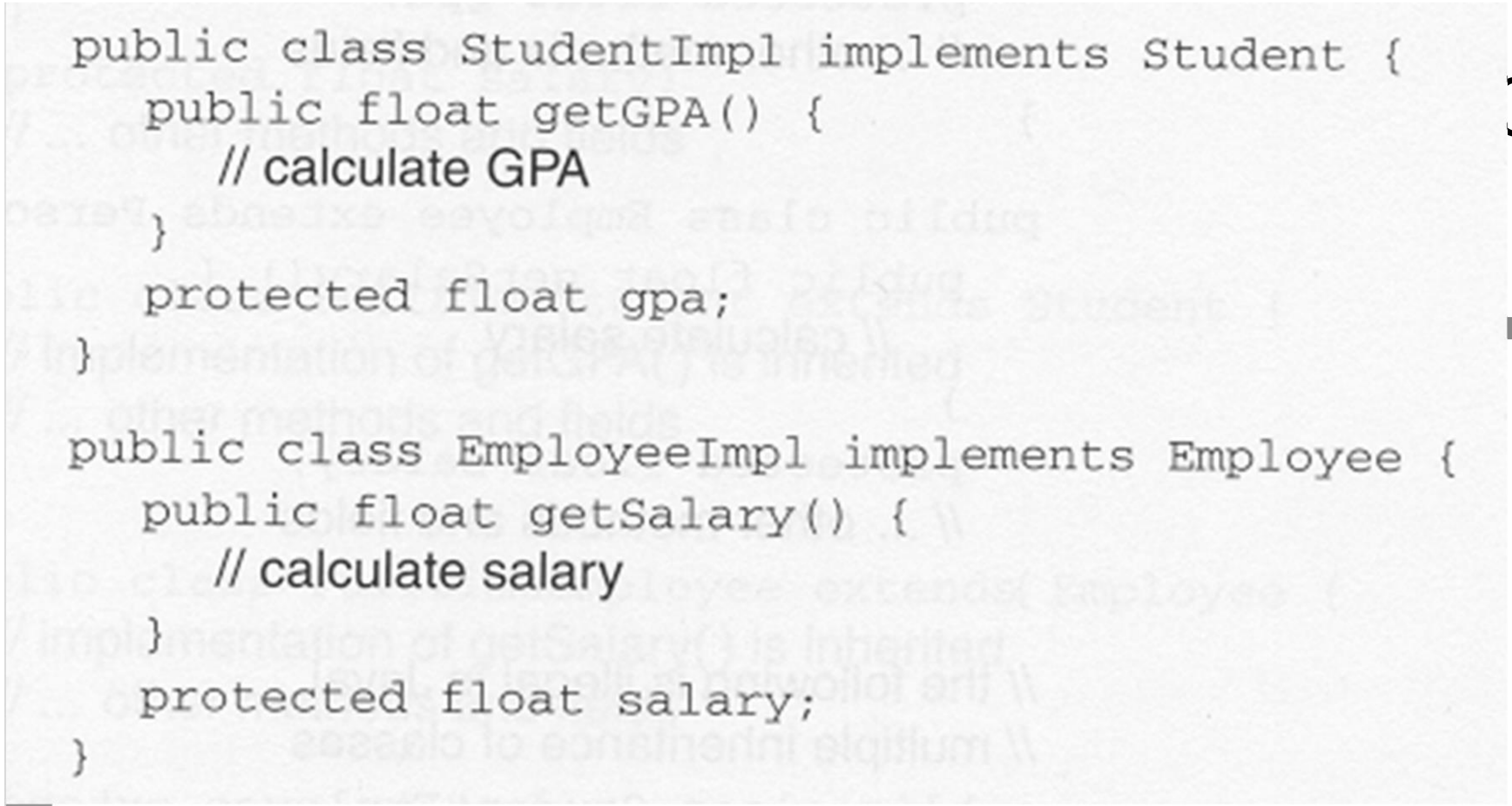
5.3.2 Single Versus Multiple Inheritance

```
interface Student {
  public float getGPA();
}

interface Employee {
  public float getSalary();
}
```

