Overriding Methods

A subclass inherits all public methods from its superclass, and these methods remain public in the subclass. We'll also usually add more methods or instance variables to the subclass–that's why we have subclasses. If we didn't need to add anything, we might stick with the superclass. Where things get interesting is when we need to modify existing inherited methods. This is called **overriding** methods.

Overriding an inherited method means providing a public method in a subclass with the same **method signature** (method name, parameter type list and return type) as a public method in the superclass. The method in the subclass will be called ***instead of*** the method in the superclass. One common method that is overridden is the `toString()` method. The example below shows an example method called `greet()`.

#Coding Exercise

In the following example the `MeanGreeter` inherits the `greet()` method from `Greeter`, but then overrides it. Run the program to see.

Add another subclass called `SpanishGreeter` that extends Greeter and overrides the `greet()` method to return “Hola!” instead of “Hi!”. Then create a `SpanishGreeter` object to test it out in the Greeter's `main()` method.

```
public class Greeter {

   public String greet() {
      return "Hi";
   }
   public static void main(String[] args) {
      Greeter g1 = new Greeter();
      System.out.println(g1.greet());
      Greeter g2 = new MeanGreeter();
      System.out.println(g2.greet());
      SpanishGreeter g3 = new SpanishGreeter();
      System.out.println(g3.greet());
   }
}
```

CONSOLESHELL

InfoWarningTip
To override an inherited method, the method in the child class must have the exact same name, parameter list, and return type (or a subclass of the return type) as the parent method. Any method that is called must be defined within its own class or its superclass.

You may see the `@Override` annotation above a method. This is optional, but it provides an extra compiler check that you have matched the method signature exactly. It also helps your future self or your fellow developers spot that you're overriding a method in the parent class.

CopyC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
@Override
public String greet() {
return "Go Away";
}
```

#Overloading Methods

Don't get **overriding** a method confused with **overloading** a method! **Overloading** a method is when several methods have the same name, but the return type, parameter types, order, or number are different. With **overriding**, the method signatures look identical, but they are in different classes; in **overloading**, only the method names are identical, and they have different parameters.

CopyC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
2
3
// overriding methods
g2.greet(); // This could be calling an overriden greet method
in g2's class
g1.greet("Sam"); // This calls an overloaded greet method
```

#Coding Exercise

In the example below the `greet(String who)` method overloads the `greet()` method of `Greeter`. Notice that `MeanGreeter` inherits this method as it isn't overridden. Override the `greet(String who)` method in the `MeanGreeter` class in the `MeanGreeter.java` file to return `"Go away "` `+` the `who` String.

2

Run

Greeter.java MeanGreeter.java

```
public class Greeter {

   public String greet() {
      return "Hi";
   }

   public static void main(String[] args) {
      Greeter g1 = new Greeter();
      System.out.println(g1.greet());
      Greeter g2 = new MeanGreeter();
      System.out.println(g2.greet());

      SpanishGreeter g3 = new SpanishGreeter();
      System.out.println(g3.greet());
   }
}
public class MeanGreeter extends Greeter {

   @Override
   public String greet() {
      return "Go Away";
   }
}
public class SpanishGreeter extends Greeter {

   @Override
   public String greet() {
      return "Hola!";
```

```
    }
}
```

InfoWarningTip
To overload a method the method must have the same name, but the parameter list must be different in some way. It can have a different number of parameters, different types of parameters, and/or a different order for the parameter types. The return type can also be different.

#Knowledge Check

Which of the following declarations in the `Student` class below would correctly *override* the `getFood` method in `Person`?

```
public class Person {
private String name = null;

public Person(String theName) {
name = theName;
}

public String getFood() {
return "Hamburger";
}
}

public class Student extends Person {
private int id;
private static int nextId = 0;

public Student(String theName) {
super(theName);
id = nextId;
nextId++;
}

public int getId() {return id;}

public void setId (int theId) {
this.id = theId;
}
```

```
}
```

public void getFood()

public String getFood(int quantity)

public String getFood()

You can see this example in the Java Visualizer by clicking on the following link: Override Example.

#

Which of the following declarations in `Person` class would correctly *overload* the `getFood` method in `Person`?

```
public class Person {
private String name = null;

public Person(String theName) {
name = theName;
}

public String getFood() {
return "Hamburger";
}
}

public class Student extends Person {
private int id;
private static int nextId = 0;

public Student(String theName) {
super(theName);
id = nextId;
nextId++;
}

public int getId() {return id;}
public void setId (int theId) {
this.id = theId;
}
}
```

public void getFood()

public String getFood()

public String getFood(int quantity)

https://youtu.be/kArGE1-vRrw

#Inherited Get/Set Methods

Inheritance means that an object of the child class automatically includes the object instance variables and methods defined in the parent class. But, if the inherited instance variables are `private`, which they should be, the child class can not directly access them using dot `.` notation. The child class can use public **accessors** (also called getters or get methods) which are methods that **get** instance variable values, and public **mutators** (also called modifier methods or setters or set methods), which **set** their values.

