Java OOP: Inheritance, Part 1*

R.G. (Dick) Baldwin

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0†

Abstract

Baldwin shows you how to extend an existing class to create a new class. The new class is the blueprint for a new type. This is the mechanism for class inheritance in Java. Inheritance provides a formal mechanism for code reuse.

1 Table of Contents

- Preface (p. 1)
  - Viewing tip (p. 1)
    - Figures (p. 1)
    - Listings (p. 2)
- Preview (p. 2)
- Discussion and sample code (p. 2)
- Summary (p. 8)
- What's next? (p. 8)
- Miscellaneous (p. 8)
- Complete program listing (p. 9)

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. 7) . Program output.

*Version 1.1: Jul 27, 2012 4:05 pm -0500
†http://creativecommons.org/licenses/by/3.0/

2.1.2 Listings

- Listing 1 (p. 3) . Beginning of the Combo class.
- Listing 2 (p. 4) . The insertTape method.
- Listing 3 (p. 5) . The removeTape method.
- Listing 4 (p. 5) . The playTape method.
- Listing 5 (p. 5) . Modified Radio class.
- Listing 6 (p. 6) . Tape status.
- Listing 7 (p. 6) . Change to the playStation method.
- Listing 8 (p. 6) . The class named Radio02.
- Listing 9 (p. 9) . The program named Radio02.

3 Preview

Extending a class

This module shows you how to extend an existing class to create a new class. The new class is the blueprint for a new type.

Inheritance and code reuse

The existing class is often called the *superclass* and the new class is often called the *subclass* . This is the mechanism for class inheritance in Java. Inheritance provides a formal mechanism for code reuse.

The subclass inherits all of the variables and all of the methods defined in the superclass.

Car radios with tape players

A class from a previous module *(whose objects represent car radios)* is extended to define a new class, whose objects represent expanded car radios that contain tape players. *(Yes, at one point in history, car radios did contain tape players instead of CDs.)*

Sending messages to the object

Objects of the new class know how to respond to messages for inserting, playing, and removing a tape, in addition to those messages that are appropriate for objects of the original Radio class.

4 Discussion and sample code

The three pillars of OOP

In an earlier module, I explained that most books on OOP will tell you that in order to understand OOP, you must understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

I agree with that assessment.

Encapsulation

The first module in this series provided an explanation of encapsulation.

Inheritance

This module *(and some modules to follow)* will provide an explanation of inheritance. I will use another simple program to explain the concept of inheritance.

Polymorphism

Polymorphism is the most complex of the three, and will be explained in future modules.

A new data type

Whenever you define a class in Java, you cause a new data type to become available to the program. Therefore, whenever you need a new data type, you can define a new class to make that type available.

Extending a class

Defining a new class *(to create a new type)* can involve a lot of effort. Sometimes you have an option that can greatly reduce the effort required to create your new type. If a class *(type)* already exists that is close to what you need, you can often extend that class to produce a new class that is closer to what you need.

In many cases, this will require much less effort than that required to start from scratch and define a new class to establish a new type. The ability to extend one class into another new class is the *essence of inheritance* .

According to the current jargon, the new class is called the *subclass* and the class that is extended is called the *superclass* .

What is inherited?

The subclass inherits all of the variables and all of the methods defined in *(or inherited into)* the superclass, almost as if you had completely defined the new class from scratch, and had reproduced all of the code already defined in the existing superclasses.

Code reuse

Therefore, inheritance often makes it possible to define a new class with a minimum requirement to write new code by formally reusing the code that was previously written into the superclasses. Sometimes you can get by with simply extending the existing class.

Sometimes, however, it is also necessary to make changes to the existing class to improve its ability to be extended in a meaningful way. *(That is the case with the sample program discussed in this module, but the next module will show you how to avoid that issue.)* It all depends on how the existing class was designed in the first place.

The Radio class

A previous program defined a class named **Radio** . Objects instantiated from the **Radio** class *(see the previous modules for a discussion of instantiating objects)* were intended to simulate car radios. *(Note that the car radios simulated by objects of the **Radio** class didn't have built-in tape players.)*

The Combo class

In this module, I will use inheritance to extend the **Radio** class into a new class named **Combo** . Objects instantiated from the **Combo** class are intended to simulate car radios with a built-in tape player.

A complete listing of the new program is shown in Listing 9 (p. 9) near the end of the module.