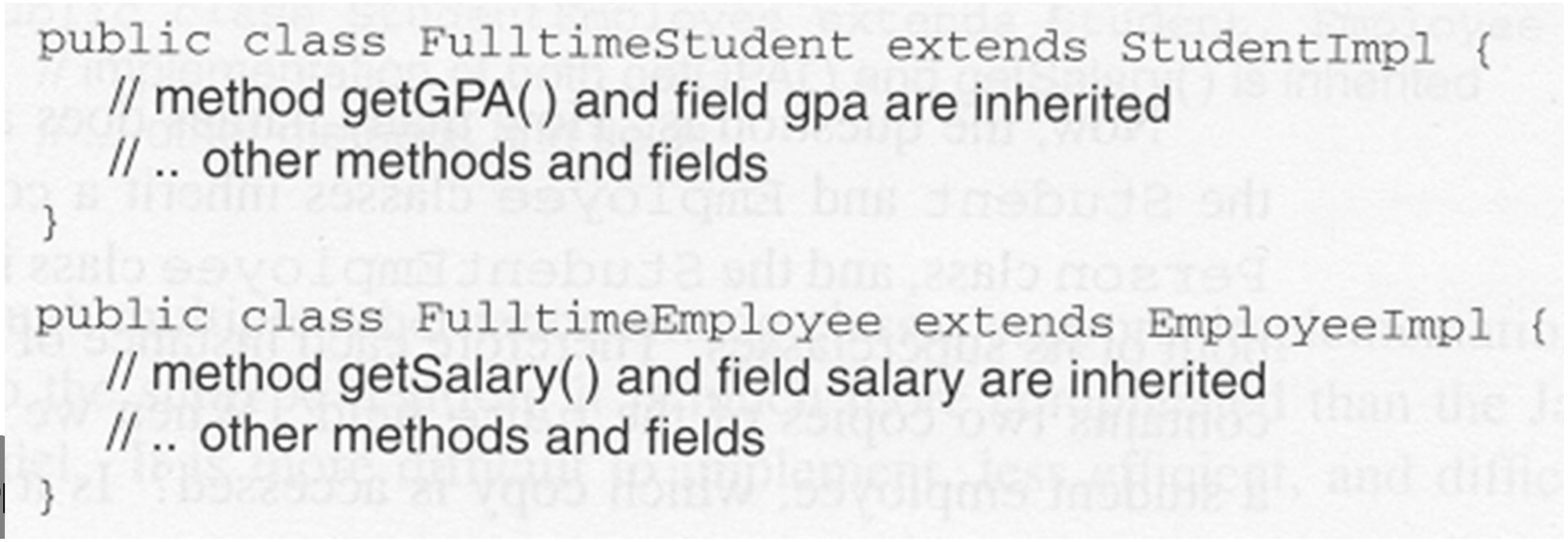
- The reuse of implementation (in Java)
  - Figure 5.5 Implementation reuse through delegation



ple

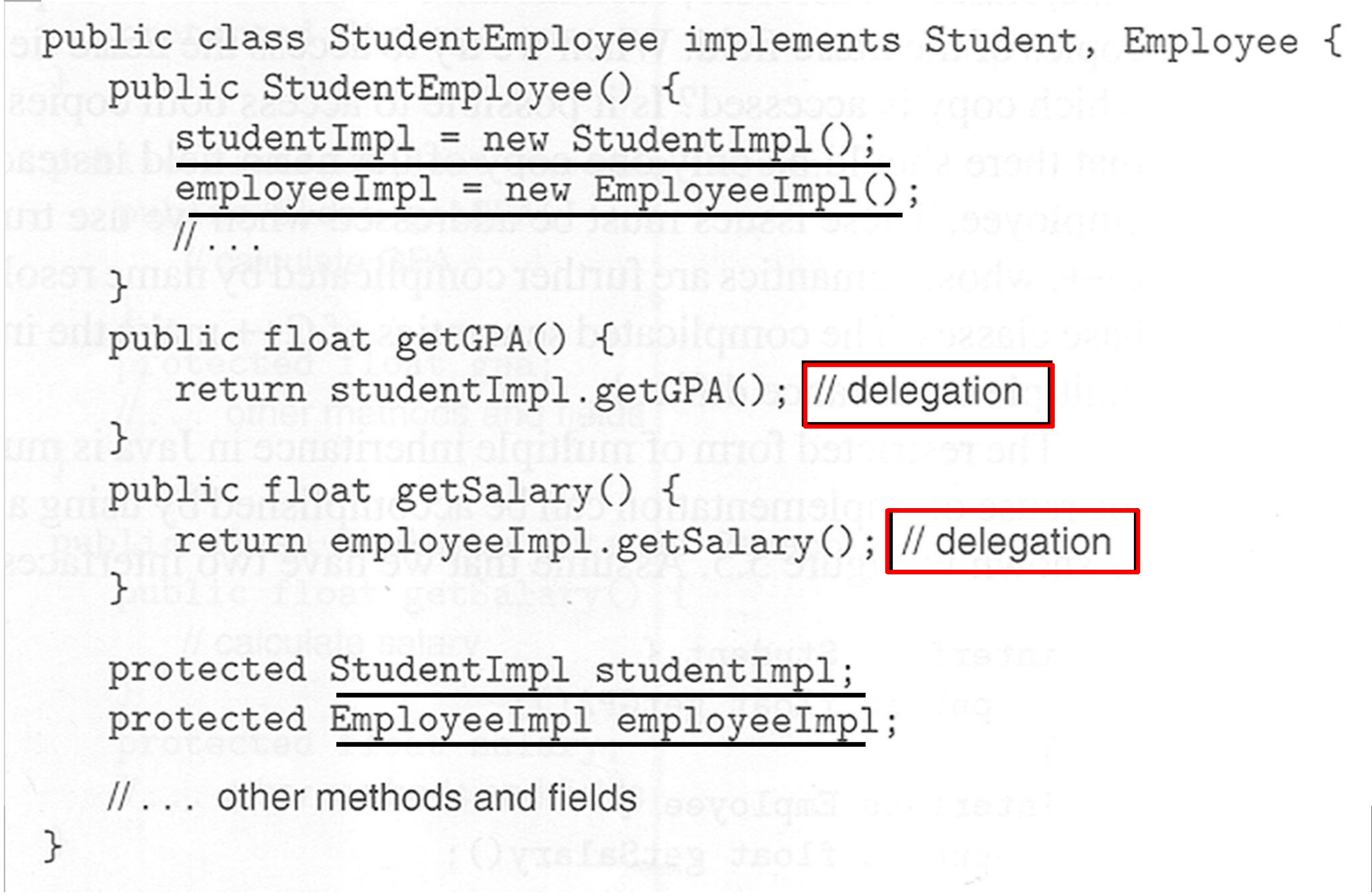


```
public class StudentImpl implements Student {
    public float getGPA() {
        // calculate GPA
    }
    protected float gpa;
}

public class EmployeeImpl implements Employee {
    public float getSalary() {
        // calculate salary
    }
    protected float salary;
}
```



```
public class FulltimeStudent extends StudentImpl {
    // method getGPA() and field gpa are inherited
    // .. other methods and fields
}

public class FulltimeEmployee extends EmployeeImpl {
    // method getSalary() and field salary are inherited
    // .. other methods and fields
}
```

