

JAVA1612: RUNTIME POLYMORPHISM THROUGH INHERITANCE*

R.G. (Dick) Baldwin

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

Abstract

With runtime polymorphism, the selection of a method for execution is based on the actual type of object whose reference is stored in a reference variable, and not on the type of the reference variable on which the method is called.

1 Table of Contents

- Preface (p. 1)
 - Viewing tip (p. 1)
 - * Listings (p. 2)
- Preview (p. 2)
- Discussion and sample code (p. 2)
- Summary (p. 7)
- What's next? (p. 7)
- Miscellaneous (p. 7)
- Complete program listing (p. 8)

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 listings while you are reading about them.

*Version 1.2: Dec 12, 2012 7:38 am -0600

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

2.1.1 Listings

- Listing 1 (p. 3) . Definition of the class named A.
- Listing 2 (p. 4) . Definition of the class named B.
- Listing 3 (p. 4) . Beginning of the driver class named Poly03.
- Listing 4 (p. 5) . Polymorphic behavior.
- Listing 5 (p. 5) . Source of a compiler error.
- Listing 6 (p. 6) . A new object of type A.
- Listing 7 (p. 8) . Complete program listing.

3 Preview

What is polymorphism?

The meaning of the word *polymorphism* is something like *one name, many forms* .

How does Java implement polymorphism?

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

In some cases, multiple methods have the same name, but different formal argument lists (*overloaded methods, which were discussed in a previous module*) .

In other cases, multiple methods have the same name, same return type, and same formal argument list (*overridden methods*) .

Three distinct forms of polymorphism

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through class inheritance
- Method overriding through the Java interface

I covered method overloading as one form of polymorphism (*compile-time polymorphism*) in a previous module. I also explained automatic type conversion and the use of the cast operator for type conversion in a previous module.

In this module ...

I will begin the discussion of runtime polymorphism through method overriding and class inheritance in this module. I will cover interfaces in a future module.

The essence of runtime polymorphic behavior

With runtime polymorphism based on method overriding,

- the decision as to which version of a method will be executed is based on
- the actual type of the object whose reference is stored in the reference variable, and
- **not** on the type of the reference variable on which the method is called.

Late binding

The decision as to which version of the method to call cannot be made at compile time. That decision must be deferred and made at runtime. This is sometimes referred to as *late binding* .

4 Discussion and sample code

Operational description of runtime polymorphism

Here is an operational description of runtime polymorphism as implemented in Java through class inheritance and method overriding:

- Assume that a class named **SuperClass** defines a method named **method** .

- Assume that a class named **SubClass** extends **SuperClass** and overrides the method named **method** .
- Assume that a reference to an object of the class named **SubClass** is assigned to a reference variable named **ref** of type **SuperClass** .
- Assume that the method named **method** is then called on the reference variable using the following syntax:


```
ref.method()
```
- **Result:** The version of the method named **method** that will actually be executed is the *overridden* version in the class named **SubClass** , and is not the version that is defined in the class named **SuperClass**, even though the reference to the object of type **SubClass** is stored in a variable of type **SuperClass** .

This is runtime polymorphism in a nutshell, which is sometimes also referred to as late-binding.

Runtime polymorphism is very powerful

As you gain more experience with Java, you will learn that much of the power of OOP using Java is centered on runtime polymorphism using class inheritance, interfaces, and method overriding. (*The use of interfaces for polymorphism will be discussed in a future module.*)

An important attribute of runtime polymorphism

The decision as to which version of the method to execute

- is based on the actual type of object whose reference is stored in the reference variable, and
- not on the type of the reference variable on which the method is called.

Why is it called runtime polymorphism?

The reason that this type of polymorphism is often referred to as runtime polymorphism is because the decision as to which version of the method to execute cannot be made until runtime. The decision cannot be made at compile time.

Why defer the decision?

The decision cannot be made at compile time because the compiler has no way of knowing (*when the program is compiled*) the actual type of the object whose reference will be stored in the reference variable .

In an extreme case, for example, the object might be de-serialized at runtime from a network connection of which the compiler has no knowledge.

Could be either type

For the situation described above, that de-serialized object could just as easily be of type **SuperClass** as of type **SubClass** . In either case, it would be valid to assign the object's reference to the same superclass reference variable.

If the object were of the **SuperClass** type, then a call to the method named **method** on the reference would cause the version of the method defined in **SuperClass** , and not the version defined in **SubClass** , to be executed. (*The version executed is determined by the type of the object and not by the type of the reference variable containing the reference to the object.*)

Sample Program

Let's take a look at a sample program that illustrates runtime polymorphism using class inheritance and overridden methods. The name of the program is **Poly03** . A complete listing of the program is shown in Listing 7 (p. 8) near the end of the module.

