Java OOP: Polymorphism, Type Conversion, Casting, etc.*

Richard Baldwin

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License †

Abstract

Baldwin teaches you about assignment compatibility, type conversion, and casting for both primitive
and reference types. He also teaches you about the relationship between reference types, method calls,
and the location in the class hierarchy where a method is defined.

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 listings while you are reading about them.

---

*Version 1.1: Jul 27, 2012 3:57 pm -0500

†http://creativecommons.org/licenses/by/3.0/

2.1.1 Listings

3 Preview

This module discusses type conversion for both *primitive* and *reference* types.

A value of a particular type may be *assignment compatible* with variables of other types, in which case the value can be assigned directly to the variable. Otherwise, it may be possible to perform a *cast* on the value to change its type and assign it to the variable as the new type.

With regard to reference types, whether or not a cast can be successfully performed

- depends on the relationships of the classes involved in the class hierarchy.

A reference to any object can be assigned to a reference variable of the type **Object** , because the **Object** class is a superclass of every other class.

When we cast a reference along the class hierarchy in a direction from the root class **Object** toward the leaves, we often refer to it as a *downcast* .

Whether or not a method can be called on a reference to an object depends on

- the current type of the reference, and
- the location in the class hierarchy where the method is defined.

In order to use a reference of a class type to call a method, the method must be defined at or above that class in the class hierarchy.

A sample program is provided that illustrates much of the detail involved in type conversion, method calls, and casting with respect to reference types.

4 Discussion and sample code

What is polymorphism?

As a quick review, 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 inheritance
- Method overriding through the Java interface

I covered method overloading as one form of polymorphism in a previous module.

We need to backtrack

In this module, I will backtrack a bit and discuss the conversion of references from one type to another.

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

Assignment compatibility and type conversion

As a background for polymorphism, you need to understand something about *assignment compatibility* and *type conversion* .

A value of a given type is assignment compatible with another type if

- a value of the first type
- can be successfully assigned to a variable of the second type.

Type conversion and the cast operator

In some cases, type conversion happens automatically. In other cases, type conversion must be forced through the use of a *cast operator* .

A cast operator is a *unary* operator, which has a single right operand. The physical representation of the cast operator is the name of a type inside a pair of matched parentheses, as in:

(int)

Applying a cast operator

Applying a cast operator to the name of a variable doesn't actually change the type of the variable. However, it does cause the contents of the variable to be treated as a different type for the evaluation of the expression in which the cast operator is contained. Thus, the application of a cast operator is a short-term event.

Primitive values and type conversion

Assignment compatibility issues come into play for both primitive types and reference types.

However, values of type **boolean** can only be assigned to variables of type boolean *(you cannot change the type of a boolean)* .

Otherwise, a primitive value can be assigned to any variable of a type

- whose range is as wide or wider
- than the range of the type of the value.

In that case, the type of the value is automatically converted to the type of the variable.

(For example, types **byte** *and* **short** *can be assigned to a variable of type* **int** *without the requirement for a cast because type* **int** *has a wider range than either type* **byte** *or type* **short** *.)*

Conversion to narrower range

On the other hand, a primitive value of a given type cannot be assigned to a variable of a type with a narrower range than the type of the value,

- unless the cast operator is used to force a type conversion.

Oftentimes, such a conversion will result in the loss of data, and that loss is the responsibility of the programmer who performs the cast.

Assignment compatibility for references

Assignment compatibility, with respect to references, doesn't involve range issues, as is the case with primitives. Instead, the reference to an object instantiated from a given class can be assigned to:

- **Any reference variable** whose type is the same as the class from which the object was instantiated.

- Any reference variable whose type is a superclass of the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by a superclass of the class from which the object was instantiated, and
- A few other cases involving the class and interface hierarchy.

Such an assignment does not require the use of a cast operator.

Type Object is completely generic

A reference to any object can be assigned to a reference variable of the type **Object**, because the **Object** class is a superclass of every other class.

Converting reference types with a cast

Assignments of references, other than those listed above (p. 4), require the use of a cast operator to purposely change the type of the reference.

Doesn't work in all cases

However, it is not possible to perform a successful cast to convert the type of a reference in all cases.