5.3.2 Single Versus Multiple Inheritance



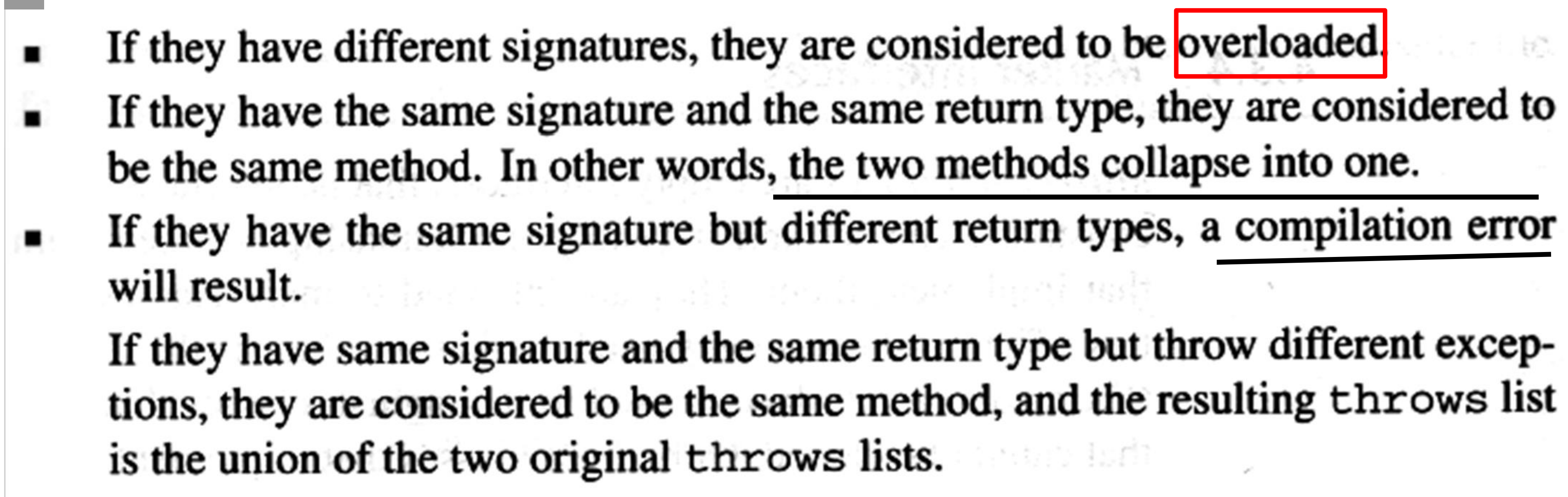
```
public class StudentEmployee implements Student, Employee {
   public StudentEmployee() {
      studentImpl = new StudentImpl();
      employeeImpl = new EmployeeImpl();
      // . . .
   }
   public float getGPA() {
      return studentImpl.getGPA(); // delegation
   }
   public float getSalary() {
      return employeeImpl.getSalary(); // delegation
   }

   protected StudentImpl studentImpl;
   protected EmployeeImpl employeeImpl;

   // . . . other methods and fields
}
```

5.3.2 Single Versus Multiple Inheritance

- Delegation (위임)
  - each method simply delegates the task to <u>another object</u>, studentImpl and Employe eImpl
  - the implementation in the StudentImpl and EmployeeImpl classes is reused through d elegation.

