# Theory: Hiding and overriding



32 minutes

Verify to skip | Start practicing

## §1. Overriding instance methods

Java provides an opportunity to declare a method in a subclass with the same name as a method in the superclass. This is known as **method overriding**. The benefit of overriding is that a subclass can give its own specific implementation of a superclass method.

**Overriding methods** in subclasses allows a class to inherit from a superclass whose behavior is **"close enough"** and then to change this behavior as the subclass needs.

Instance methods can be overridden if they are inherited by the subclass. The overriding method must have the same name, parameters (number and type of parameters), and the return type (or a subclass of the type) as the overridden method.

**Example.** Here is an example of overriding.

```
class Mammal {

    public String sayHello() {
        return "ohlllalalalalalaoaoaoa";
    }
}

class Cat extends Mammal {

    @Override
    public String sayHello() {
        return "meow";
    }
}

class Human extends Mammal {

    @Override
    public String sayHello() {
        return "hello";
    }
}
```

The hierarchy includes three classes: `Mammal`, `Cat` and `Human`. The class `Mammal` has the method `sayHello`. Each subclass overrides this method. The `@Override` annotation indicates that the method is overridden. This annotation is optional but helpful.

Let's create instances and invoke the method.

```
Mammal mammal = new Mammal();

System.out.println(mammal.sayHello()); // it prints "ohlllalalalalalaoaoaoa"

Cat cat = new Cat();
System.out.println(cat.sayHello()); // it prints "meow"

Human human = new Human();
System.out.println(human.sayHello()); // it prints "hello"
```

As you can see, each subclass has its own implementation of the method `sayHello`.

3 required topics

- Annotations basics
- Static members (In project)
- The keyword super

4 dependent topics

- Polymorphism
- Covariant return types
- toString()
- hashCode() and equals()

You can invoke the base class method in the overridden method using the keyword `super`.

## §2. Rules for overriding methods

There are several rules for methods of subclasses which should override methods of a superclass:

- the method must have the same name as in the superclass;
- the arguments should be exactly the same as in the superclass method;
- the return type should be the same type or a subtype of the return type declared in the method of the superclass;
- the access level must be the same or more open than the overridden method's access level;
- a private method cannot be overridden because it's not inherited by subclasses;
- if the superclass and its subclass are in the same package, then package-private methods can be overridden;
- static methods cannot be overridden.

To verify these rules, there is a special annotation `@Override`. It allows you to know whether a method will be actually **overridden** or not. If for some reason, the compiler decides that the method cannot be overridden, it will generate an error. But, remember, the annotation is not required, it's only for convenience.

## §3. Forbidding overriding

If you'd like to forbid overriding of a method, declare it with the keyword `final`.

```
1  public final void method() {
2      // do something
3  }
```

Now, if you try to override this method in a subclass, a compile-time error will occur.

## §4. Overriding and overloading methods together

Recall, that **overloading** is a feature that allows a class to have more than one method with the same name, if their arguments are different.

We can also override and overload an instance method in a subclass at the same time. Overloaded methods do not override superclass instance methods. They are new methods, unique to the subclass.

The following example demonstrates it.

```
class SuperClass {

    public void invokeInstanceMethod() {
        System.out.println("SuperClass: invokeInstanceMethod");
    }
}

class SubClass extends SuperClass {

    @Override
    public void invokeInstanceMethod() {

System.out.println("SubClass: invokeInstanceMethod is overridden");
    }

    // @Override -- method doesn't override anything
    public void invokeInstanceMethod(String s) {

System.out.println("SubClass: overloaded invokeInstanceMethod");
    }
}
```

The following code creates an instance and calls both methods:

```
 SubClass clazz = new SubClass();


clazz.invokeInstanceMethod();    // SubClass: invokeInstanceMethod() is overr

clazz.invokeInstanceMethod("s"); // SubClass: overloaded invokeInstanceMethod
```

Remember, overriding and overloading are different mechanisms but you can mix them together in one class hierarchy.

## §5. Hiding static methods

Static methods cannot be overridden. If a subclass has a static method with the same signature (name and parameters) as a static method in the superclass then the method in the subclass hides the one in the superclass. It's completely different from method overriding.

You will get a compile-time error if a subclass has a static method with the same signature as an instance method in the superclass or vice versa. But if the methods have the same name but different parameters there should be no problems.