For example, if a parent has **private** instance variables, `name`, then the parent typically provides a **public** `getName` method and a **public** `setName` method as shown below. In the `setName` method below, the code checks if the passed string is `null` before it sets it and returns `true` if the set was successful or `false` otherwise. The `Employee` class inherits the `name` field but must use the public method `getName` and `setName` to access it.

Take a look: (the code below is for you to observe what was described above, so zero points are assigned)
2
Run
Employee.java Person.java

```
public class Employee extends Person {

   private static int nextId = 1;
   private int id;

   public Employee() {
      id = nextId;
      nextId++;
   }

   public int getId() {
```

```
      return id;
   }

   public static void main(String[] args) {
      Employee emp = new Employee();
      emp.setName("Dina");
      System.out.println(emp.getName());
      System.out.println(emp.getId());
   }
}
public class Person {
   private String name;

   public String getName() {
      return name;
   }

   public boolean setName(String theNewName) {
      if (theNewName != null) {
         this.name = theNewName;
         return true;
      }
      return false;
   }
}
```

#Check your understanding

Given the following class definitions, which of the following options would *not* compile if it was used in place of the missing code in the `main()` method?

```
class Item {
private int x;

public void setX(int theX) {
x = theX;
}
// ... other methods not shown
}

public class EnhancedItem extends Item {

private int y;

public void setY(int theY) {
y = theY;
}

// ... other methods not shown
public static void main(String[] args) {
```

```
EnhancedItem currItem = new EnhancedItem();
// Missing code
}
}
```

currItem.setX(3);

currItem.setY(2);

currItem.x = 3;

currItem.y = 2;

You can step through this code in the Java Visualizer by clicking on the following link Private Fields Example.

#Programming Challenge: Pet Sounds

Pets

The following `Pet` class keeps track of a pet's name and the type of pet it is. It has a constructor, getter methods, and a method called `speak()` that prints an animal noise.

1. Write a subclass called `Dog` that inherits from `Pet`.
2. Write the Dog class' constructor with a String parameter called `name` that initializes the class attribute name inherited from the parent `Pet`. In this constructor, call the super constructor, passing the name and the animal type "dog" to it.
3. Override the method `speak()` in the `Dog` class to print out a barking sound "Woof!". (Do not override the getter methods. The superclass getters should work for all subclasses).
4. Uncomment the `Dog` object in the `main` method to test it out.
5. Write a similar `Cat` class inherited from `Pet` with a similar constructor with animal type "cat" and overrides the method `speak()` with a "Meow!". Test it out.

Complete the Dog and Cat classes below to inherit from Pet with a constructor and a method speak() that prints out "Woof!" or "Meow!".

3

Run

Pet.java Dog.java Cat.java

```
public class Pet {
    private String name;
    private String type;

    public Pet(String n, String t) {
        name = n;
        type = t;
    }
    public String getType() {
        return type;
    }
    public String getName() {
        return name;
    }

    public String speak() {
        return "grr!";
    }
    //Do not modify this code
    public static void main(String[] args) {
```

```
        Pet p = new Pet("Sammy", "hamster");
        System.out.println(p.getType());
        System.out.println(p.speak());

        Dog d = new Dog("Fido");
        System.out.println(d.getType());
        System.out.println(d.speak());

        Cat c = new Cat("Fluffy");
        System.out.println(c.getType());
        System.out.println(c.speak());

    }
}
class Dog extends Pet {
    public Dog(String name) {
        super(name, "dog");
    }

    @Override
    public String speak() {
        return "Woof!";
    }
}
class Cat extends Pet {
    public Cat(String name) {
        super(name, "cat");
    }

    @Override
    public String speak() {
        return "Meow!";
    }
}
```

#Summary

- Method **overriding** occurs when a public method in a subclass has the same method signature as a public method in the superclass.
- Any method called must be defined within its own class or superclass.
- A subclass is usually designed to have modified (overridden) or additional methods or instance variables.

- A subclass will inherit all public methods from the superclass (for example, all the setter and getter methods); these methods remain public in the subclass.
- **Overloading** a method is when several methods have the same name, but the parameter types, order, or number are different.

#Extra Resources

The following YouTube video might be a bit too advanced in some places, but part of becoming a dev is learning to watch things you aren't quite ready for and still getting what you need from them. The comparison table toward the end of the video is particularly helpful if you need a review of the differences between overriding and overloading methods.

Open in new tab
https://youtu.be/ryDeTfmSY_o

#Method Overloading

Overloading is when there are different methods with the same name but different definitions - either by having a different number of parameters or parameter types.

