

Rules of Method Overloading and Overriding in Java? Examples

Since you can either overload or override methods in Java, it's important to know what are the *rules of overloading and overriding in Java*. any **overloaded method** or **overridden method** must follow rules of method overloading and method overriding to avoid compile-time error and logical runtime errors; where you intend to override a method but the method gets overloaded. That is not uncommon and happens several times when a Java programmer tries to **override equals in Java** or **overriding the compareTo method** in Java while implementing a Comparable interface, which accepts the Object type of argument.

From Java 5 onwards which introduces *@Override annotation* along with Enum, Generics, and varargs method you can completely avoid that problem.

Anyway, let's see the rules of method overloading and the rule of method overriding in Java. By the way, for those who are completely new in programming, both overloading and overriding means creating a method of the same name in either the same class or child class.

In the case of overriding, the original method in the parent class is known as the overridden method, while the new method in the child class is called the overriding method.

1. Rules of overloading a method in Java

Here is the list of the rule which needs to be followed to overload a method in Java :

1. Method Signature

The first and foremost rule to overload a method in Java is to change the **method signature**. the method signature is made of a number of arguments, types of arguments, and order of arguments if they are of different types. You can change any of these or combinations of them to overload a method in Java. For example, in the following code, we have changed a number of arguments to overload `whoAmI()` method in Java :

```
class UNIX {  
  
    protected final void whoAmI() {  
        System.out.println("I am UNIX");  
    }  
  
    protected final void whoAmI(String name){  
        System.out.println("I am " + name);  
    }  
  
}
```

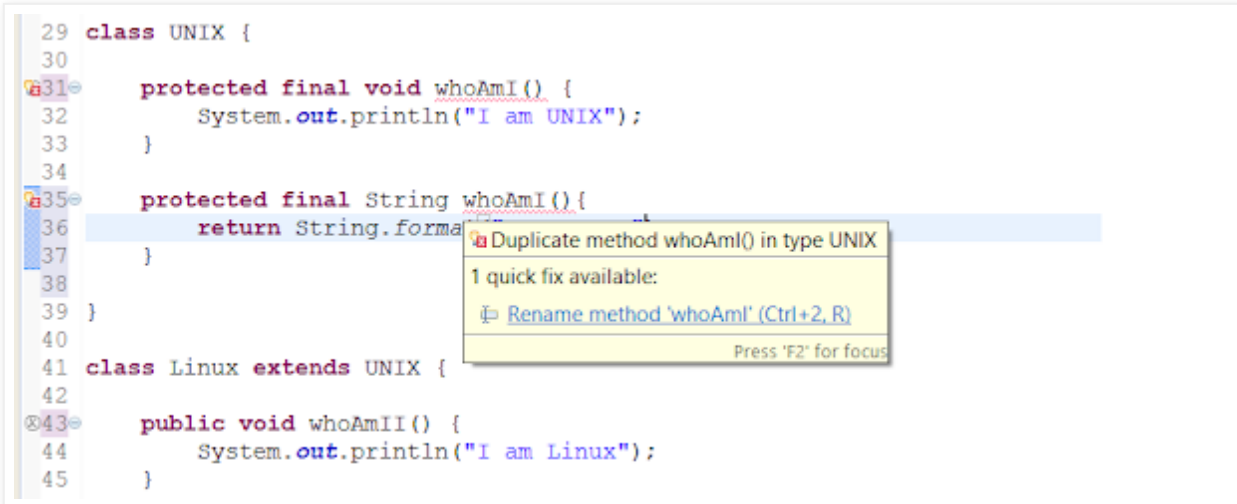
By the way, the right way to overload a method is to change the number of arguments or types of arguments because it denotes that it offers the same functionality but the input is different. For example, the `println()` method of `PrintStream` class has several overloaded versions to accept different data types like `String`, `int`, `long`, `float`, `double`, `boolean`, etc.

Similarly, the `EnumSet.of()` method is overloaded to accept one, two, or more values. The third way to overload method by changing the order of argument or type of argument creates a lot of confusion and best to be avoided, as discussed in my post about **overloading best practices in Java**.

2. Method Return Type

The return type of method is not part of the method signature, so just changing the return type will not overload a method in Java. In fact, just changing the return type will result in a

compile-time error as "duplicate method X in type Y. Here is a screenshot of that error in Eclipse :



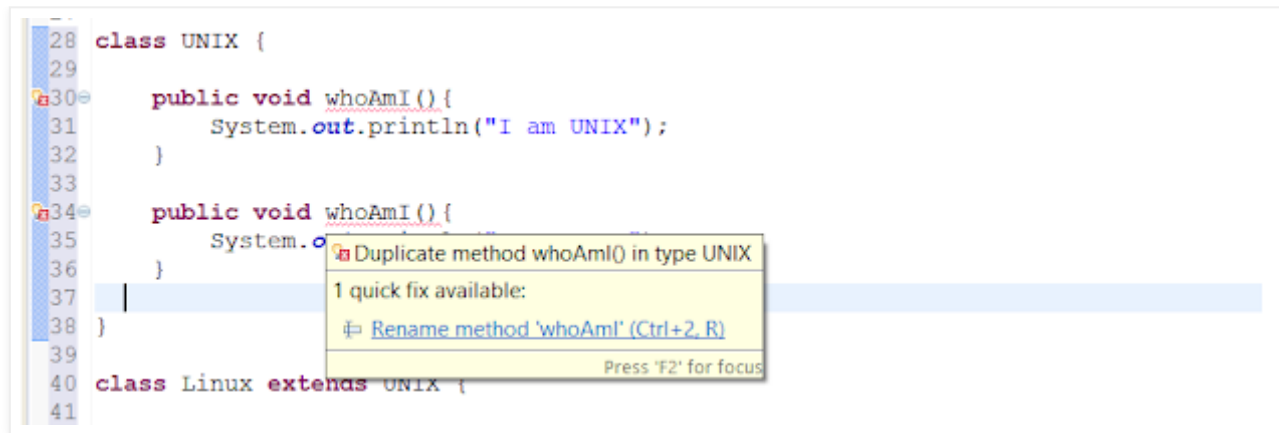
See [what is method overloading in Java](#) for code examples of these rules,