Listing 1 (p. 3) shows the definition of a class named **A** , which extends the class named **Object** . (*Remember that any class that doesn't extend some other class automatically extends **Object** by default, and it is not necessary to show that explicitly as I did in this example.*)

Listing 1: Definition of the class named A.

```

    class A extends Object{
    public void m(){
        System.out.println("m in class A");
    }//end method m()
} //end class A

```

The class named **A** defines a method named **m()** .

Behavior of the method

The behavior of the method, as defined in the class named **A** , is to display a message indicating that it has been called, and that it is defined in the class named **A** .

This message will allow us to determine which version of the method is executed in each case discussed later.

The class named B

Listing 2 (p. 4) shows the definition of a class named **B** that extends the class named **A** .

Listing 2: Definition of the class named B.

```

    class B extends A{
    public void m(){
        System.out.println("m in class B");
    }//end method m()
} //end class B

```

The class named **B** overrides (*redefines*) the method named **m()** , which it inherits from the class named **A** .

Behavior of the overridden version of the method

Like the inherited version, the overridden version displays a message indicating that it has been called. However, the message is different from the message displayed by the inherited version discussed above. The overridden version tells us that it is defined in the class named **B** . (*The behavior of the overridden version of the method is appropriate for an object instantiated from the class named B .*)

Again, this message will allow us to determine which version of the method is executed in each case discussed later.

The driver class

Listing 3 (p. 4) shows the beginning of the driver class named **Poly03** .

Listing 3: Beginning of the driver class named Poly03.

```

    public class Poly03{
    public static void main(String[] args){
        Object var = new B();
        ((B)var).m();
    }

```

A new object of the class B

The code in the **main** method begins by instantiating a new object of the class named **B** , and assigning the object's reference to a reference variable of type **Object** .

(*Recall that this is legal because an object's reference can be assigned to any reference variable whose type is a superclass of the class from which the object was instantiated. The class named **Object** is the superclass of all classes.*)

Downcast and call the method

If you read the earlier module on casting, it will come as no surprise to you that the second statement in the **main** method, which casts the reference down to type **B** and calls the method named **m()** on it, will compile and execute successfully.

Which version is executed?

The execution of the method produces the following output on the computer screen:

```
m in class B
```

By examining the output, you can confirm that the version of the method that was overridden in the class named **B** is the version that was executed.

Why was this version executed?

This should also come as no surprise to you. The cast converts the type of the reference from type **Object** to type **B** .

You can always call a public method belonging to an object using a reference to the object whose type is the same as the class from which the object was instantiated.

Not runtime polymorphic behavior

Just for the record, the above call to the method does not constitute runtime polymorphism (*in my opinion*) . I included that call to the method to serve as a backdrop for what follows.

Runtime polymorphic behavior

However, the following call to the method does constitute runtime polymorphism.

The statement in Listing 4 (p. 5) casts the reference down to type **A** and calls the method named **m()** on that reference.

It may not come as a surprise to you that the call to the method shown in Listing 4 (p. 5) also compiles and runs successfully.

Listing 4: Polymorphic behavior.

```
((A)var).m();
```

The method output

Here is the punch line. Not only does the statement in Listing 4 (p. 5) compile and run successfully, it produces the following output, (*which is exactly the same output as before*) :

```
m in class B
```

Same method executed in both cases

It is important to note that this output, (*produced by casting the reference variable to type **A** instead of type **B***) , is exactly the same as that produced by the earlier call to the method when the reference was cast to type **B** . This means that the same version of the method was executed in both cases.

This confirms that, even though the type of the reference was converted to type **A** , (*rather than type **Object** or type **B***) , the overridden version of the method defined in class **B** was actually executed.

This is an example of runtime polymorphic behavior.

The version of the method that was executed was based on

- the actual type of the object, **B** , and
- not on the type of the reference, **A** .

This is an extremely powerful and useful concept.

Another call to the method

Now take a look at the statement in Listing 5 (p. 5) . Will this statement compile and execute successfully? If so, which version of the method will be executed?

Listing 5: Source of a compiler error.

```
var.m();
```

Compiler error

The code in Listing 5 (p. 5) attempts, unsuccessfully, to call the method named `m()` using the reference variable named `var`, which is of type `Object`. The result is a compiler error, which, depending on your version of the JDK, will be similar to the following:

```
Poly03.java:40: cannot resolve symbol
symbol   : method m ()
location: class java.lang.Object
    var.m();
        ^
```

Some important rules

The `Object` class does not define a method named `m()`. Therefore, the overridden method named `m()` in the class named `B` is not an overridden version of a method that is defined in the class named `Object`.

