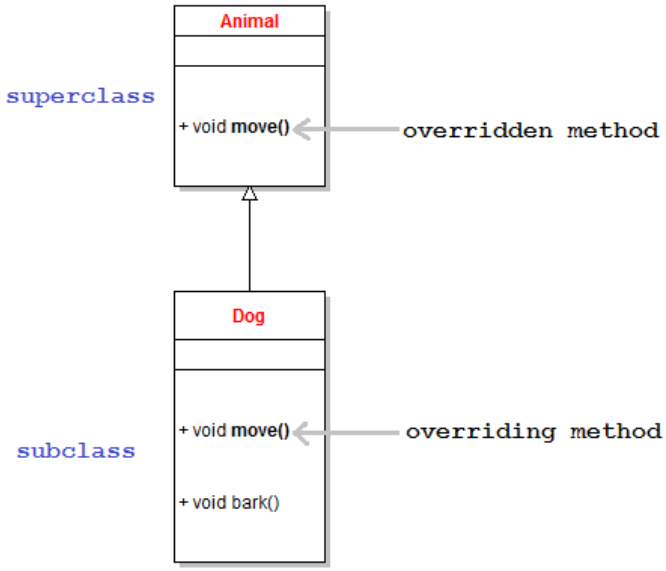
# 12 Rules of Overriding in Java You Should Know

Last Updated on 17 November 2017 |

## 1. What is Overriding?

**Overriding** refers to the ability of a subclass to re-implement an instance method inherited from a superclass. Let's take a look at the following class diagram:




Here, `Animal` is the superclass and `Dog` is the subclass, thus `Dog` inherits the `move()` method from `Animal`. However, `Dog` re-implements the `move()` method for some behaviors which are specific to only dogs (like walk and run). In this respect:

- The `Dog`'s `move()` method is called the **overriding method**.
- The `Animal`'s `move()` method is called the **overridden method**.

Basically, the overriding method must have same name and same arguments list as the overridden one. It's the way by which a subtype extends or re-defines behaviors of its supertype.

## 2. What methods can be overridden?

### Rule #1:Only inherited methods can be overridden.

Because overriding happens when a subclass re-implements a method inherited from a superclass, so only inherited methods can be overridden, that's straightforward. That means only methods declared with the following access modifiers: **public**, **protected** and default (in the same package) can be overridden. That also means **private** methods cannot be overridden. Let's see some examples:

- The `Dog` class overrides both the `move()` (public) and `eat()` (protected) methods from the `Animal` class (regardless of packages where the both classes are declared):

```
1  public class Animal {
2
3      public void move() {
```

```
1  public class Dog extends Animal {
2
3      public void move() {
```

```
        // animal moving code...
    }

    protected void eat() {
        // animal eating code...
    }
}
```

```
        // dog moving code...
    }

    protected void eat() {
        // dog eating code...
    }
}
```

- In the following example, the `Dog` class perfectly overrides the `move()` method which is declared with default access modifier in the `Animal` class, as long as both the classes are in the same package:

```
package net.codejava.core;

public class Animal {

    void move() {
        // Animal moving code...
    }
}
```

```
package net.codejava.core;

public class Dog extends Animal {

    void move() {
        // Dog moving code...
    }
}
```

- In the following example, the `Dog` and `Animal` classes are in different packages. Thus it isn't considered an overriding because the `Dog` class does not inherit the `Animal`'s `move()` method:

```
package net.codejava.animal;

public class Animal {

    void move() {
        // Animal moving code...
    }
}
```

```
package net.codejava.dog;

import net.codejava.core.animal.Animal;

public class Dog extends Animal {

    void move() {
        // Dog moving code...
    }
}
```

Here, the `Dog`'s `move()` method is just a new method, not an overriding one.