#Problem Statement

Complete the tasks below in the `Add` Class. Don't forget to create a new class in your Eclipse IDE, too:

1. Define the method add() that accepts **two integers** and adds the numbers.
2. Define the method `add()` that accepts **three integers** and adds the numbers.
3. Define the method `add()` that accepts **two doubles** and adds the numbers.

```
public class Add {

    // Define the method add() that accepts two integers and adds the numbers.
    public int add(int a, int b) {
        return a + b;
    }
```

```
    // Define the method add() that accepts three integers and adds the number
s.
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Define the method add() that accepts two doubles and adds the numbers.
    public double add(double a, double b) {
        return a + b;
    }

    // Do not modify the code below. The code below gives you the idea of how
the different methods are called.
    public static void main(String args[]) {
        Add obj = new Add();
        System.out.println(obj.add(12, 21));
        System.out.println(obj.add(11, 23, 10));
        System.out.println(obj.add(100.10, 200.5));
    }
}
```

Method Overriding

Method overriding occurs when a subclass provides its own implementation of a method declared by one of its parent classes.

#Problem Statement

You are provided with the starter code below to practice Inheritance and Overriding Methods. In the code below, the `Car` class extends the `Vehicle` class, which means `Car` is also a type of `Vehicle` with its own vehicle properties.

Complete the code below by following the given steps.

#Steps to follow:

Don't forget to create a sub-package with all of the necessary files in your Eclipse IDE. When you've accomplished the goal, paste your code over here to Coding Rooms, so we can check it.

1. Fill in the `Vehicle` class in the file provided:
    a. Declare a String `color` and a String `brand` variables in the `Vehicle` class.

b. Define a parameterized constructor with color and brand as parameters in the `Vehicle` class.
c. Create a print() method in the Vehicle class.

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
public void print() {
System.out.println(this.color + " " + this.brand);
}
```

2. Create a `Car` class that extends the `Vehicle` class:
   a. Declare a String `steeringWheel` variable in the `Car` class.
   b. Define a parameterized constructor with `color`, `brand`, and `steeringWheel` as parameters in the `Car` class. Call the parent class constructor with the **super** keyword, i.e. `super(color, brand)`, to initialize the `color` and `brand` variables.
   c. Create a print() method in the Car class, which calls the parent's class print method.

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
public void print() {
super.print();
```

```
}
```

3. Create a `Bike` class that extends the `Vehicle` class.
   a. Declare a String `bikeHandle` variable in the `Bike` class.
   b. Define a parameterized constructor with `color`, `brand`, and `bikeHandle` as parameters in the Bike class. Call the parent class constructor with the **super** keyword, i.e. `super(color, brand)`, to initialize the `color` and `brand` variables.
   c. Create a `print()` method in the `Bike` class, which calls the parent class's print method.

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
public void print() {
super.print();
}
```

4

Run

TestClass.java Car.java Vehicle.java Bike.java

```
public class TestClass {
```

```
   // Use the code below for Sanity check. Do not Modify the code below for te
sting purpose.
   public static void main(String args[]) {
        Car car = new Car("White", "Audi", "ChromeColored");
        car.print();
        Bike bike = new Bike("Grey", "Bianchi", "SilverColored");
        bike.print();
   }
}
```

CONSOLESHELL

Now, Let's override the `print()` method of the Super/Parent Class. Feel free to copy/paste your code from the above IDEs into the one below, but be ready to edit each file to achieve the new goals.

#Override the `print()` Method: Steps to be followed

4. Copy and paste the code below into the code you wrote previously:
5. Override the `print()` method inherited from the `Vehicle` class in the `Car` class.

   CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
@Override
public void print() {
System.out.println("I am Car");
}
```

6. Override the `print()` method inherited from the `Vehicle` class in the `Bike` class.

CopyC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
@Override
public void print() {
System.out.println("I am Bike");
}
```

4

Run

Main.java Vehicle.java Car.java Bike.java

```
public class Main {
   // Use the code below for Sanity check. Do not Modify the code below for testing purpose.
   public static void main(String args[]) {
       Car car = new Car("White", "Audi", "ChromeColored");
       car.print();
       Bike bike = new Bike("Grey", "Bianchi", "SilverColored");
       bike.print();
   }
}
```

```
public class Vehicle {
    protected String color;
    protected String brand;

    public Vehicle(String color, String brand) {
        this.color = color;
        this.brand = brand;
    }

    public void print() {
        System.out.print(this.color + " " + this.brand + " ");
    }
}
 public class Car extends Vehicle {

    private String steeringWheel;

    public Car(String color, String brand, String steeringWheel) {
        super(color, brand);
        this.steeringWheel = steeringWheel;
    }

    @Override
    public void print() {
        System.out.println("I am Car");
    }
}
 public class Bike extends Vehicle {

    private String bikeHandle;

    public Bike(String color, String brand, String bikeHandle) {
        super(color, brand);
        this.bikeHandle = bikeHandle;
    }

    @Override
    public void print() {
        System.out.println("I am Bike");
    }
}
```

Which annotation do we use to point out method overriding?

@Override

@Overiding

@Overloading

@Overrided

#Reference:

[#6.1 Java Tutorial | Inheritance]

Back
Unsubmit
Next