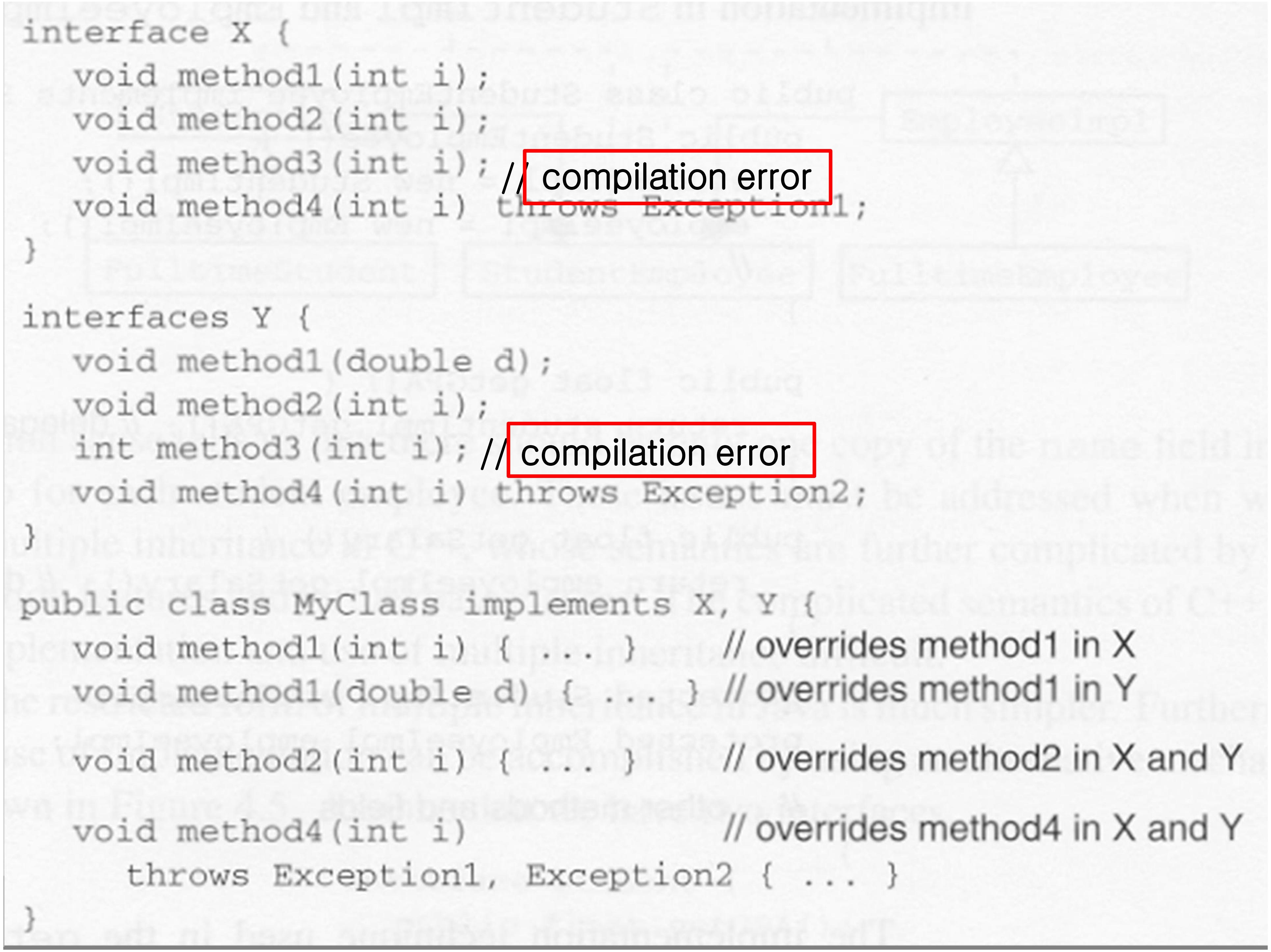
5.3.3 Name Collisions among Interface

- Name Collisions
  - Names inherited from one interface <u>may collide with</u> names inherited from another interface or classes
- Two methods with the same name offers the following possibilities.