Generally, a cast can only be performed among reference types that fall on the same ancestral line of the class hierarchy, or on an ancestral line of an interface hierarchy. For example, a reference cannot be successfully cast to the type of a sibling or a cousin in the class hierarchy.

Downcasting

When we cast a reference along the class hierarchy in a direction from the root class **Object** toward the leaves, we often refer to it as a *downcast*.

While it is also possible to cast in the direction from the leaves to the root, this happens automatically, and the use of a cast operator is not required.

A sample program

The program named **Poly02**, shown in Listing 11 (p. 9) near the end of the module, illustrates the use of the cast operator with references.

When you examine that program, you will see that two classes named **A** and **C** each extend the class named **Object**. Hence, we might say that they are siblings in the class hierarchy.

Another class named **B** extends the class named **A**. Thus, we might say that **A** is a child of **Object**, and **B** is a child of **A**.

The class named A

The definition of the class named A is shown in Listing 1 (p. 4). This class extends the class named **Object**.

*(Recall that it is not necessary to explicitly state that a class extends the class named **Object**. Any class that does not explicitly extend some other class will automatically extend **Object** by default. The class named **A** is shown to extend **Object** here simply for clarity of presentation.)*

Listing 1: Definition of the class named A.

```
  class A extends Object{
  //this class is empty
}//end class A
```

The class named **A** is empty. It was included in this example for the sole purpose of adding a layer of inheritance to the class hierarchy.

The class named B

Listing 2 (p. 5) shows the definition of the class named **B**. This class 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 method named m()

The class named **B** defines a method named **m()** . The behavior of the method is simply to display a message each time it is called.

The class named C

Listing 3 (p. 5) contains the definition of the class named **C** , which also extends **Object** .

Listing 3: Definition of the class named C.

```
class C extends Object{
  //this class is empty
}//end class C
```

The class named **C** is also empty. It was included in this example as a sibling class for the class named **A** . Stated differently, it was included as a class that is not in the ancestral line of the class named **B** .

The driver class

Listing 4 (p. 5) shows the beginning of the driver class named **Poly02** .

Listing 4: Beginning of the class named Poly02.

```
public class Poly02{
  public static void main(String[] args){
    Object var = new B();
```

An object of the class named B

The code in Listing 4 (p. 5) instantiates an object of the class **B** and assigns the object's reference to a reference variable of type **Object** .

*(It is important to note that the reference to the object of type **B** was not assigned to a reference variable of type **B** . Instead, it was assigned to a reference variable of type **Object** .)*

This assignment is allowable because **Object** is a superclass of **B** . In other words, the reference to the object of the class **B** is assignment compatible with a reference variable of the type **Object** .

Automatic type conversion

In this case, the reference of type **B** is automatically converted to type **Object** and assigned to the reference variable of type **Object** . *(Note that the use of a cast operator was not required in this assignment.)*

Only part of the story

However, assignment compatibility is only part of the story. The simple fact that a reference is assignment compatible with a reference variable of a given type says nothing about what can be done with the reference after it is assigned to the reference variable.

An illegal operation

For example, in this case, the reference variable that was automatically converted to type **Object** cannot be used directly to call the method named **m()** on the object of type **B** . This is indicated in Listing 5 (p. 6) .

Listing 5: An illegal operation.

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

An attempt to call the method named **m()** on the reference variable of type **Object** in Listing 5 (p. 6)
resulted in a compiler error. It was necessary to convert the statement into a comment in order to get the
program to compile successfully.

An important rule

In order to use a reference of a class type to call a method, the method must be defined at or above that
class in the class hierarchy.

This case violates the rule

In this case, the method named **m()** is defined in the class named **B** , which is two levels down from
the class named **Object** .

When the reference to the object of the class **B** was assigned to the reference variable of type **Object**
, the type of the reference was automatically converted to type **Object** .

Therefore, because the reference is of type **Object** , it cannot be used directly to call the method
named **m()** .

The solution is a downcast

In this case, the solution to the problem is a downcast. The code in Listing 6 (p. 6) shows an attempt
to solve the problem by casting the reference down the hierarchy to type **A** .

Listing 6: An ineffective downcast.

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

Still doesn't solve the problem

However, this still doesn't solve the problem, and the result is another compiler error. Again, it was
necessary to convert the statement into a comment in order to get the program to compile.