Will discuss in fragments

As usual, I will discuss this program in fragments. I will begin my discussion with the definition of the new class named **Combo** . Then I will come back and discuss the class named **Radio** and the driver class named **Radio02** .

The combo class

The code in Listing 1 (p. 3) shows the beginning of the class named **Combo** .

Listing 1: Beginning of the Combo class.

```
    class Combo extends Radio{

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

Two new items

There are two new items in Listing 1 (p. 3) that you did not see in the code in the previous modules.

Combo extends Radio

First, the class named **Combo** extends the class named **Radio** . This means that an object instantiated from the **Combo** class will contain all of the variables and all the methods defined in the **Combo** class,

plus all the variables and methods defined in the **Radio** class, and its superclasses. *(The variables and methods of the superclass are inherited into the subclass.)*

An explicit constructor

Second, the class named **Combo** defines an explicit constructor.

Defining a constructor is optional

When defining a new class, it is not necessary to define a constructor. If you don't define a constructor, a default constructor will be provided automatically.

Why define a constructor?

The intended purpose of a constructor is to initialize the instance variables belonging to the new object. However, constructors can do other things as well. In this case, I used an explicit constructor to display a message when the object is instantiated from the class named **Combo** .

Brief discussion of constructors

I'm not going to discuss constructors in detail at this point. However, I will give you a few rules regarding constructors.

- Constructors *(like methods)* can be overloaded. *(I will explain what overloading means in a subsequent module.)*
- The names of constructors must match the names of the classes in which they are defined.
- A constructor signature never indicates a return type *(such as void or double)* .
- The code in a constructor never contains a return statement.

Instance methods

The new class named **Combo** defines three instance methods, each of which has to do with the handling of tape in the tape player:

- insertTape
- removeTape
- playTape

(If you feel ambitious, you could upgrade this class even further to add features such as rewind, fast forward, pause, etc.).

The insertTape method

The entire method named **insertTape** is shown in Listing 2 (p. 4) . This is the method that is used to simulate the insertion of a tape by the user.

Listing 2: The insertTape method.

```
  public void insertTape(){
System.out.println("Insert Tape");
tapeIn = true
```

The most significant thing about the code in Listing 2 (p. 4) is the assignment of the **true** value to the **boolean** variable named **tapeIn** . Other than setting the value of the **tapeIn** variable to **true** , the code in Listing 2 (p. 4) simply prints some messages to indicate what is going on.

What is tapeIn used for?

As you will see shortly, the value of the variable named **tapeIn** is used to determine if it is possible to play the tape or to play the radio.

According to that logic:

- If **tapeIn** is true, it is possible to play the tape but it is not possible to play the radio.
- If **tapeIn** is false, it is possible to play the radio, but it is not possible to play the tape.

tapeIn is not declared in the Combo class

It is also worthy of note that in this version of the program, the variable named **tapeIn** is not declared in the **Combo** class *(this will change in the next module where the program uses method overriding)* . Rather, this variable is inherited from the **Radio** class that is extended by the **Combo** class.

The removeTape method

The **removeTape** method of the **Combo** class is shown in Listing 3 (p. 5) . Its behavior is pretty much the reverse of the **insertTape** method, so I won't discuss it further.

Listing 3: The removeTape 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\
```

The playTape method

Listing 4 (p. 5) shows the method named **playTape** defined in the new **Combo** class.

Listing 4: The playTape method .

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

Confirm that the tape is ready

Calling the method named **playTape** can be thought of as sending a message to the **Combo** object asking it to play the tape. The code in the **playTape** method checks to confirm that the value stored in the **tapeIn** variable is **true** before executing the request to play the tape.

If **tapeIn** is **false** , an error message is displayed advising the user to insert the tape first.

If **tapeIn** is **true** , the method prints a message indicating that the tape is playing.

Modified Radio class

Listing 5 (p. 5) shows the definition of the modified version of the class named **Radio** .

Listing 5: Modified Radio class.

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

Tape status

The first significant change that was made to the class named **Radio** is shown in Listing 6 (p. 6) below.

Listing 6: Tape status.

```
protected boolean tapeIn = false;
```

The statement in Listing 6 (p. 6) declares and initializes a new instance variable named **tapeIn** . As explained earlier, this instance variable is used to indicate whether or not a tape is inserted. *(The Combo class inherits this variable.)*

Earlier in this module, I explained how the **playTape** method of the **Combo** class uses this value to determine whether or not to attempt to play a tape.