2. Method Overriding Rules in Java

Overriding is completely different than overloading and so its rules are also different. For terminology, the original method is known as overridden method and the new method is known as the overriding method. Following rules must be followed to correctly override a method in Java :

1. Location

A method can only be overridden in sub-class, not in the same class. If you try to create two methods with the same signature in one class compiler will complain about it saying *"duplicate method in type Class"*, as shown in the following screenshot :



2. Exception

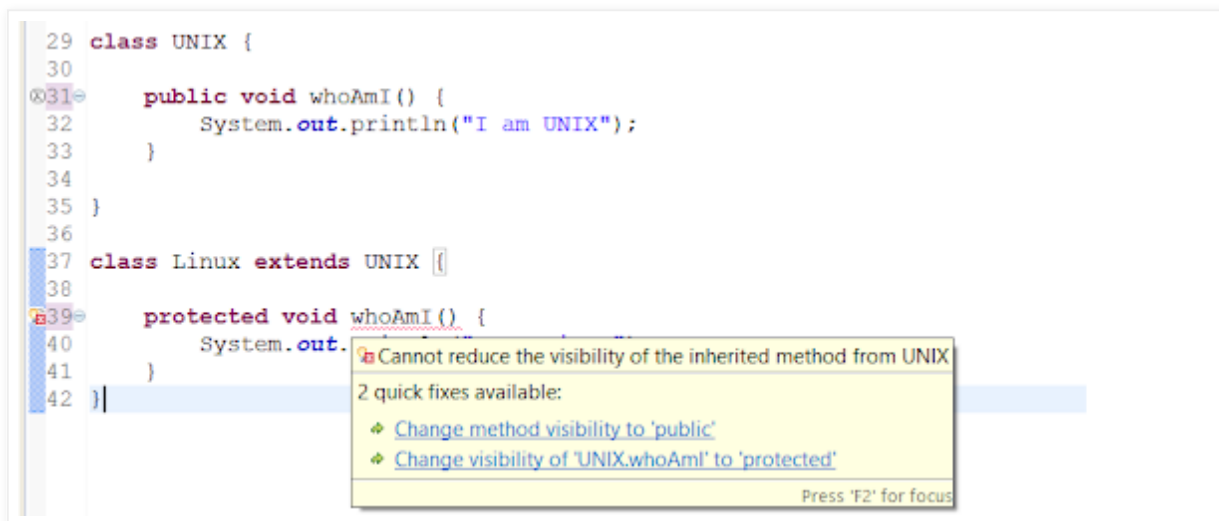
Overriding method cannot throw **checked Exception** which is higher in the hierarchy, than checked Exception thrown by the overridden method. For example, if an overridden method throws `IOException` or **`ClassNotFoundException`**, which are checked Exception then the overriding method can not throw `java.lang.Exception` because it comes higher in type hierarchy (it's the superclass of `IOException` and `ClassNotFoundException`). If you do so, the compiler will catch you as seen in the following image :



3. Visibility

The overriding method can not reduce access of overridden method. It means if the overridden method is defined as public then the overriding method can not be protected or package-private. Similarly, if the original method is protected then the overriding method cannot be package-private.

You can see what happens if you violate this rule in Java, as seen in this screenshot it will throw compile time error saying "You cannot reduce the visibility of inherited method of a class".



```
29 class UNIX {
30
31     public void whoAmI() {
32         System.out.println("I am UNIX");
33     }
34
35 }
36
37 class Linux extends UNIX {
38
39     protected void whoAmI() {
40         System.out.
41     }
42 }
```

Cannot reduce the visibility of the inherited method from UNIX

2 quick fixes available:

- Change method visibility to 'public'
- Change visibility of 'UNIX.whoAmI' to 'protected'

Press 'F2' for focus

4. Accessibility

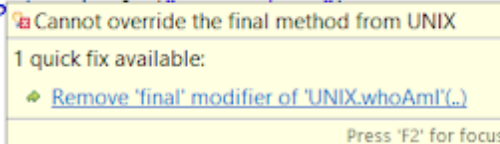
Overriding method can increase access of overridden method. This is the opposite of the earlier rule, according to this if the overridden method is declared as protected then the overriding method can be protected or public. Here is an example to see that it's allowed in Java :

```
29 class UNIX {
30
31     protected void whoAmI() {
32         System.out.println("I am UNIX");
33     }
34
35 }
36
37 class Linux extends UNIX {
38
39     public void whoAmI() {
40         System.out.println("I am Linux");
41     }
42 }
```

5. Types of Methods

The **private**, **static**, and **final methods** can not be overridden in Java. See other articles in this blog to learn why you cannot override private, static, or final methods in Java. By the way, you can hide private and static methods but trying to override the final method will result in compile-time error "Cannot override the final method from a class" as shown in the below screenshot :

```
29 class UNIX {
30
31     protected final void whoAmI() {
32         System.out.println("I am UNIX");
33     }
34
35 }
36
37 class Linux extends UNIX {
38
39     public void whoAmI() {
40         System.out.
41     }
42 }
```



6. Return Type

The return type of overriding method must be the same as overridden method. Trying to change the return type of method in the child class will throw compile-time error "return type is incompatible with parent class method" as shown in the following screenshot.