
ORACLE
Academy

Oracle Academy
Java for AP Computer Science A

10-2

Polymorphism

ORACLE
Academy

Objectives

- This lesson covers the following objectives:
  - Apply superclass references to subclass objects
  - Write code to override methods
  - Use dynamic method dispatch to support polymorphism
  - Create abstract methods and classes
  - Recognize a correct method override

Overview

- This lesson covers the following topics:
  - Use the final modifier
  - Explain the purpose and importance of the Object class
  - Write code for an applet that displays two triangles of different colors
  - Describe object references

Review of Inheritance

- When one class inherits from another, the subclass "is-a" type of the superclass
- Objects of a subclass can be referenced using a superclass reference, or type

Learn More

- Visit Oracle's tutorial pages to learn more:
- Inheritance:
  - http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
- Polymorphism:
  - http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

Inheritance Example

- If classes are created for a Bicycle class and a RoadBike class that extends Bicycle, a reference of type Bicycle can reference a RoadBike object
  - Because RoadBike "is-a" type of Bicycle, it is perfectly legal to store a RoadBike object as a Bicycle reference
  - The type of a variable (or reference) does not determine the actual type of the object hat it refers to

Inheritance Example

- Therefore, a Bicycle reference, or variable, may or may not contain an object of the superclass type Bicycle since it can contain any subclass of Bicycle

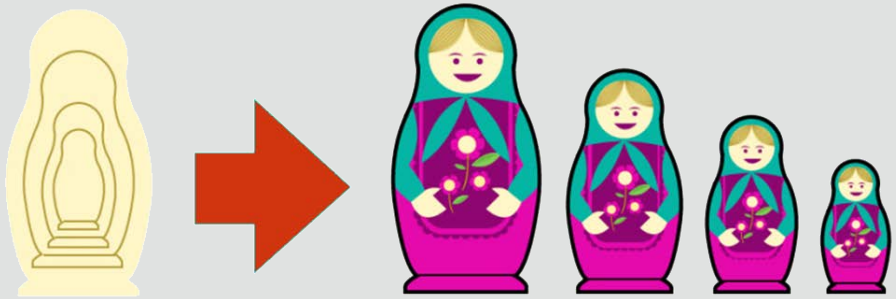
```
Bicycle bike = new RoadBike();
```

Polymorphism

- When a variable or reference may refer to different types of objects is called polymorphism
- Polymorphism is a term that means "many forms"
- In the case of programming, polymorphism allows variables to refer to many different types of objects, meaning they can have multiple forms
- For example, because RoadBike "is-a" Bicycle, there are two possible references that define the type of object it is (Bicycle or RoadBike)

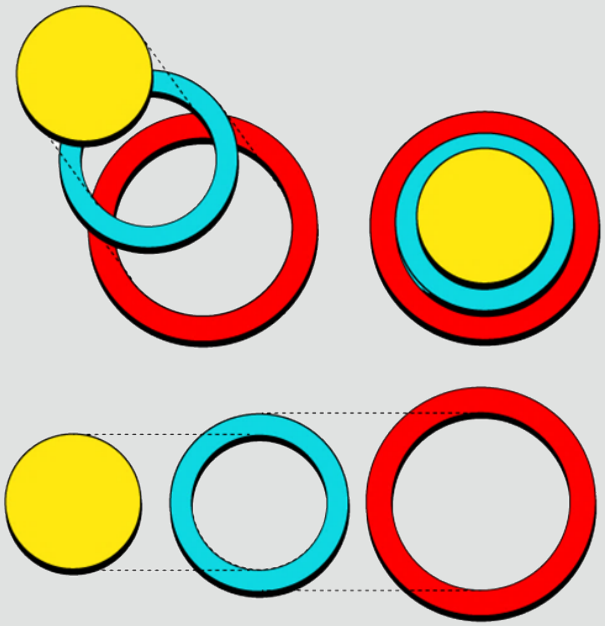
Polymorphism and Nesting Dolls

- Polymorphism can be visualized like a set of nesting dolls: The set of dolls share a type and appearance, but are all unique in some way
- Each doll has the same shape, with a size determined by the doll it must fit inside of
- Each smaller doll is stored inside of the next larger doll
- From the outside you do not see the smaller dolls, but you can open each doll to find a smaller doll inside



Superclasses and Subclasses

- In a similar way, subclasses can "fit" within the reference type of a superclass
- A superclass variable can hold, or store, a subclass's object, while looking and acting like the superclass



Superclass Variables

- If you "open up" a superclass variable, or invoke one of its methods, you will find that you actually have a subclass object stored inside
- For example
  - With nesting dolls, you cannot see the smaller doll until you open the larger doll
  - Its type may be ambiguous
  - When Java code is compiled, Java does not check to see what type (supertype or subtype) of object is inside a variable
  - When Java code is executed, Java will "open up" to see what type of object is inside the reference, and call methods that are of that type

Uncertainty when Referencing Objects

- There are benefits to uncertainty when referencing different objects at compile time
- For example:
  - You write a program that calculates the different tube lengths of a bicycle frame, given a rider's measurements and type of bicycle desired (RoadBike or MountainBike)
  - You want a list to keep track of how many bike objects you have built
  - You want only one list, not two separate lists for each type of bike
  - How do you build that list? An array, perhaps?
  - What's problematic about using an array to build the list?

