Java OOP: Inheritance, Part 2*

Richard Baldwin

Abstract

Baldwin shows you how to use method overriding to cause the behavior of a method inherited into a subclass to be appropriate for an object instantiated from the subclass.

1 Table of Contents

2 Preface

This module is one of a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

2.1.1 Figures

- Figure 1 (p. 6) . Program output.

*Version 1.1: Jul 27, 2012 3:54 pm -0500
†http://creativecommons.org/licenses/by/3.0/

2.1.2 Listings

3 Preview

This module builds on the previous module. It is recommended that you study that module before embarking on this module.

The program discussed in this module extends a **Radio** class to produce a new class that simulates an upgraded car radio containing a tape player.

Method overriding is used to modify the behavior of a method of the **Radio** class named **playStation** , to cause that method to behave appropriately when a tape has been inserted into the tape player.

4 Discussion and sample code

Inheriting methods and variables

When you define a class that extends another class, an object instantiated from your new class will contain all of the methods and all of the variables defined in your new class. The object will also contain all of the methods and all of the variables defined in all of the superclasses of your new class.

The behavior of the methods

The behavior of the methods defined in a superclass and inherited into your new class may, or may not, be appropriate for an object instantiated from your new class. If those methods are appropriate, you can simply leave them alone.

Overriding to change behavior

If the behavior of one or more methods defined in a superclass and inherited into your new class is not appropriate for an object of your new class, you can change that behavior by overriding the method in your new class.

How do you override a method?

To override a method in your new class, simply reproduce the name, argument list, and return type of the original method in a new method definition in your new class. Then provide a body for the new method. Write code in that body to cause the behavior of the overridden method to be appropriate for an object of your new class.

Here is a more precise description of method overriding taken from the excellent book entitled *The Complete Java 2 Certification Study Guide* , by Roberts, Heller, and Ernest:

"A valid override has identical argument types and order, identical return type, and is not less accessible than the original method. The overriding method must not throw any checked exceptions that were not declared for the original method."

Any method that is not declared **final** can be overridden in a subclass.

Overriding versus overloading

Don't confuse method overriding with method overloading. Here is what Roberts, Heller, and Ernest have to say about overloading methods:

"A valid overload differs in the number or type of its arguments. Differences in argument names are not significant. A different return type is permitted, but is not sufficient by itself to distinguish an overloading method."

Car radios with built-in tape players

This module presents a sample program that duplicates the functionality of the program named **Radio02** discussed in the previous module. A class named **Radio** is used to define the specifics of objects intended to simulate car radios.

A class named **Combo** extends the **Radio** class to define the specifics of objects intended to simulate improved car radios having built-in tape players.

Modification of the superclass

In the program named **Radio02** in the previous module, it was necessary to modify the superclass before extending it to provide the desired functionality. *(The requirement to modify the superclass before extending it seriously detracts from the benefits of inheritance.)*

No superclass modification in this module

The sample program *(named **Radio03**)* in this module uses method overriding to provide the same functionality as the previous program named **Radio02** , without any requirement to modify the superclass before extending it. *(Thus this program is more representative of the benefits available through inheritance than was the program in the previous module.)*

Overridden playStation method

In particular, a method named **playStation** , defined in the superclass named **Radio** , is overridden in the subclass named **Combo** .

The original version of **playStation** in the superclass supports only radio operations. The overridden version of **playStation** defined in the subclass supports both radio operations and tape operations.

*(The behavior of the version of **playStation** defined in the **Radio** class is not appropriate for an object of the **Combo** class. Therefore, the method was overridden in the **Combo** class to cause its behavior to be appropriate for objects instantiated from the **Combo** class.)*

A complete listing of the program is shown in Listing 5 near the end of this module.

The class named Radio

As usual, I will discuss the program in fragments.

Listing 1 (p. 3) shows the superclass named **Radio** . This code is shown here for easy referral. It is identical to the code for the same class used in the program named **Radio01** discussed in an earlier module.

Listing 1: The class named Radio.

```
  class Radio{
protected double[] stationNumber =
                       new double[5];

public void setStationNumber(
              int index,double freq){
  stationNumber[index] = freq;
}//end method setStationNumber

public void playStation(int index){
  System.out.println(
          "Playing the station at "
             + stationNumber[index]
             + " Mhz");
}//end method playStation
```

Will override playStation

The class named **Combo** *(discussed below)* will extend the class named **Radio** . The method named **playStation** , shown in Listing 1 (p. 3) , will be overridden in the class named **Combo** .