Necessary, but not sufficient

Runtime polymorphism based on class inheritance requires that the type of the reference variable be a superclass of the class from which the object (*on which the method will be called*) is instantiated. However, while necessary, that is not sufficient.

The type of the reference variable must also be a class that either *defines or inherits* the method that will ultimately be called on the object.

This method is not defined in the Object class

Since the class named `Object` neither defines nor inherits the method named `m()`, a reference of type `Object` does not qualify as a participant in runtime polymorphic behavior in this case. The attempt to use it as a participant resulted in the compiler error given above.

One additional scenario

Before leaving this topic, let's look at one additional scenario to help you distinguish what is, and what is not, runtime polymorphism. Consider the code shown in Listing 6 (p. 6).

Listing 6: A new object of type A.

```
var = new A();
((A)var).m();
```

A new object of type A

The code in Listing 6 (p. 6) instantiates a new object of the class named `A`, and stores the object's reference in the original reference variable named `var` of type `Object`.

(As a side note, this overwrites the previous contents of the reference variable with a new reference and causes the object whose reference was previously stored there to become eligible for garbage collection.)

Downcast and call the method

Then the code in Listing 6 (p. 6) casts the reference down to type `A`, (*the type of the object to which the reference refers*), and calls the method named `m()` on the downcast reference.

The output

As you would probably predict, this produces the following output on the computer screen:

```
m in class A
```

In this case, the version of the method defined in the class named `A`, (*not the version defined in `B`*) was executed.

Not polymorphic behavior

In my view, this is not polymorphic behavior (*at least it isn't a very useful form of polymorphic behavior*). This code simply converts the type of the reference from type `Object` to the type of the class from which

the object was instantiated, and calls one of its methods. Nothing special takes place regarding a selection among different versions of the method.

Some authors may disagree

While some authors might argue that this is technically runtime polymorphic behavior, in my view at least, it does not illustrate the real benefits of runtime polymorphic behavior. The benefits of runtime polymorphic behavior generally accrue when the actual type of the object is a subclass of the type of the reference variable containing the reference to the object.

Once again, what is runtime polymorphism?

As I have discussed in this module, runtime polymorphic behavior based on class inheritance occurs when

- The type of the reference is a superclass of the class from which the object was instantiated.
- The version of the method that is executed is the version that is either defined in, or inherited into, the class from which the object was instantiated.

More than you ever wanted to hear

And that is probably more than you ever wanted to hear about runtime polymorphism based on class inheritance.

A future module will discuss runtime polymorphism based on the Java interface. From a practical viewpoint, you will find the rules to be similar but somewhat different in the case of the Java interface.

A very important concept

For example, the entire event-driven graphical user interface structure of Java is based on runtime polymorphism involving the Java interface.

5 Summary

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through class inheritance
- Method overriding through the Java interface

This module discusses method overriding through class inheritance.

With runtime polymorphism based on method overriding, the decision as to which version of a method will be executed is based on the actual type of object whose reference is stored in the reference variable, and not on the type of the reference variable on which the method is called.

The decision as to which version of the method to call cannot be made at compile time. That decision must be deferred and made at runtime. This is sometimes referred to as late binding.

This is illustrated in the sample program discussed in this module.

6 What's next?

In the next module, I will continue my discussion of the implementation of polymorphism using method overriding through class inheritance, and I will concentrate on a special case in that category.

Specifically, I will discuss the use of the **Object** class as a completely generic type for storing references to objects of subclass types, and explain how that results in a very useful form of runtime polymorphism.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: OOP 101, Runtime Polymorphism through Class Inheritance
- File: Java1612.htm
- Published: February 27, 2002
- Revised: July 25, 2012
- Keywords:
 - runtime polymorphism
 - class inheritance
 - method overriding

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 7 (p. 8) below.

Listing 7: Complete program listing.

```
/*File Poly03.java
Copyright 2002, R.G.Baldwin
```

This program illustrates downcasting
and polymorphic behavior

Program output is:

```
m in class B
m in class B
m in class A
*****/

class A extends Object{
  public void m(){
    System.out.println("m in class A");
  }//end method m()
} //end class A
//=====//
```

```
class B extends A{
    public void m(){
        System.out.println("m in class B");
    }//end method m()
}//end class B
//=====//

public class Poly03{
    public static void main(String[] args){
        Object var = new B();
        //Following will compile and run
        ((B)var).m();
        //Following will also compile
        // and run due to polymorphic
        // behavior.
        ((A)var).m();
        //Following will not compile
        //var.m();
        //Instantiate obj of class A
        var = new A();
        //Call the method on it
        ((A)var).m();
    }//end main
}//end class Poly03
```

-end-