

JAVA OOP: POLYMORPHISM AND INTERFACES, PART 2*

R.G. (Dick) Baldwin

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

Abstract

Baldwin uses a sample program to illustrate (in a very basic way) some of the things that you can do with interfaces, along with some of the things that you cannot do with interfaces.

1 Table of Contents

- Preface (p. 1)
 - Viewing tip (p. 1)
 - * Listings (p. 2)
- Preview (p. 2)
- Discussion and sample code (p. 3)
- Summary (p. 9)
- What's next? (p. 9)
- Miscellaneous (p. 10)
- Complete program listing (p. 10)

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 listings while you are reading about them..

*Version 1.1: Jul 27, 2012 10:44 pm -0500

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

2.1.1 Listings

- Listing 1 (p. 3) . Definition of the interfaces named I1 and I2.
- Listing 2 (p. 3) . Definition of the class named A.
- Listing 3 (p. 4) . Definition of the class named B.
- Listing 4 (p. 4) . Definition of the class named C.
- Listing 5 (p. 5) . Beginning of the class named Poly06.
- Listing 6 (p. 5) . Try unsuccessfully to call the method named q.
- Listing 7 (p. 6) . Successfully call the method named q.
- Listing 8 (p. 6) . Instantiate a new object of the class B.
- Listing 9 (p. 6) . Try unsuccessfully to call the method named x.
- Listing 10 (p. 7) . Successfully call the method named x.
- Listing 11 (p. 7) . Call the toString method.
- Listing 12 (p. 8) . Try unsuccessfully to call the method named p.
- Listing 13 (p. 8) . Successfully call the method named p.
- Listing 14 (p. 8) . A walk in the park.
- Listing 15 (p. 10) . Complete program listing.

3 Preview

Method overloading

I covered method overloading as one form of polymorphism (*compile-time polymorphism*) in a previous module.

Method overriding and class inheritance

I discussed *runtime polymorphism* implemented through method overriding and class inheritance in more than one previous module.

Using the Java interface

In this and the previous module, I am explaining runtime polymorphism as implemented using method overriding and the Java interface.

A very important concept

In my opinion, this is one of the most important concepts in Java OOP, and the one that seems to give students the greatest amount of difficulty. Therefore, I am trying to take it slow and easy. As usual, I am illustrating the concept using sample programs.

A skeleton program

In the previous module, I presented a simple skeleton program that illustrated many of the important aspects of polymorphic behavior based on the Java interface.

Multiple inheritance and the cardinal rule

I explained how the implementation of interfaces in Java is similar to multiple inheritance. I explained the cardinal rule of interface implementation.

A new relationship

I explained that objects instantiated from classes that implement the same interface have a new relationship that goes beyond the relationship imposed by the standard class hierarchy.

One object, many types

I explained that due to the combination of the class hierarchy and the fact that a class can implement many different interfaces, a single object in Java can be treated as many different types. However, for any given type, there are restrictions on the methods that can be called on the object.

Many classes, one type

I explained that because different classes can implement the same interface, objects instantiated from different classes can be treated as a common interface type.

Interfaces are critical to Java programming

I suggested that there is little if anything useful that can be done in Java without understanding and using interfaces. In support of this suggestion, I discussed several real-world examples of the use of the Java interface, including the Delegation Event Model and the Model View Control paradigm.

Another sample program

In this module, I will present another sample program that will take you deeper into the world of polymorphism as implemented using the Java interface.

The sample program that I will discuss in this module will illustrate (*in a very basic form*) some of the things that you can do with interfaces, along with some of the things that you cannot do with interfaces. In order to write programs that do something worthwhile, you will need to extend the concepts illustrated by this sample program into real-world requirements.

4 Discussion and sample code

Now, let's take a look at a sample program named **Poly06** that is much simpler than any of the real-world examples that you will see in future modules.

This program is designed to be very simple, while still illustrating runtime polymorphism using interfaces, class inheritance, and overridden methods.

You can view a complete listing of the program in Listing 15 (p. 10) near the end of the module.

Same structure as before

Note that this program has the same structure as **Poly05** discussed in the previous module. (*I strongly recommend that you study the previous module before continuing with this module.*) However, unlike the program in the previous module, the methods in this version of the program are not empty. When a method is called in this version, something happens. (*Admittedly not much happens. Text is displayed on the computer screen, but that is something.*)

The interface definitions

Listing 1 (p. 3) shows the definition of the two interfaces named **I1** and **I2** .

Listing 1: Definition of the interfaces named I1 and I2.

```
interface I1{
    public void p();
} //end interface I1

//=====//

interface I2 extends I1{
    public void q();
} //end interface I2
```

Since the methods declared in an interface are not allowed to have a body, these interface definitions are identical to those shown in the program from the previous module.

