

JAVA OOP: INDIRECTION, ARRAY OBJECTS, AND CASTING*

R.G. (Dick) Baldwin

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

Abstract

Learn about indirection, array objects, and casting.

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. 3)
- Run the program (p. 6)
- Summary (p. 7)
- What's next? (p. 7)
- Online video links (p. 7)
- Miscellaneous (p. 7)
- Complete program listings (p. 8)

2 Preface

This module is one of a series of modules designed to teach you about 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. 2) . Program output on command line screen.

*Version 1.3: Nov 14, 2012 9:38 am +0000

[†]<http://creativecommons.org/licenses/by/3.0/>

2.1.2 Listings

- Listing 1 (p. 3) . Beginning of the Prob05 class.
- Listing 2 (p. 4) . The class named Prob05MyClassA.
- Listing 3 (p. 5) . The next statement in the main method.
- Listing 4 (p. 5) . The class named Prob05MyClassB.
- Listing 5 (p. 6) . The end of the main method.
- Listing 6 (p. 8) . Complete program listing.

3 Preview

The program that I will explain in this module produces no graphics and does not require the use of Ericson's media library.

OOP concepts

The program illustrates the following OOP concepts among others:

- Multiple levels of indirection
- A one-element array of type **Object**
- Storing a reference to an object in an array element as type **Object**
- An anonymous object
- Passing a reference to a subclass object as type **Object**
- Downcasting an incoming object reference to access a method

Program specifications

Write a program named **Prob05** that uses the class definition shown in Listing 1 (p. 3) to produce an output similar to that shown in Figure 1 (p. 2) on the command-line screen.

Program output on command line screen.

```
    Prob05
Dick
Baldwin
-28
-28
```

Figure 1: Program output on command line screen.

A random value

Because the program generates and uses a random data value, the actual values displayed will differ from one run to the next. However, in all cases, the two values shown in Figure 1 (p. 2) must match.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob05** which begins in Listing 1 (p. 3) .

4 Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing is provided in Listing 6 (p. 8) near the end of the module.

I will begin with the driver class named **Prob05** , which begins in Listing 1 (p. 3) .

Listing 1: Beginning of the Prob05 class.

```
import java.util.*;

class Prob05{
    public static void main(String[] args){

        Random generator = new Random(new Date().getTime());
        int randomNumber = (byte)generator.nextInt();

        Object[] objRef = {new Prob05MyClassA(randomNumber)};
    }
}
```

Everything in Listing 1 (p. 3) should be familiar to you except for the last statement, which I will explain shortly.

Characteristics of arrays in Java

Before explaining that statement, however, I will discuss some of the important characteristics of array objects in Java. A list of such characteristics follows in no particular order:

- All arrays in Java are one-dimensional arrays. (*Multidimensional arrays are created by creating tree structures of one-dimensional arrays.*)
- Each array in Java is encapsulated in a special type of object that I will refer to as an *array object* .
- As with all objects, an array object must be accessed using a reference to the array object.
- When the declared type of an array is one of the eight primitive types, the actual values are stored in the array elements in the array object.
- When the declared type of an array is the type of an object (*array object or ordinary object*), references to the objects are stored in the array elements and the objects actually exist elsewhere in memory.
- As with instance variables, the elements in an array are typically initialized with the standard default values for the types involved (*zero, false, or null*) . That is not the case in this program however.
- The array that is encapsulated in an array object may have none, one, or more elements. (*Yes, it is possible for a Java array to have no elements, but that normally occurs only in special circumstances.*)
- The **length** or size of the array is established when the array object is instantiated and cannot be changed thereafter.
- Every array object contains a special property named **length** that contains the number of elements in an array. It is always possible to determine the number of elements in an array object at runtime by accessing the value of the **length** property for the array object.

A special instantiation syntax

There is a special syntax that allows for the instantiation of an array object and the initialization of the array elements in a single statement. (*I explain this in detail in my online OOP tutorial modules.*) The last statement in Listing 1 (p. 3) is an example of this syntax.

Briefly, the syntax consists of a comma separated list of element values (*expressions*) inside a pair of matching curly braces. The **length** of the array is determined by the number of values in the list. The type of the array is determined by the types of the elements in the list.

This syntax instantiates an array object of the correct **length** and populates the elements with the specified values.