Why Not Use an Array of Class Objects?

- Arrays are a collection of elements of the same type, such as a collection of integers, doubles, or Bicycles
- While it is possible to have an array of objects, they must be the same type
- This is fine for classes that are not extended

Why Not Use an Array of Class Objects?

- Polymorphism solves this problem
- Since RoadBike and MountainBike are both types of Bicycle objects, use an array of Bicycle references to store the list of bikes you have built
- Either type of Bicycle can be added to this array

```
Bicycle[] bikes = new Bicycle[size];
```

Object References

- The Object class is the highest superclass in Java, since it does not extend another class
- As a result, any class can be stored in an Object reference

```
Object[] objects = new Bicycle[size];
```

- In this array example, it is also valid to store our bikes in an array of Object references
- However, this makes our array type even more ambiguous, and should be avoided unless there is a reason to do so

Overriding Methods from Object

- Since Object is the superclass of all classes, we know that all classes inherit methods from Object
- Two of these methods are very useful, such as the equals() method and the toString() method
- The equals() method allows us to check if two references are referencing the same object
- The toString() method returns a String that represents the object
- The String provides basic information about the object such as it's class, name, and a unique hashcode

Overriding or Redefining Methods

- Although the **equals()** and **toString()** methods are useful, they are missing functionality for more specific use
- For the Bicycle class, we may want to generate a String containing the model number, color, frame type, and price
- Using the method from Object will not return this information
- The toString() method in the Object class returns a String representation of the object's location in memory
- Rather than creating a method by another name, we can override the toString() method, and redefine it to suit our needs

Overriding Methods

- Overriding methods is a way of redefining methods with the same return type and parameters by adding, or overriding the existing logic, in a subclass method
- Overriding is different than overloading a method
- Overloading a method means the programmer keeps the same name (i.e., toString()), but changes the input parameters (method signature)
- Overriding essentially hides the parent's method with the same signature, and it will not be invoked on a subclass object unless the subclass uses the keyword `super`

Overriding Methods

- Overriding does not change the parameters
- It only changes the logic inside the method defined in the superclass

Java Tutorials on Overriding Methods

- Visit Oracle's Java tutorials for more information on overriding methods:
  - http://docs.oracle.com/javase/tutorial/java/IandI/override.html

Overriding equals()

- By default, two object references are considered equal ONLY when they both **reference the same object**
- For example, given the following code for class Bicycle:

```
public class Bicycle {
    private int modelNum;
    private String color;
    private int frameType;
    private double price;

    public Bicycle(int m, String c, int f, double p){
        modelNum = m;
        color = c;
        frameType = f;
        price = p
    }
}
```

Overriding equals()

- Then implemented in a main method as below:

```
Bicycle bike1 = new Bicycle(321, "red", 3, 300.00);
Bicycle bike2 = new Bicycle(321, "red", 3, 300.00);
System.out.println(bike1.equals(bike2));
```

- The result of the output is **`false`** – WHY?
  - Although bike1 and bike2 appear to be identical, they are unique and different objects

Overriding equals()

- However, if the equals() method is overridden to compare each and every variable of the bicycle class, and then added to the Bicycle class as follows:

```
public boolean equals(Bicycle otherObject){
     return (modelNum == otherObject.modelNum &&
             color.equals(otherObject.color) &&
             frameType == otherObject.frameType &&
            price == otherObject.price);
 }
```

Overriding equals()

- The result of the output is now **`true`** because the values of the parameters are compared in the overridden equals() method in the Bicycle class

```
Bicycle bike1 = new Bicycle(321, "red", 3, 300.00);
Bicycle bike2 = new Bicycle(321, "red", 3, 300.00);
System.out.println(bike1.equals(bike2));
```

Overriding toString()

- We can override toString() to return a String that provides information about the object instead of the location of the object in memory
  - First, start with the prototype:

```
public String toString()
```

  - There is no reason to change the return type or parameters, so we will override toString()
  - Given our private data (model number, color, frame type, and price) we can return the following String:

```
return "Model: " + modelNum + " Color: " + color +
       " Frame Type: " + frameType + " Price: " + price;
```

Overriding toString()