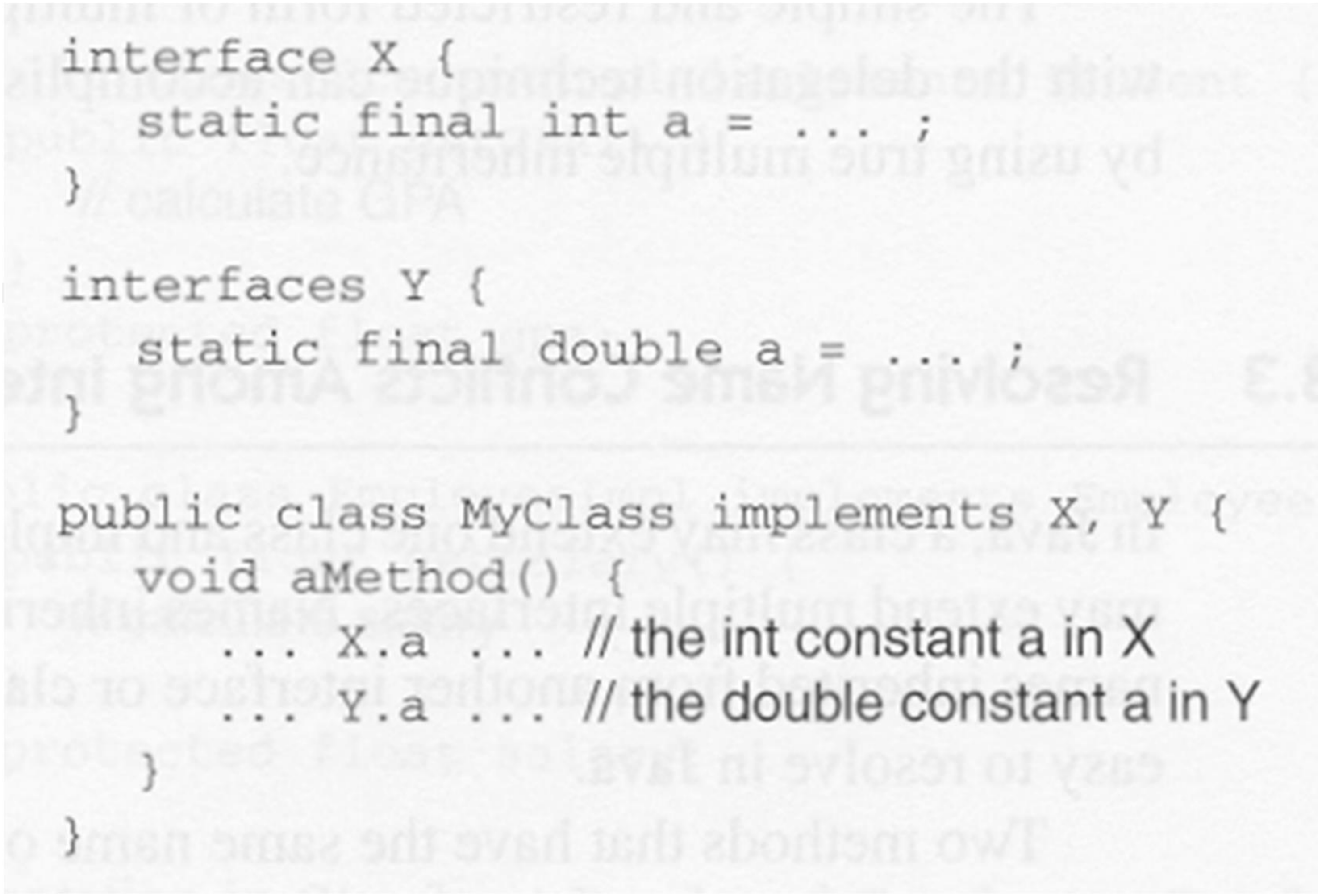


- If they have different signatures, they are considered to be overloaded.
- If they have the same signature and the same return type, they are considered to be the same method. In other words, the two methods collapse into one.
- If they have the same signature but different return types, a compilation error will result.

If they have same signature and the same return type but throw different exceptions, they are considered to be the same method, and the resulting `throws` list is the union of the two original `throws` lists.



```
interface X {
   void method1(int i);
   void method2(int i);
   void method3(int i); // compilation error
   void method4(int i) throws Exception1;
}

interfaces Y {
   void method1(double d);
   void method2(int i);
   int method3(int i); // compilation error
   void method4(int i) throws Exception2;
}

public class MyClass implements X, Y {
   void method1(int i) { ... }        // overrides method1 in X
   void method1(double d) { ... }     // overrides method1 in Y

   void method2(int i) { ... }        // overrides method2 in X and Y

   void method4(int i)                // overrides method4 in X and Y
      throws Exception1, Exception2 { ... }
}
```



```
interface X {
   static final int a = ... ;
}

interfaces Y {
   static final double a = ... ;
}

public class MyClass implements X, Y {
   void aMethod() {
      ... X.a ... // the int constant a in X
      ... Y.a ... // the double constant a in Y
   }
}
```