If you examine the code for the **playStation** method in Listing 1 (p. 3) , you will see that it assumes radio operations only and doesn't support tape operations. That is the reason that it needs to be overridden. *(For example, it doesn't know that it should refuse to play a radio station when a tape is being played.)*

The Combo class

Listing 2 (p. 4) shows the beginning of the class definition for the class named **Combo** . The **Combo** class extends the class named **Radio** .

Listing 2: Beginning of the Combo class.

```
class Combo extends Radio{

private boolean tapeIn = false;
```

The tapeIn variable

The most important thing about the code in Listing 2 (p. 4) is the declaration of the instance variable named **tapeIn** .

(In the program named Radio02 in the previous module, this variable was declared in the class named ***Radio*** *and inherited into the class named* ***Combo*** *. That was one of the undesirable changes required for the class named* ***Radio*** *in that module.)*

In this version of the program, the variable named **tapeIn** is declared in the subclass instead of in the superclass. Thus, it is not necessary to modify the superclass before extending it.

The constructor

The constructor in Listing 2 (p. 4) is the same as in the previous program named **Radio02** , so I won't discuss it further.

The overridden playStation method

The overridden version of the method named **playStation** is shown in Listing 3 (p. 4) . As you can see, this version of the method duplicates the signature of the **playStation** method in the superclass named **Radio** , but provides a different body.

Listing 3: The overridden playStation method.

```
  public void playStation(int index){
System.out.println("Play Radio");
if(!tapeIn){//tapeIn is false
  System.out.println(
      "  Playing the station at "
        + stationNumber[index]
        + " Mhz");
}else{//tapeIn is true
  System.out.println(
        "  Remove the tape first");
}//end if/else
}//end method playStation
```

Aware of the tape system

This overridden version of the **playStation** method in Listing 3 (p. 4) is aware of the existence of the tape system and behaves accordingly.

Depending on the value of the variable named **tapeIn** , this method will either

- tune and play a radio station, or
- display a message instructing the user to remove the tape.

Which version of playStation is executed?

When the **playStation** method is called on an object of the **Combo** class, the overridden version of the method *(and not the original version defined in the superclass named **Radio**)* is the version that is actually executed.

Although not particularly obvious in this example, this is one of the important characteristics of *runtime polymorphism* . When a method is called on a reference to an object, it is the type of the object *(and not the type of the variable containing the reference to the object)* that is used to determine which version of the method is actually executed.

Three other instance methods

The subclass named **Combo** defines three other instance methods:

- insertTape
- removeTape
- playTape

The code in these three methods is identical to the code in the methods having the same names in the program named **Radio02** in the previous module. I discussed that code in the previous module and won't repeat that discussion here. You can view those methods in the complete listing of the program shown in Listing 5 (p. 7) near the end of this module.

The driver class

Listing 4 (p. 5) shows the code for the driver class named Radio03.

Listing 4: The driver class.

```
 public class Radio03{
//This class simulates the
// manufacturer and the human user
public static void main(
                      String[] args){
```

The code in Listing 4 (p. 5) is also identical to the code in the program named **Radio02** discussed in the previous module. Therefore, I won't discuss it in detail here.

A new object of the Combo class

I present this code here solely to emphasize that this code instantiates a new object of the **Combo** class. This assures that the overridden version of the method named **playStation** will be executed by the statements in Listing 4 (p. 5) that call the **playStation** method.

(Although it is not the case in Listing 4 (p. 5) *, even if the reference to the object of type **Combo** had been stored in a reference variable of type **Radio** , instead of a reference variable of type **Combo** , calling the **playStation** method on that reference would have caused the overridden version of the method to have been executed. That is the essence of runtime polymorphism based on overridden methods in Java.)*

Program output

This program produces the output shown in Figure 1 (p. 6) on the computer screen.

Program output.

```
    Combo object constructed
Play Radio
  Playing the station at 93.5 Mhz
Insert Tape
  Tape is in
  Radio is off
Play Radio
  Remove the tape first
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
Play Tape
  Insert the tape first
Insert Tape
  Tape is in
  Radio is off
Play Tape
  Tape is playing
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
```

Figure 1: Program output.

I will leave it as an exercise for the student to compare this output with the messages sent to the object by the code in Listing 4 (p. 5) .