- In the following example, the `Animal`'s `move()` method is private, so the `Dog`'s `move()` method is just a new method, not an overriding one:

```
public class Animal {

    private void move() {
        // Animal moving code...
    }
}
```

```
public class Dog extends Animal {

    public void move() {
        // Dog moving code...
    }
}
```

# 3. What methods that cannot be overridden?

## Rule #2:Final and static methods cannot be overridden.

A final method means that it cannot be re-implemented by a subclass, thus it cannot be overridden. Consider the following example:

```
public class Animal {

    final void sleep() {
        // animal sleeping code...
    }
}
```

```
public class Dog extends Animal {

    public void sleep() {
        // Dog sleeping code...
    }
}
```

Here, the `Animal`'s `sleep()` method is marked as final, therefore the `Dog` class won't compile. The compiler will complain:

```
error: sleep() in Dog cannot override sleep() in Animal
    public void sleep() {
                ^
  overridden method is final
1 error
```

In case of static method, because a static method is available to all instances of the superclass and its subclasses, so it's not permissible to re-implement the static method in a particular subclass. Consider the following example:

```
public class Animal {

    static void sleep() {
        // animal sleeping code...
    }
}
```

```
public class Dog extends Animal {

    public void sleep() {
        // Dog sleeping code...
    }
}
```

The compiler will issue the following complaint when trying to compile the `Dog` class:

```
error: sleep() in Dog cannot override sleep() in Animal
    public void sleep() {
                ^
  overridden method is static
1 error
```

This book helps you improve your Java programming skills to a new level: **Effective Java (2nd Edition)**

# 4. Requirements for the overriding method

With respect to the overridden method, the overriding method must obey the following rules:

## Rule #3: The overriding method must have same argument list.

Let's see the following example:

```java
public class Animal {

    protected void eat(String food) {
        // animal eating code...
    }
}
```

```java
public class Dog extends Animal {

    protected void eat(String food) {
        // dog eating code...
    }
}
```

The `eat()` method of the `Dog` class is a legal overriding, as it keeps the same argument (String food) as the superclass' version. If we add a new argument to the method like this:

```java
protected void eat(String food, int amount) {
    // dog eating code...
}
```

Then this method is not an overriding, it is an overload instead.

## Rule #4: The overriding method must have same return type (or subtype).

Suppose that a `Food` class has a subclass called `DogFood`, the following example shows a correct overriding:

```java
public class Animal {

    protected Food seekFood() {

        // animal seeking for food code...

        return new Food();
    }
}
```

```java
public class Dog extends Animal {

    protected Food seekFood() {

        // dog seeking for food code...

        return new DogFood();
    }
}
```

It's possible to modify the return type of the `Dog`'s `seekFood()` method to `DogFood` - a subclass of `Food`, as shown below:

```java
protected DogFood seekFood() {

    // dog seeking for food code...

    return new DogFood();
}
```

That's perfectly a legal overriding, and the return type of `Dog`'s `seekFood()` method is known as **covariant return type**.

The `Dog` class won't compile if we change the `seekFood()` method's return type to another, as shown below:

```java
protected String seekFood() {

    // dog seeking for food code...

    return new String();
}
```

As the complier issues this error:

```
error: seekFood() in Dog cannot override seekFood() in Animal
    protected String seekFood() {
                     ^
  return type String is not compatible with Food
1 error
```

## Rule #5: The overriding method must not have more restrictive access modifier.

This rule can be understood as follows:

- If the overridden method is has default access, then the overriding one must be default, `protected` or `public`.
- If the overridden method is `protected`, then the overriding one must be `protected` or `public`.
- If the overridden method is `public`, then the overriding one must be only `public`.

In other words, the overriding method may have less restrictive (more relaxed) access modifier. The following example shows a legal overriding:

```
public class Animal {

    protected void move() {
        // animal moving code...
    }
}
```

```
public class Dog extends Animal {

    public void move() {
        // Dog moving code...
    }
}
```

However, in the following example, the `Dog` class won't compile:

```
public class Dog extends Animal {

    void move() {
        // Dog moving code...
    }
}
```

It is because the `move()` method now has default access, which is more restrictive than the protected access of the superclass' version.

## Rule #6: The overriding method must not throw new or broader checked exceptions.

In other words, the overriding method may throw fewer or narrower checked exceptions, or any unchecked exceptions.