Change to the playStation method

The significant change that was made to the method named **playStation** of the **Radio** class is shown in Listing 7 (p. 6) below.

Listing 7: Change to the playStation method.

```
    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")
```

Check the tape status

The code in Listing 7 (p. 6) uses **tapeIn** to check the tape status before attempting to tune the radio station and play the radio. If a tape is inserted, this method simply displays an error message instructing the user to remove the tape first.

So, what's the big deal with inheritance?

The fact that it was necessary for me to make changes to the class named **Radio** greatly reduced the benefit of inheritance in this case. However, even in this case, the use of inheritance eliminated the need for me to define a new class that reproduces all of the code in the class named **Radio** .

*(In the next module, I will explain the process of overriding methods. I will show you how to use method overriding to accomplish these same purposes by extending the **Radio** class, without any requirement to modify the code in the **Radio** class. That will be a much better illustration of the benefits of inheritance.)*

The driver class

The new driver class named **Radio02** is shown in Listing 8 (p. 6) .

Listing 8: The class named Radio02.

```
 public class Radio02{
//This class simulates the
// manufacturer and the human user
public static void main(
                      String[] args){
 Combo myObjRef = new Combo()
```

New object of the Combo class

The most significant change in this class *(relative to the driver class named* ***Radio01*** *in a previous module)* is the statement that instantiates a new object of the **Combo** class *(instead of the Radio class)* .

All of the other new code in Listing 8 (p. 6) is used to send messages to the new object in order to exercise its behavior.

Program output

The **Combo** object responds to those messages by producing the screen output shown in Figure 1 (p. 7) .

Program output.

```
    Combo object constructed
Button 3 programmed
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.

An exercise for the student

As the old saying goes, I will leave it as an exercise for the student to correlate the messages in Listing 8 (p. 6) with the output shown in Figure 1 (p. 7) .

5 Summary

Extending an existing class often provides an easy way to create a new type. This is primarily true when an existing class creates a type whose features are close to, but not identical to the features needed in the new type.

When an existing class is extended to define a new class, the existing class is often called the superclass and the new class is often called the subclass.

The subclass inherits all of the variables and all of the methods defined in the superclass and its superclasses.

Inheritance provides a formal mechanism for code reuse.

This module modifies slightly, and then extends the **Radio** class from a previous module to define a new class named **Combo** . Objects of the **Combo** class simulate car radios that contain tape players. Objects of the **Combo** class know how to respond to messages for inserting, playing, and removing a tape, in addition to those messages appropriate for an object of the **Radio** class.

The changes that were required in the definition of the **Radio** class provide for the fact that it is not possible to play a radio station and to play a tape at the same time. This change was necessary because the original designer of the **Radio** class *(this author)* didn't design that class with the idea of extending it to include a tape player. This points out the importance of thinking ahead when defining a new class.

6 What's next?

In the next module, I will show 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.

I will also show you how to use method overriding to eliminate the above requirement to modify the **Radio** class before extending it.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Essence, Inheritance, Part 1
- File: Java1604.htm
- Published: January 14, 2002
- Revised: June 7, 2012
- Keywords:
  - inheritance
  - extends
  - superclass
  - subclass
  - type

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 9 (p. 9) below.

The primary difference between this program and the program in the earlier module *(whose objects simulate car radios)* is the inclusion in this program of a new class named **Combo** . The class named **Combo** extends the original **Radio** class to create a new type of radio that also contains a tape player.

Listing 9: The program named Radio02.

```
    /*File Radio02.java
Copyright 2002, R.G.Baldwin
Simulates the manufacture and use of a
combination car radio and tape player.

This program produces the following
output on the computer screen:

Combo object constructed
Button 3 programmed
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 Radio02{
  //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 Radio02
//===================================//

class Radio{
  //This class simulates the plans from
  // which the radio object is created.
  protected double[] stationNumber =
                        new double[5];
  protected boolean tapeIn = false;
  //---------------------------------//

  public void setStationNumber(
                int index,double freq){
    stationNumber[index] = freq;
    System.out.println("Button "
              + index + " programmed");
  }//end method setStationNumber
  //---------------------------------//

  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

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

class Combo extends Radio{

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

  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-