The class named A

Similarly, Listing 2 (p. 3) shows the definition of the class named **A** along with the definition of the method named **x** , and the overridden method named **toString** .

Listing 2: Definition of the class named A.

```
class A extends Object{

    public String toString(){
        return "toString in A";
    }
}
```

```

} //end toString()
//-----//

public String x(){
    return "x in A";
} //end x()
//-----//
} //end class A

```

These two methods were also fully defined in the program from the previous module, so there is no change here either.

The method named B

Listing 3 (p. 4) defines the class named **B**, which extends **A**, and implements **I2**.

Listing 3: Definition of the class named B.

```

class B extends A implements I2{
public void p(){
    System.out.println("p in B");
} //end p()
//-----//

public void q(){
    System.out.println("q in B");
} //end q();
//-----//
} //end class B

```

Actually implements two interfaces

Although it isn't obvious from an examination of Listing 3 (p. 4) alone, the class named **B** actually implements both **I2** and **I1**. This is because the interface named **I2** extends **I1**. Thus, the class named **B** implements **I2** directly, and implements **I1** through interface inheritance.

The cardinal rule

In case you have forgotten it, the cardinal rule for implementing interfaces is:

If a class implements an interface, it must provide a concrete definition for all the methods declared by that interface, and all the methods inherited by that interface. Otherwise, the class must be declared abstract and the definitions must be provided by a class that extends the abstract class.

Must define two methods

As a result, the class named **B** must provide concrete definitions for the methods **p** and **q**. (The method named *p* is declared in interface **I1** and the method named *q* is declared in interface **I2**.)

As you can see from Listing 3 (p. 4), the behavior of each of these methods is to display a message indicating that it has been executed. This will be useful later to tell us exactly which method is executed when we exercise the objects in the **main** method of the driver class.

The class named C

Listing 4 (p. 4) shows the upgraded version of the class named **C**.

Listing 4: Definition of the class named C.

```

class C extends Object implements I2{
public void p(){
    System.out.println("p in C");
}

```

```

} //end p()
//-----//

public void q(){
    System.out.println("q in C");
} //end q();
//-----//
} //end class B

```

In this upgraded version, the methods named **p** and **q** each display a message indicating that they have been executed. Again, this will be useful later to let us know exactly which version of the methods named **p** and **q** get executed when we exercise the objects.

The driver class

Listing 5 (p. 5) shows the beginning of the class named **Poly06**. The **main** method in this class instantiates objects of the classes named **B** and **C**, and exercises them to illustrate what can, and what cannot be done with them.

Listing 5: Beginning of the class named Poly06.

```

public class Poly06{
public static void main(
    String[] args){
    I1 var1 = new B();
    var1.p();//OK

```

A new data type

As explained in the previous module, when you define a new interface, you create a new data type.

You can store the reference to any object instantiated from any class that implements the interface in a reference variable of that type.

A new object of the class B

The code shown in Listing 5 (p. 5) instantiates a new object of the class **B**.

Important: stored as type I1

It is important to note that the code in Listing 5 (p. 5) stores the object's reference in a reference variable of the interface type **I1** (*not as the class type B*).

Call an interface method

Following this, the code in Listing 5 (p. 5) successfully calls the method named **p** on the reference, producing the following output on the computer screen:

```
p in B
```

Why is this allowed?

This is allowable because the method named **p** is declared in the interface named **I1**.

Which version of the method was executed?

It is also important to note, (*by observing the output*), that the version of the method defined in the class named **B** (*and not the version defined in the class named C*) was actually executed.

Attempt unsuccessfully to call q

Next, the code in Listing 6 (p. 5) attempts, unsuccessfully, to call the method named **q** on the same reference variable of type **I1**.

Listing 6: Try unsuccessfully to call the method named q.

```
var1.q();//won't compile
```

Why did it fail?

Even though the class named **B**, from which the object was instantiated, defines the method named **q**, that method is neither declared nor inherited into the interface named **I1**.

Therefore, a reference of type **I1** cannot be used to call the method named **q**.

The solution is a type conversion

Listing 7 (p. 6) shows the solution to the problem presented by Listing 6 (p. 5).

Listing 7: Successfully call the method named q.

```
((I2)var1).q();//OK
```

As in the case of polymorphism involving class inheritance, the solution is to change the type of the reference to a type that either declares or inherits the method named **q**.

In this case, this takes the form of using a cast operator to convert the type of the reference from type **I1**, to type **I2**, and then calling the method named **q** on that reference of a new type.