A reference is returned

A reference to the array object is returned in much the same way that a constructor for an ordinary object returns a reference to the object.

As is always the case, if the reference is stored in a variable, the type of the reference must be *assignment compatible* with the type of the variable.

What is assignment compatible?

I recommend that you go to Google and search for the following keywords to learn more about this topic:

baldwin java "assignment compatible"

A one-element array

The last statement in Listing 1 (p. 3) instantiates an array object containing a one-element array. The array element is initialized with a reference to a new object of type **Prob05MyClassA**, which exists somewhere else in memory.

The value of a random number that was generated earlier in the **main** method is passed as a parameter to the constructor for the object of type **Prob05MyClassA**.

Save the reference to the array object

The reference to the array object is stored in the local reference variable named **objRef** of type **Object**. We know that the reference is *assignment compatible* with this reference variable because the **Object** type is completely generic. All non-primitive types are assignment compatible with type **Object**.

The class named Prob05MyClassA

At this point, I am going to put the explanation of the class named **Prob05** temporarily on hold and explain the class named **Prob05MyClassA**, which is shown in its entirety in Listing 2 (p. 4).

Listing 2: The class named Prob05MyClassA.

```
class Prob05MyClassA extends Prob05{
private int data;

public Prob05MyClassA(int inData){
    System.out.println("Prob05");
    System.out.println("Dick");
    data = inData;
} //end constructor

public int getData(){
    return data;
} //end getData()

} //end class Prob05MyClassA
```

Note that the class named **Prob05MyClassA** extends the class named **Prob05**, which is partially shown in Listing 1 (p. 3).

Familiar code

All of the code in Listing 2 (p. 4) should be familiar to you because it is very similar to the code in the previous module. Therefore, no explanation of Listing 2 (p. 4) is warranted.

Save the incoming value

In summary, when the object of type **Prob05MyClassA** is instantiated, it saves the value of an incoming constructor parameter in a private instance variable.

Return the saved value

When the method named **getData** is called on a reference to the object, it returns a copy of that value.

A review

To review what I have already said, the array object that was instantiated in Listing 1 (p. 3) contains a reference to this object of type **Prob05MyClassA** in the only element of the one-element array.

The reference to the array object is stored in the reference variable named **objRef** .

Indirection at work

At this point, **objRef** contains a reference to an array object, one element of which contains a reference to an ordinary object, which is located somewhere else in memory. This is indirection.

The next statement in the main method

Returning now to the **main** method that began in Listing 1 (p. 3) , Listing 3 (p. 5) shows the next statement in the **main** method following the last statement in Listing 1 (p. 3) .

Listing 3: The next statement in the main method.

```
System.out.println(
    new Prob05MyClassB().getDataFromObj(objRef[0]));
```

What is an anonymous object?

An anonymous object is an object whose reference is not saved in a named reference variable.

Instantiate an anonymous object

Consider the parameter list of the **println** method shown in Listing 3 (p. 5) . A new object of the **Prob05MyClassB** class is instantiated in the parameter list. However, the reference to that object is not saved in a named reference variable. Instead, that reference is used to immediately call the method named **getDataFromObj** that belongs to the anonymous object.

The parameter that is passed...

Now consider the parameter that is passed to the method named **getDataFromObj** . The expression inside that parameter list extracts the contents of the zeroth element in the array object that is referred to by the contents of the variable named **objRef** .

And those contents are...

That element contains a reference to an object of the class **Prob05MyClassA** (see Listing 1 (p. 3))

Therefore, a reference to an object of type **Prob05MyClassA** is passed as a parameter to the method named **getDataFromObj** .

The class named Prob05MyClassB

It is time to take a look at the class in which the **getDataFromObj** method is defined.

The class named **Prob05MyClassB** is shown in its entirety in Listing 4 (p. 5) .

Listing 4: The class named Prob05MyClassB.

```
class Prob05MyClassB{

    Prob05MyClassB(){
        System.out.println("Baldwin");
    }//end constructor

    public int getDataFromObj(Object refToObj){
        return ((Prob05MyClassA)refToObj).getData();
    }//end getDataFromObj()

} //end class Prob05MyClassB
```

Extends the Object class

Note that this class does not extend the class named **Prob05** . In fact, it doesn't explicitly extend any class. This means that it extends the class named **Object** by default because every class is a subclass of the class named **Object** .

