Overriding and Hiding Methods

Instance Methods

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a covariant return type.

When overriding a method, you might want to use the `@Override` annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error. For more information on `@Override`, see the section Annotations.

Static Methods

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.
- Consider an example that contains two classes. The first is `Animal`, which contains one instance method and one static method:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```

The second class, a subclass of `Animal`, is called `Cat`:

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
```

```
11          Animal myAnimal = myCat;
12          Animal.testClassMethod();
13          myAnimal.testInstanceMethod();
14      }
15  }
```

The `Cat` class overrides the instance method in `Animal` and hides the static method in `Animal`. The main method in this class creates an instance of `Cat` and invokes `testClassMethod` on the class and `testInstanceMethod` on the instance.

The output from this program is as follows:

```
1  The static method in Animal
2  The instance method in Cat
```

As promised, the version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

Interface Methods

Default methods and abstract methods in interfaces are inherited like instance methods. However, when the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict. These rules are driven by the following two principles.

- Instance methods are preferred over interface default methods.

Consider the following classes and interfaces:

```
1  public class Horse {
2      public String identifyMyself() {
3          return "I am a horse.";
```

```
    }
}

public interface Flyer {
    default public String identifyMyself() {
        return "I am able to fly.";
    }
}

public interface Mythical {
    default public String identifyMyself() {
        return "I am a mythical creature.";
    }
}

public class Pegasus extends Horse implements Flyer, Mythical {
    public static void main(String... args) {
        Pegasus myApp = new Pegasus();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Pegasus.identifyMyself()` returns the string `I am a horse`.

- Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.

Consider the following interfaces and classes:

```
public interface Animal {
    default public String identifyMyself() {
        return "I am an animal.";
    }
}

public interface EggLayer extends Animal {
    default public String identifyMyself() {
        return "I am able to lay eggs.";
    }
}

```

```
public interface FireBreather extends Animal { }

public class Dragon implements EggLayer, FireBreather {
    public static void main (String... args) {
        Dragon myApp = new Dragon();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Dragon.identifyMyself()` returns the string `I am able to lay eggs`.

If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. You must explicitly override the supertype methods.

Consider the example about computer-controlled cars that can now fly. You have two interfaces (`OperateCar` and `FlyCar`) that provide default implementations for the same method, (`startEngine()`):

```
public interface OperateCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}

public interface FlyCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
```

A class that implements both `OperateCar` and `FlyCar` must override the method `startEngine()`. You could invoke any of the of the default implementations with the `super` keyword.

```
public class FlyingCar implements OperateCar, FlyCar {
    // ...
    public int startEngine(EncryptedKey key) {
        FlyCar.super.startEngine(key);
        OperateCar.super.startEngine(key);
    }
}
```

The name preceding `super` (in this example, `FlyCar` or `OperateCar`) must refer to a direct superinterface that defines or inherits a default for the invoked method. This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature. You can use the `super` keyword to invoke a default method in both classes and interfaces.

Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

```
public interface Mammal {
    String identifyMyself();
}

public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}

public class Mustang extends Horse implements Mammal {
    public static void main(String... args) {
        Mustang myApp = new Mustang();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Mustang.identifyMyself()` returns the string `I am a horse`. The class `Mustang` inherits the method `identifyMyself()` from the class `Horse`, which overrides the abstract method of the same name in the interface `Mammal`.

> *Note: Static methods in interfaces are never inherited.*

Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a `protected` instance method in the superclass can be made `public`, but not `private`, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

Summary

The following table summarizes what happens when you define a method with the same signature as a method in a superclass.

	Superclass Instance Method	**Superclass Static Method**
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

> *Note: In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.*

Last update: *September 14, 2021*