This produces the following output:

```
q in B
```

Using type I2 directly

Listing 8 (p. 6) instantiates a new object of the class **B** and stores the object's reference in a reference variable of the interface type **I2**.

Listing 8: Instantiate a new object of the class B.

```
I2 var2 = new B();
var2.p();//OK
var2.q();//OK
```

Call both methods successfully

Then the code successfully calls both the methods **p** and **q** on that reference, producing the following output:

```
p in B
q in B
```

Why does this work?

This works because:

- The interface named **I2** declares the method named **q**
- The interface named **I2** inherits the declaration of the method named **p**
- The class named **B** implements the interface named **I2** and provides concrete definitions of both the methods **p** and **q**.

Attempt, unsuccessfully, to call x on var2

Following this, the code in Listing 9 (p. 6) attempts, unsuccessfully, to call the method named **x** on the reference variable named **var2** of type **I2**. This code produces a compiler error.

Listing 9: Try unsuccessfully to call the method named x.

```
String var3 = var2.x();
```

The object of class B has a method named x

At this point, the reference variable named `var2` contains a reference to an object instantiated from the class named `B`.

Furthermore, the class named `B` inherits the method named `x` from the class named `A`.

Necessary, but not sufficient

However, the fact that the object contains the method is not sufficient to make it executable in this case.

Same song, different verse

The interface named `I2` neither declares nor inherits the method named `x`.

Therefore, the method named `x` cannot be called using the reference stored in the variable named `var2` unless the reference is converted either to type `A` (*where the method named x is defined*) or type `B` (*where the method named x is inherited*).

Do the type conversion

The required type conversion is accomplished in Listing 10 (p. 7) where the reference is temporarily converted to type `A` using a cast operator. (*It would also work to cast it to type B.*)

Listing 10: Successfully call the method named x.

```
String var3 = ((A)var2).x();//OK
System.out.println(var3);
```

The String produced by the first statement in Listing 10 (p. 7) is passed to the `println` method causing the following text to be displayed on the computer screen:

```
x in A
```

Get ready for a surprise

If you have now caught onto the general scheme of things, the next thing that I am going to show you may result in a little surprise.

Successfully call the toString method on var2

The first statement in Listing 11 (p. 7) successfully calls the `toString` method on the object of the class `B` whose reference is stored as type `I2`.

Listing 11: Call the toString method.

```
var3 = var2.toString();//OK
System.out.println(var3);
```

How can this work?

How can this work when the interface named `I2` neither declares nor inherits a method named `toString`?

A subtle difference in behavior

I am unable to point you to any Sun documentation to verify the following. (*I also admit that I haven't spent a large amount of time searching for such documentation*).

With respect to the eleven methods declared in the `Object` class (*listed in an earlier module*), a reference of an interface type acts like it is also of type `Object`.

And the end result is ...

This allows the methods declared in the `Object` class to be called on references held as interface types without a requirement to cast the references to type `Object`. (*Later, I will show you that the reverse is not true.*)

The output

Therefore, the two statements shown in Listing 11 (p. 7) cause the following to be displayed on the computer screen:

`toString` in `A`

Polymorphism applies

Note that the object whose reference is held in `var2` was instantiated from the class named `B`, which extends the class named `A`.

Due to polymorphism, the `toString` method that was actually executed in Listing 11 (p. 7) was the overridden version defined in class `A`, and not the default version defined in the `Object` class. The overridden version in class `A` was inherited into class `B`.

The reverse is not true

While a reference of an interface type also acts like type `Object`, a reference of type `Object` does not act like an interface type.

Store a reference as type `Object`

The code in Listing 12 (p. 8) instantiates a new object of type `B` and stores it in a reference of type `Object`.

Attempt unsuccessfully to call `p`

Then it attempts, unsuccessfully, to call the method named `p` on the reference.

Listing 12: Try unsuccessfully to call the method named `p`.

```
Object var4 = new B();
var4.p(); //won't compile
```

Same song, an even different verse

The code in Listing 12 (p. 8) won't compile, because the `Object` class neither defines nor inherits the method named `p`.

In order to call the method named `p` on the reference of type `Object`, the type of the reference must be changed to either:

- The class in which the method is defined
- An interface that declares the method, which is implemented by the class in which the method is defined
- A couple of other possibilities involving subclasses or sub-interfaces

Convert reference to type `I1`

The code in Listing 13 (p. 8) uses a cast operator to convert the reference from type `Object` to type `I1`, and calls the method named `p` on the converted reference.