- The result for our overridden toString() method:

```
public String toString(){
  return "Model      : " + modelNum +
         "Color      : " + color +
         "Frame Type : " + frameType +
         "Price      : " + price;
}//end method toString
```

- It is very common and very helpful when creating Java classes to override the toString() method to test your methods and data

Overriding equals() and toString()

- Consider the Animal class defined here:

```
public class Animal {
    private String type;
    private int weight;

    public Animal(String t, int w){
        type = t;
        weight = w;
    }

    public boolean equals(Animal otherObject){
        return type.equals(otherObject.type) && weight == otherObject.weight;
    }

    public String toString(){
        return "This animal is a " + type + ", and weights " + weight + " pounds";
    }
}
```

Overriding equals() and toString()

- What is the output of the following?

```
Animal a1 = new Animal("Horse", 1000);
Animal a2 = new Animal("Cow" , 1000);
Animal a3 = new Animal("Cow", 1000);

System.out.println(a1.toString());
System.out.println(a2.toString());
System.out.println(a3.toString());

System.out.println(a1.equals(a2));
System.out.println(a1.equals(a3));
System.out.println(a2.equals(a3));

System.out.println(a1 == a2);
System.out.println(a1 == a3);
System.out.println(a2 == a3);
```

Understanding the Object Model

- Polymorphism, like inheritance, is central to the object model and object-oriented programming
- Polymorphism provides for versatility in working with objects and references while keeping the objects discrete or distinct
- At the heart of the philosophy, the object model turns programs into a collection of objects versus a set of tasks, encapsulating the data and creating smaller pieces of a program, rather than a single large chunk of code

Object Model Goals

- The object model has several goals:
  - Data abstraction
  - Protecting information and limiting other classes' ability to change or corrupt data
  - Concealing implementation
  - Providing modular code that can be reused by other programs or classes

Polymorphism and Methods

- How are the methods in the subclass affected by polymorphism?
- Remember, subclasses may inherit methods from their superclasses
- If a Bicycle variable can hold a subclasses' object type, how does Java know which methods to invoke when an overridden method is called?

Polymorphism and Methods

- Methods called on a reference (bike) will always refer to methods within the object's (RoadBike) type

```
Bicycle bike = new RoadBike();
```

- Imagine that our Bicycle class contains a method setColor(Color color) to set the color of the bike the rider wants
- RoadBike inherits this method
- What happens when we do the following?

```
bike.setColor(new Color(0, 26, 150));
```

Dynamic Method Dispatch

- Java is able to determine which method to invoke based on the type of the object being referred to at the time the method is called
- Dynamic Method Dispatch, also known as Dynamic Binding, allows Java to correctly and automatically determine which method to invoke based on the reference type and the object type

Abstract Classes

- Is it really necessary to define a Bicycle class if we are only going to create objects of its subclasses:
  - roadBikes
  - mountainBikes?
- Abstract classes are one alternative that addresses this concern
- An abstract class is one that cannot be instantiated:
  - This means that you cannot create objects of this type
  - It is possible to create variables, or references of this type

Abstract Classes

- If we declare the Bicycle class to be abstract, we can still use the syntax below, but we cannot actually create a Bicycle object
- This means all references of type Bicycle will reference subclass objects MountainBike or RoadBike

```
Bicycle bike  = new RoadBike();
```

```
Bicycle bike2 = new mountainBike();
```

Abstract Classes

- Abstract classes can contain fully-implemented methods that they "pass on" to any class that extends them
- Make a class abstract by using the keyword abstract

```
public abstract class Bicycle
```

Abstract Classes

- Abstract classes can also declare at least one abstract method (method that does not contain any implementation
- This means the subclasses must use the method prototype (outline) and must implement these methods
- Abstract methods are declared with the abstract keyword

```
abstract public void setPrice();
```

Declare as abstract public. Do not use {}.

Abstract Methods

- Abstract methods:
  - Cannot have a method body
  - Must be declared in an abstract class
  - Must be overridden in a subclass
- This forces programmers to implement and redefine methods
- Typically, abstract classes contain abstract methods, partially implemented methods, or fully-implemented methods

Partially Implemented Methods

- Recall that subclasses can call their superclass's constructor and methods using the keyword super
- With abstract classes, subclasses can also use super to use their superclass's method
- Typically, this is done by first overriding the superclass's method, then calling the super, or overridden method, and then adding code
- For example:
  - let's override the equals() method from the abstract class Bicycle, which is implemented partially
  - This means that the equals() method in Bicycle is not abstract

Partially Implemented Methods

- This compares two Bicycle objects based on price and model number
  - Note, this method overrides the equals() method from Object because it has the same parameters and return type

```
public boolean equals(Object obj) {
   if(this.price == obj.price &&
      this.modelNum == obj.modelNum) {
      return true;
   }
   else {
      return false;
   }//end if
}//end method equals
```