5 Summary

An object instantiated from a class that extends another class will contain all of the methods and all of the variables defined in the subclass, plus all of the methods and all of the variables inherited into the subclass.

The behavior of methods inherited into the subclass may not be appropriate for an object instantiated from the subclass. You can change that behavior by overriding the method in the definition of the subclass.

To override a method in the subclass, reproduce the name, argument list, and return type of the original method in a new method definition in the subclass. Make sure that the overridden method is not less accessible than the original method. Also, make sure that it doesn't throw any checked exceptions that were

not declared for the original method.

Provide a body for the overridden method, causing the behavior of the overridden method to be appropriate for an object of the subclass. Any method that is not declared **final** can be overridden in a subclass. The program discussed in this module extends a **Radio** class to produce a subclass that simulates an upgraded car radio containing a tape player.

Method overriding is used to modify the behavior of an inherited method named **playStation** to cause that method to behave appropriately when a tape has been inserted into the radio.

Method *overriding* is different from method overloading. Method *overloading* will be discussed in the next module.

6 What's next?

In the next module, I will explain the use of overloaded methods for the purpose of achieving compile-time polymorphism.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: OOP 101, Inheritance, Part 2
- File: Java1606.htm
- Published: January 28, 2002
- Revised: June 8, 2012
- Keywords:
  - method overriding
  - method overloading

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

8 Complete program listing

A complete listing of the program is shown in Listing 5 (p. 7) below.

Listing 5: The program named Radio03.

```
    Copyright 2002, R.G.Baldwin
Simulates the manufacture and use of a
combination car radio and tape player.
Uses method overriding to avoid
modifying the class named Radio.

This program produces the following
output on the computer screen:

Combo object constructed
Play Radio
  Playing the station at 93.5 Mhz
Insert Tape
  Tape is in
  Radio is off
Play Radio
  Remove the tape first
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
Play Tape
  Insert the tape first
Insert Tape
  Tape is in
  Radio is off
Play Tape
  Tape is playing
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
************************************/

public class Radio03{
  //This class simulates the
  // manufacturer and the human user
  public static void main(
                        String[] args){
    Combo myObjRef = new Combo();
    myObjRef.setStationNumber(3,93.5);
    myObjRef.playStation(3);
    myObjRef.insertTape();
    myObjRef.playStation(3);
    myObjRef.removeTape();
    myObjRef.playStation(3);
    myObjRef.playTape();
    myObjRef.insertTape();
    myObjRef.playTape();
```

```
    myObjRef.removeTape();
    myObjRef.playStation(3);
  }//end main
}//end class Radio03
//===================================//

class Radio{
  //This class simulates the plans from
  // which the radio object is created.
  // This code is the same as in the
  // program named Radio01.
  protected double[] stationNumber =
                        new double[5];

  public void setStationNumber(
               int index,double freq){
    stationNumber[index] = freq;
  }//end method setStationNumber

  //This version of playStation doesn't
  // accommodate tape operations.
  public void playStation(int index){
    System.out.println(
            "Playing the station at "
               + stationNumber[index]
               + " Mhz");
  }//end method playStation

}//end class Radio
//===================================//

class Combo extends Radio{

  private boolean tapeIn = false;
  //---------------------------------//

  public Combo(){//constructor
    System.out.println(
           "Combo object constructed");
  }//end constructor
  //---------------------------------//

  //Overridden playStation method. This
  // overridden version accommodates
  // tape operations.
  public void playStation(int index){
    System.out.println("Play Radio");
    if(!tapeIn){
      System.out.println(
          "  Playing the station at "
               + stationNumber[index]
```

```
              + " Mhz");
    }else{
      System.out.println(
            "  Remove the tape first");
    }//end if/else
  }//end method playStation
  //---------------------------------//

  public void insertTape(){
    System.out.println("Insert Tape");
    tapeIn = true;
    System.out.println(
                       "  Tape is in");
    System.out.println(
                     "  Radio is off");
  }//end insertTape method
  //---------------------------------//

  public void removeTape(){
    System.out.println("Remove Tape");
    tapeIn = false;
    System.out.println(
                      "  Tape is out");
    System.out.println(
                      "  Radio is on");
  }//end removeTape method
  //---------------------------------//

  public void playTape(){
    System.out.println("Play Tape");
    if(!tapeIn){
      System.out.println(
            "  Insert the tape first");
    }else{
      System.out.println(
                  "  Tape is playing");
    }//end if/else
  }//end playTape
}//end class combo
```

-end-