Consider the following superclass - `Animal`:

```
public class Animal {

    protected void move() throws IOException {
        // animal moving code...
    }
}
```

The following subclass - `Dog`, correctly overrides the `move()` method because the `FileNotFoundException` is a subclass of the `FileIOException`:

```
public class Dog extends Animal {

    protected void move() throws FileNotFoundException {
        // Dog moving code...
    }
}
```

The following example shows an illegal overriding attempt because the `InterruptedException` is a new and checked exception:

```
public class Dog extends Animal {

    protected void move() throws IOException, InterruptedException {
        // Dog moving code...
    }
}
```

However, the following example is a legal overriding, because the `IllegalArgumentException` is an unchecked exception:

```
public class Dog extends Animal {

    protected void move() throws IOException, IllegalArgumentException {
        // Dog moving code...
    }
}
```

And in the example below, the `Dog` class won't compile because its `move()` method throws `Exception` which is superclass (broader) of the `IOException`:

```
public class Dog extends Animal {

    protected void move() throws Exception {
        // Dog moving code...
    }
}
```

.

# 5. Invoking the overridden method

### Rule #7:Use the super keyword to invoke the overridden method from a subclass.

It's very common that a subclass extends a superclass' behavior rather than re-implementing the behavior from scratch. In such case, invoke the superclass' method in the following form:

**`super.overriddenMethodName()`**

Consider the following example:

```
public class Animal {

    protected void move() {
        // animal moving code...
    }
}
```

```
public class Dog extends Animal {

    protected void move() {
        super.move();   // Animal movement

        // Dog-specific moving code...
    }
}
```

Here, the `Dog` class overrides the `move()` method from the `Animal` class. Then in the `Dog`'s `move()` method, it calls the superclass' version of the method first, then add behaviors specific to only dogs.

# 6. Overriding and constructor

### Rule #8:Constructors cannot be overridden.

Because constructors are not methods and a subclass' constructor cannot have same name as a superclass' one, so there's nothing relates between constructors and overriding.

# 7. Overriding and abstract method

### Rule #9: Abstract methods must be overridden by the first concrete (non-abstract) subclass.

Consider the following interface:

```
public interface Animal {
    void move();
}
```

If an abstract class implements the above interface, then it doesn't require the subclass to override the `move()` method, as shown in the following `AbstractDog` class:

```
public abstract class AbstractDog implements Animal {

    protected abstract void bark();

}
```

But if a concrete (non-abstract) class, says `BullDog`, is a subclass of the `AbstractDog` class or the `Animal` interface, then it must override all the inherited abstract methods, as shown below:

```
public class BullDog extends AbstractDog {

    public void move() {

        // Bulldog moves...

    }

    protected void bark() {

        // Bulldog barks...

    }

}
```

In this respect, the `BullDog` class is said to *implement* the `move()` and `bark()` abstract methods of its supertypes - the `Animal` interface and the `AbstractDog` class. Although all the rules of overriding must be obeyed in this context, the term *implement* is more exact then the term *override*, since the overridden method is abstract.



# 8. Overriding and static method

## Rule #10: A static method in a subclass may hide another static one in a superclass, and that's called hiding.

Consider the following example:

```
public class Animal {

    static void sleep() {
        System.out.println("Animal sleeps");
    }

}
```

```
public class Dog extends Animal {

    static void sleep() {
        System.out.println("Dog sleeps");
    }
}
```

Here, the `sleep()` method of the `Dog` class is said to *hide* the `sleep()` method of the `Animal` class. When a static method of the superclass is hidden, it requires the subclass to use a fully qualified class name of the superclass to invoke the hidden method, as shown in the `doSomething()` method of the `Dog` class below:

```
public class Dog extends Animal {

    static void sleep() {
        System.out.println("Dog sleeps");
    }

    void doSomething() {
         sleep();           // this calls the hiding method

```

```
        // because the Animal's sleep() is hidden, it requires to use
        // a fully qualified class name to access it.
        Animal.sleep();
    }
}
```

Note that the rules of overriding are still applied for the hiding method.

# 9. Overriding and synchronized method

### Rule #11: The synchronized modifier has no effect on the rules of overriding.

The **synchronized** modifier relates to the acquiring and releasing of a monitor object in multi-threaded context, therefore it has totally no effect on the rules of overriding. That means a synchronized method can override a non-synchronized one and vice versa.

# 10. Overriding and strictfp method

### Rule #12: The strictfp modifier has no effect on the rules of overriding.

That means the presence or absence of the **strictfp** modifier has absolutely no effect on the rules of overriding: it's possible that a FP-strict method can override a non-FP-strict one and vice-versa.

## References

- Overriding and Hiding Methods