Partially Implemented Methods

- We can override the method in our subclass MountainBike and check for equivalence on other attributes

```
public boolean equals(Object obj) {
    if(super.equals(obj)) {
       if(this.suspension == obj.suspension)
       return true;
    }//end if
    return false;
}//end method equals
```

Subclassing Abstract Classes

- When inheriting from an abstract class, you must do either of the following:
  - Declare the child class as abstract
  - Override all abstract methods inherited from the parent class
  - Failure to do so will result in a compile-time error

Using Final

- Although it is nice to have the option, in some cases, you may not want some methods to be overridden or to have your class extended
- Java provides a tool to prevent programmers from overriding methods or creating subclasses:
  - the keyword final

Using Final

- A good example is the String class
- It is declared:

```
public final class String {}
```

- Programmers will refer to classes like this as immutable, meaning that no one can extend String and modify or override its methods

Using Final

- The final modifier can be applied to variables
- Final variables may not change their values after they are initialized

Using Final

- Final variables can be:
  - Class fields
    - Final fields with compile-time constant expressions are constant variables
    - Static can be combined with final to create an always-available, never-changing variable
  - Method parameters
  - Local variables

Using Final

- Final references must always reference the same object
- The object to which the variable is referencing cannot be changed
- The contents of that object may be modified
- Visit Oracle's Java tutorial for more information on using final:
  - http://docs.oracle.com/javase/tutorial/java/IandI/final.html

Triangle Applet Code

- The following code shows the steps involved in writing an applet with two triangles of different colors

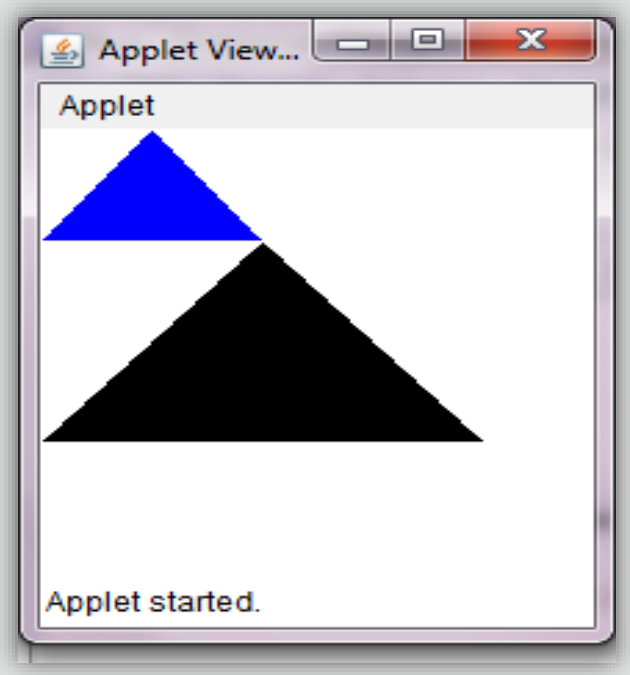
```
public class TrianglesApplet extends Applet{
   public void paint(Graphics g){
      int[] xPoints = {0, 40, 80};
      int[] yPoints = {50, 0, 50};
      g.setColor(Color.blue);
      g.fillPolygon(xPoints, yPoints, 3);
      int[] x2Points = {80, 160, 0};
      int[] y2Points = {50, 140, 140};
      g.setColor(Color.black);
      g.fillPolygon(x2Points, y2Points, 3);
   }//end method paint
}//end class TrianglesApplet
```

Triangle Applet Code Explained

- Step 1:
  - Extend Applet class to inherit all methods including paint
- Step 2:
  - Override the paint method to include the triangles
- Step 3:
  - Draw the triangle using the inherited fillPolygon method
- Step 4:
  - Draw the 2nd triangle using the inherited fillPolygon method
- Step 5:
  - Run and compile your code

Triangle Applet Image

- The Triangle Applet code displays the following image:



Polymorphism Excercise

- Access the **Cowboy Project** from Section 10 of the Learning Path for this course

Terminology

- Key terms used in this lesson included:
  - abstract
  - Dynamic Method Dispatch
  - final
  - Immutable
  - Overloading methods
  - Overriding methods
  - Polymorphism

Summary

- In this lesson, you should have learned how to:
  - Apply superclass references to subclass objects
  - Write code to override methods
  - Use dynamic method dispatch to support polymorphism
  - Create abstract methods and classes
  - Recognize a correct method override
  - Use the final modifier
  - Explain the purpose and importance of the Object class
  - Write code for an applet that displays two triangles of different colors
  - Describe object references


ORACLE
Academy