The constructor

The constructor for this class is inconsequential. It simply displays my last name when the object is instantiated, producing part of the output text shown in Figure 1 (p. 2) .

The getDataFromObj method

The interesting part of Listing 4 (p. 5) is the definition of the method named **getDataFromObj** .

As we saw before, this method receives a reference to an object of type **Prob05MyClassA** (see Listing 3 (p. 5)) . However, this reference is not received as the true type of the object. Instead, it is received as type **Object** , which is the ultimate superclass of the class named **Prob05MyClassA** .

The objective of the method

The objective is to call the method named **getData** on the incoming reference. However, the **Object** class doesn't know anything about a method named **getData** because the **Object** class neither defines nor inherits a method having that signature. Instead, the **getData** method is defined in the class named **Prob05MyClassA** , which is the true type of the object.

A cast is required

Therefore, it is necessary to convert the type of the reference back to its true type using a cast operator before that reference can be used to call the method named **getData** . (The cast operator is shown in Listing 4 (p. 5) .)

The returned value

The **getData** method returns a copy of the value that was passed as a constructor parameter when the object was instantiated. (See Listing 2 (p. 4) .) Recall that the value was the original random value. (See Listing 1 (p. 3) .)

Referring back to Listing 4 (p. 5) , that is the value that is returned from the call to the **getDataFromObj** method in Listing 3 (p. 5) , which cause the value to be displayed as the first numeric value in Figure 1 (p. 2) .

The end of the main method

Returning once more to the **main** method and picking up where we left off in Listing 3 (p. 5) , Listing 5 (p. 6) shows the final statement in the **main** method.

Listing 5: The end of the main method.

```

        System.out.println(randomNumber);

    } //end main
} //end class Prob05

```

This statement simply displays the original random value that was passed to the constructor for the **Prob05MyClassA** in Listing 1 (p. 3) . This statement displays the second numeric value shown as the last line of text in Figure 1 (p. 2) .

The end of the program

At this point, the **main** method terminates causing the program to terminate.

5 Run the program

I encourage you to copy the code from Listing 6 (p. 8) , compile it and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6 Summary

You learned about the following OOP concepts, among others in this module.

- Multiple levels of indirection
- A one-element array of type **Object**
- Storing a reference to an object in an array element as type **Object**
- An anonymous object
- Passing a reference to a subclass object as type **Object**
- Downcasting an incoming object reference to access a method

7 What's next?

You will learn how to use nested loops to process pixels on a row and column basis in the next module.

8 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 05 ¹
 - Part01 ²
 - Part02 ³
 - Part03 ⁴

9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Indirection, Array Objects, and Casting
- File: Java3010.htm
- Published: August 2, 2012
- Revised: November 14, 2012

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 copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

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

¹<http://www.youtube.com/playlist?list=PL13622F7BA83F110C>

²http://www.youtube.com/watch?v=Ow_XzlSrmsw

³http://www.youtube.com/watch?v=UiT_ZYtNqWo

⁴<http://www.youtube.com/watch?v=fCiMM4ps3o4>

10 Complete program listings

A complete listing of the program discussed in this module is shown in Listing 6 (p. 8) below.

Listing 6: Complete program listing.

```

/*File Prob05 Copyright 2001, R.G.Baldwin
Rev 12/16/08
*****/
import java.util.*;

class Prob05{
    public static void main(String[] args){

        Random generator = new Random(new Date().getTime());
        int randomNumber = (byte)generator.nextInt();

        Object[] objRef = {new Prob05MyClassA(randomNumber)};

        System.out.println(
            new Prob05MyClassB().getDataFromObj(objRef[0]));

        System.out.println(randomNumber);

    }//end main
}//end class Prob05
//=====//

class Prob05MyClassA extends Prob05{
    private int data;

    public Prob05MyClassA(int inData){
        System.out.println("Prob05");
        System.out.println("Dick");
        data = inData;
    }//end constructor

    public int getData(){
        return data;
    }//end getData()

}//end class Prob05MyClassA
//=====//

class Prob05MyClassB{

    Prob05MyClassB(){
        System.out.println("Baldwin");
    }//end constructor

    public int getDataFromObj(Object refToObj){

```

```
        return ((Prob05MyClassA)refToObj).getData();
    }//end getDataFromObj()

} //end class Prob05MyClassB

-end-
```