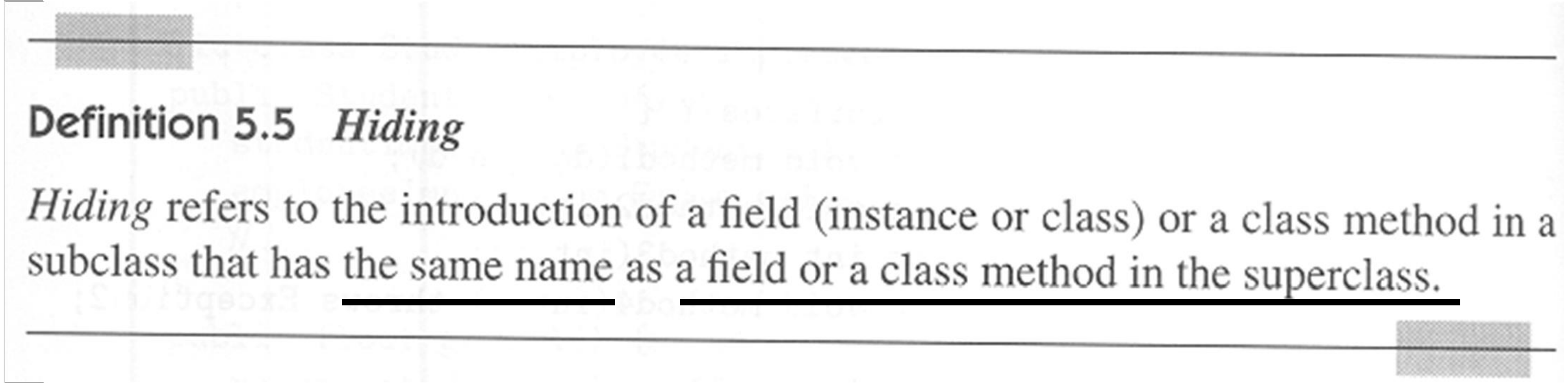
- Two constants having the same name is always allowed.

5.3.4 Marker Interfaces

- Marker Interfaces
  - Empty interfaces
  - interfaces that declare no methods or constants
  - 1) establish a subtype relationship between themselves and the classes that implement them.
  - 2) to mark classes as having certain properties.
  - Ex) Cloneable interface
    - used to distinguish classes that can be cloned from those that cannot be cloned.
    - only those classes that implement that the Cloneable interface can be cloned.

5.4 Hiding Fields And Class(Static) Methods

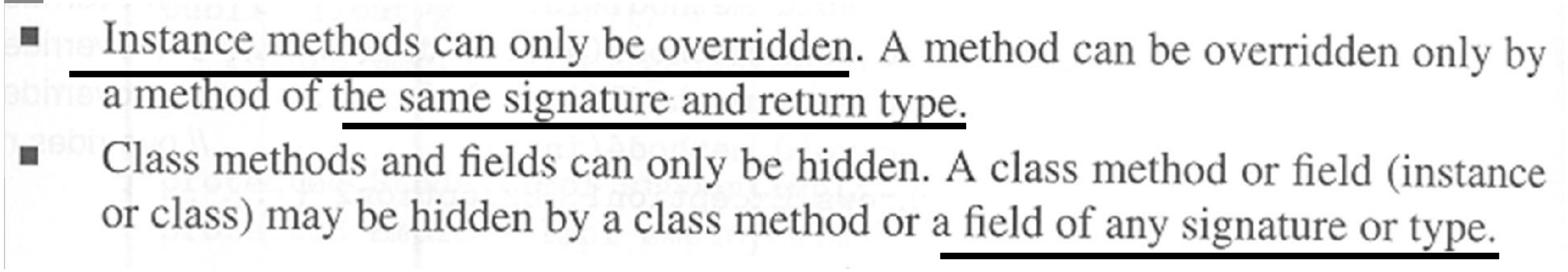
- Definition 4.5 Hiding



Definition 5.5 ***Hiding***

Hiding refers to the introduction of a field (instance or class) or a class method in a subclass that has the same name as a field or a class method in the superclass.