Listing 13: Successfully call the method named `p`.

```
((I1)var4).p(); //OK
```

The output

The code in Listing 13 (p. 8) compiles and executes successfully, producing the following text on the computer screen:

```
p in B
```

A walk in the park

If you understand all of the above, understanding the code in Listing 14 (p. 8) should be like a walk in the park on a sunny day.

Listing 14: A walk in the park.

```

    var2 = new C();
var2.p();//OK
var2.q();//OK

```

Class C implements I2

Recall that the class named `C` also implements the interface named `I2`.

The code in Listing 14 (p. 8) instantiates a new object of the class named `C`, and stores the object's reference in the existing reference variable named `var2` of type `I2`.

Then it calls the methods named `p` and `q` on that reference, causing the following text to be displayed on the computer screen:

```

p in C
q in C

```

Which methods were executed?

This confirms that the methods that were actually executed were the versions defined in the class named `C` (and not the versions defined in the class named `B`).

Same method name, different behavior

It is important to note that the behavior of the methods named `p` and `q`, as defined in the class named `C`, is different from the behavior of the methods having the same signatures defined in the class named `B`. Therein lies much of the power of the Java interface.

The power of the Java interface

Using interface types, it is possible to collect many objects instantiated from many different classes (provided all the classes implement a common interface), and store each of the references in some kind of collection as the interface type.

Appropriate behavior

Then it is possible to call any of the interface methods on any of the objects whose references are stored in the collection.

To use the current jargon, when a given interface method is called on a given reference, the behavior that results will be *appropriate* to the class from which that particular object was instantiated.

This is runtime polymorphism based on interfaces and overridden methods.

5 Summary

If you don't understand interfaces ...

If you don't understand interfaces, you don't understand Java, and it is highly unlikely that you will be successful as a Java programmer.

Interfaces are indispensable in Java

Beyond writing "Hello World" programs, there is little if anything that can be accomplished using Java without understanding and using interfaces.

What can you do with interfaces?

The sample program that I discussed in this module has illustrated (*in a very basic form*) some of the things that you can do with interfaces, along with some of the things that you cannot do with interfaces.

In order to write programs that do something worthwhile, you will need to extend the concepts illustrated by this sample program into real-world requirements.

6 What's next?

Java supports the use of **static** member variables and **static** methods in class definitions.

While **static** members can be useful in some situations, the existence of **static** members tends to complicate the overall object-oriented structure of Java.

Furthermore, the overuse of **static** members can lead to problems similar to those experienced in languages like C and C++ that support global variables and global functions.

The use of static members will be discussed in the next module.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Polymorphism and Interfaces, Part 2
- File: Java1618.htm
- Published: April 10, 2002
- Revised: July 27, 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 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 sample program is shown in Listing 15 (p. 10) below.

Listing 15: Complete program listing.

```
/*File Poly06.java  
Copyright 2002, R.G.Baldwin
```

```
This program illustrates polymorphic  
behavior using interfaces in addition  
to class inheritance.
```

The program output is:

```
p in B  
q in B
```

```
p in B  
q in B  
x in A
```

```
toString in A

p in B

p in C
q in C
*****/

interface I1{
    public void p();
} //end interface I1
//=====//

interface I2 extends I1{
    public void q();
} //end interface I2
//=====//

class A extends Object{

    public String toString(){
        return "toString in A";
    } //end toString()
    //-----//

    public String x(){
        return "x in A";
    } //end x()
    //-----//
} //end class A
//=====//

class B extends A implements I2{
    public void p(){
        System.out.println("p in B");
    } //end p()
    //-----//

    public void q(){
        System.out.println("q in B");
    } //end q();
    //-----//
} //end class B
//=====//

class C extends Object implements I2{
    public void p(){
        System.out.println("p in C");
    } //end p()
    //-----//
```

```
public void q(){
    System.out.println("q in C");
} //end q();
//-----//
} //end class B
//=====//

public class Poly06{
    public static void main(
        String[] args){
        I1 var1 = new B();
        var1.p();//OK
        //var1.q();//won't compile
        ((I2)var1).q();//OK
        System.out.println(""); //blank line

        I2 var2 = new B();
        var2.p();//OK
        var2.q();//OK
        //Following won't compile
        //String var3 = var2.x();
        String var3 = ((A)var2).x();//OK
        System.out.println(var3);
        var3 = var2.toString();//OK
        System.out.println(var3);
        System.out.println(""); //blank line

        Object var4 = new B();
        //var4.p();//won't compile
        ((I1)var4).p();//OK
        System.out.println(""); //blank line

        var2 = new C();
        var2.p();//OK
        var2.q();//OK
        System.out.println(""); //blank line
    } //end main
} //end class Poly06
```

-end-