- Overriding & Hiding

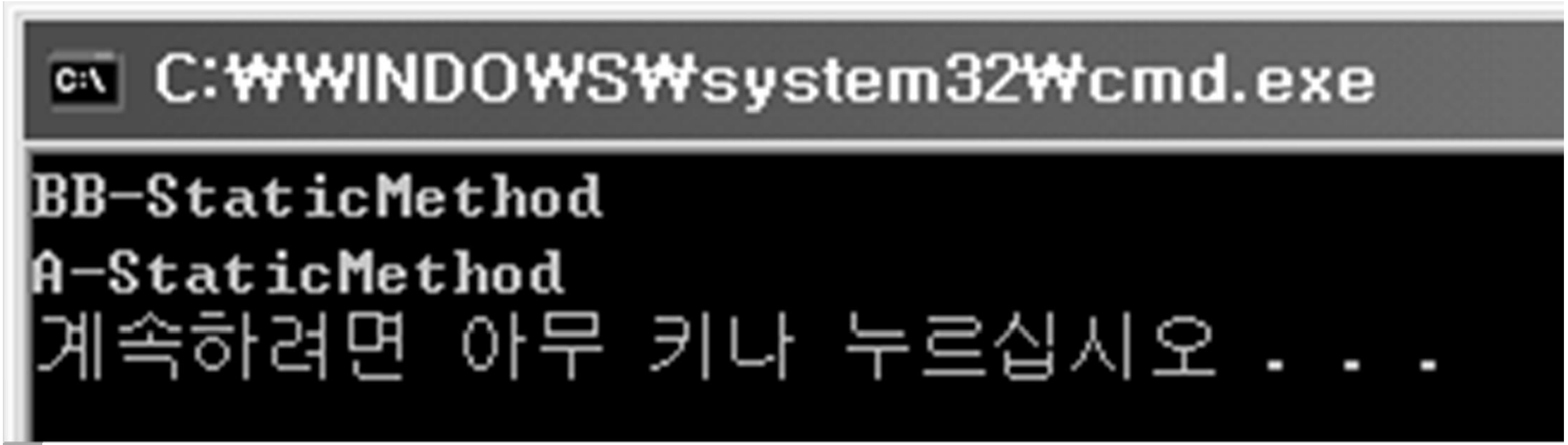


- Instance methods can only be overridden. A method can be overridden only by a method of the same signature and return type.
- Class methods and fields can only be hidden. A class method or field (instance or class) may be hidden by a class method or a field of any signature or type.

```
// p 184.
public class A{
   int x=1;
   void y() { System.out.println("AAAAAA");};
   static void z() {System.out.println("A-StaticMethod");};
}
====
// p. 184
public class B extends A{
   double x=2.2;//??
   void y() {System.out.println("BBBBBBB");};//??
   static int z(){System.out.println("A-StaticMethod");//??
   return 3;}
   public static void main(String[] args)
     {
          System.out.println("RETURN Value " + z() );
    }
}    //????
```

```
D:\My Documents\@@@@강의-객체지향
\@@@@@@oo\Chapter05\p184hidingtest\B.java:5: z() in B
cannot override z() in A; attempting to use incompatible  re
turn type
found   : int  r
equired: void
   static int z(){System.out.println("A-StaticMethod");//??
              ^
1 error

Tool completed with exit code 1
```

```
// p 184.
public class A{
   int x=1;
   void y() { System.out.println("AAAAAA");};
   static void z() {System.out.println("A-StaticMethod");};
}
====
// p. 184
public class BB extends A{
   double x=2.2;   //???
   void y() {System.out.println("BBBBBBB");};  //???
   static void z(){System.out.println("BB-StaticMethod");}  //???
   public static void main(String[] args)
  {
                  BB b1 = new BB();
                  A a1 = b1;
                  b1.z();
                  a1.z();
  }
}    //????
```



```
BB-StaticMethod
A-StaticMethod
계속하려면 아무 키나 누르십시오 . . .
```

- Run time – an overridden method is invoked …. The i mplementation that will be executed is chosen at run time
- Compile Time… when a hidden method or field is i nvoked or accessed… the copy that will be used is determined at compile time.
  - Static(class) methods and fields .. Are statically bound.

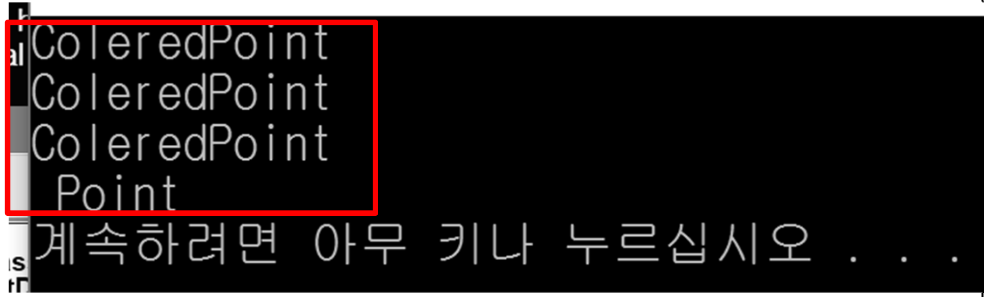
```
public class Point{
        public String className="Point";  st
        atic public String getDescription() {
                return "Point";
        }
        // other declaratioins
}
====
public class ColoredPoint extends Point{
        public String className="ColeredPoint";
        static public String getDescription() {
                return "ColeredPoint";
        }
        // other declaratioins
}
====

ColoredPoint p1= new ColoredPoint(10.0, 10.0, Color.blue);
Point p2 = p1;
System.out.println(p1.getDescription());  Syste
m.out.println(p2.getDescription());  System.out.
println(p1. className);  System.out.println(p2.
className);                                   //????
```

Output is

```
ColeredPoint
Point  Colere
dPoint  Point
```

- both p1 and p2 refer to the same object
  - But, the binding of the static methods and fields is based on the declared types of the variables at compile time
- Hiding
  - confusing in readability of program
  - the rule of hiding .... To resolve coincidental name collisions...

```
public class Point{
          public String className="Point";
          public String getDescription() {
                    return "Point";
          }
          // other declaratioins
}
====
public class ColoredPoint extends Point{
          public String className="ColeredPoint";
          public String getDescription() {
                    return "ColeredPoint";
          }
          // other declaratioins
}
====
```

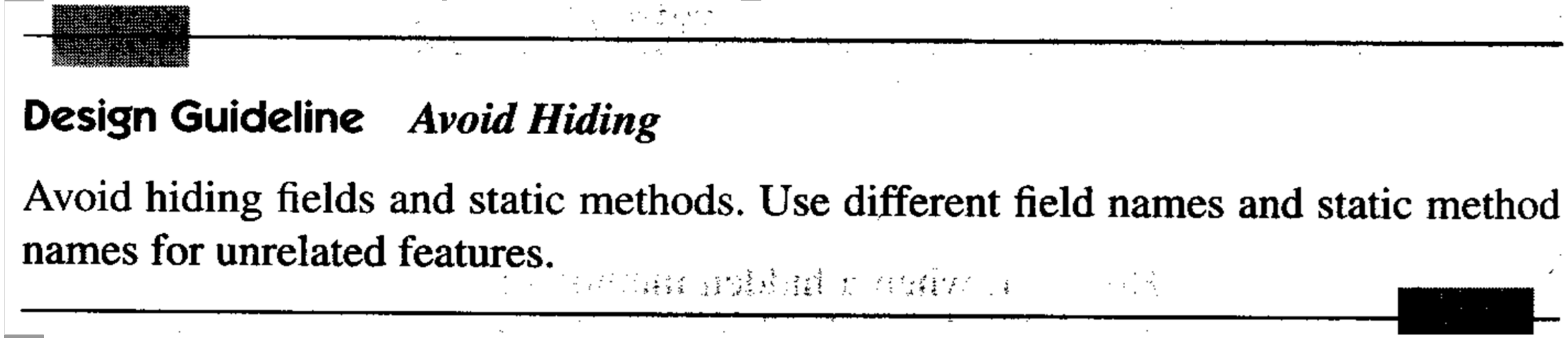


```
ColeredPoint
ColeredPoint
ColeredPoint
 Point
계속하려면 아무 키나 누르십시오 . . .
```

```
ColoredPoint p1= new ColoredPoint(10.0, 10.0, Color.blue);
Point p2 = p1;
System.out.println(p1.getDescription());  Syste
m.out.println(p2.getDescription());  System.out.
println(p1. className);  System.out.println(p2.
className);                                    //????
```

5.4 Hiding Fields And Static Methods

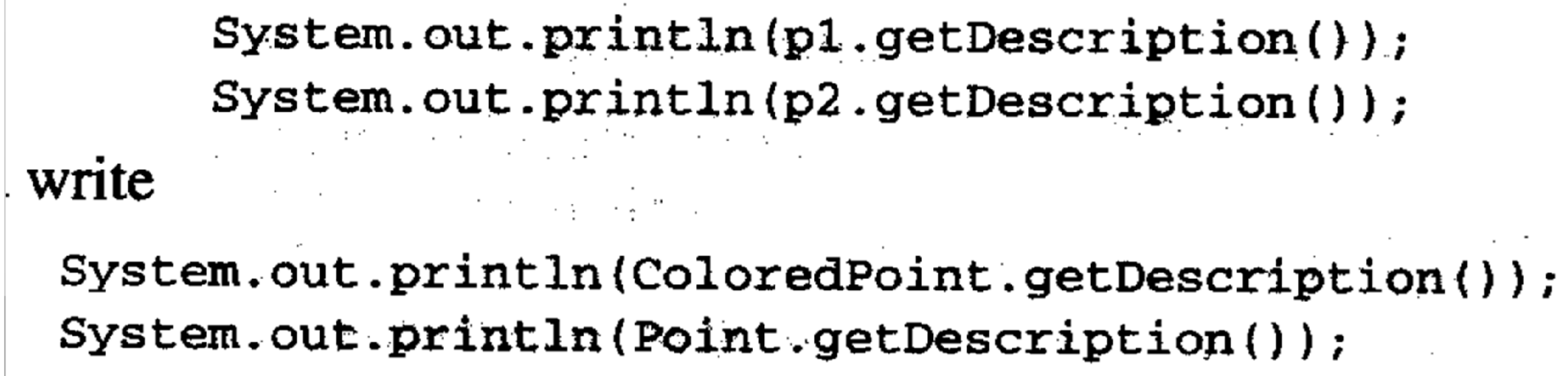
- Solution 1) avoid hiding



Design Guideline ***Avoid Hiding***

Avoid hiding fields and static methods. Use different field names and static method names for unrelated features.

- Solution 2) Static method의 Call에는 Class Name을 씀



```
System.out.println(p1.getDescription());
System.out.println(p2.getDescription());
```

write

```
System.out.println(ColoredPoint.getDescription());
System.out.println(Point.getDescription());
```