What is the problem here?

The problem is that the downcast simply didn't go far enough down the inheritance hierarchy.

The class named **A** neither defines nor inherits the method named **m()** . The method named **m()**
is defined in class **B** , which is a subclass of class **A** .

Therefore, a reference of type **A** is no more useful than a reference of type **Object** insofar as calling
the method named **m()** is concerned.

The real solution

The solution to the problem is shown in Listing 7 (p. 6) .

Listing 7: A downcast to type B.

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

The code in Listing 7 (p. 6) casts *(converts)* the reference value contained in the **Object** variable named
var down to type **B** .

The method named **m()** is defined in the class named **B** . Therefore, a reference of type **B** can be
used to call the method.

The code in Listing 7 (p. 6) compiles and executes successfully. This causes the method named **m()** to
execute, producing the following output on the computer screen.

```
m in class B
```

A few odds and ends

Before leaving this topic, let's look at a couple more issues. The code in Listing 8 (p. 7) declares and populates a new variable of type **B** .

Listing 8: Declare a variable of type B.

```
B v1 = (B)var;
```

The code in Listing 8 also uses a cast to:

- Convert the contents of the **Object** variable to type **B**
- **Assign the converted reference to the new reference variable of type B.**

A legal operation

This is a legal operation. In this class hierarchy, the reference to the object of the class **B** can be assigned to a reference variable of the types **B** , **A** , or **Object** .

Cannot be assigned to type C

However, the reference to the object of the class **B** cannot be assigned to a reference variable of any other type, including the type **C** . An attempt to do so is shown in Listing 9 (p. 7) .

Listing 9: Cannot be assigned to type C.

```
//C v2 = (C)var;
```

The code in Listing 9 (p. 7) attempts to cast the reference to type **C** and assign it to a reference variable of type **C** .

A runtime error

Although the program will compile, it won't execute. An attempt to execute the statement in Listing 9 (p. 7) results in a **ClassCastException** at runtime. As a result, it was necessary to convert the statement into a comment in order to execute the program.

Another failed attempt

Similarly, an attempt to cast the reference to type B and assign it to a reference variable of type **C** , as shown in Listing 10 (p. 7) , won't compile.

Listing 10: Another failed attempt.

```
//C v3 = (B)var;
```

The problem here is that the class **C** is not a superclass of the class named **B** . Therefore, a reference of type **B** is not assignment compatible with a reference variable of type **C** .

Again, it was necessary to convert the statement into a comment in order to compile the program.

5 Summary

This module discusses type conversion for both primitive and reference types.

A value of a particular type may be assignment compatible with variables of other types.

If the type of a value is not assignment compatible with a variable of a given type, it may be possible to perform a cast on the value to change its type and assign it to the variable as the new type. For primitive types, this will often result in the loss of information.

Except for type **boolean** , values of primitive types can be assigned to any variable whose type represents a range that is as wide or wider than the range of the value's type. (*Values of type* **boolean** *can only be assigned to variables of type boolean.*)

With respect to reference types, the reference to an object instantiated from a given class can be assigned to any of the following without the use of a cast:

- Any reference variable whose type is the same as the class from which the object was instantiated.
- Any reference variable whose type is a superclass of the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by a superclass of the class from which the object was instantiated.
- A few other cases involving the class and interface hierarchy.

Assignments of references, other than those listed above, require the use of a cast to change the type of the reference.

It is not always possible to perform a successful cast to convert the type of a reference. Whether or not a cast can be successfully performed depends on the relationship of the classes involved in the class hierarchy.

A reference to any object can be assigned to a reference variable of the type **Object**, because the **Object** class is a superclass of every other class.

When we cast a reference along the class hierarchy in a direction from the root class **Object** toward the leaves, we often refer to it as a downcast.

Whether or not a method can be called on a reference to an object depends on the current type of the reference and the location in the class hierarchy where the method is defined. In order to use a reference of a class type to call a method, the method must be defined at or above that class in the class hierarchy.

A sample program is provided that illustrates much of the detail involved in type conversion, method invocation, and casting with respect to reference types.

6 What's next?

I will begin the discussion of runtime polymorphism through method overriding and inheritance in the next module.

I will demonstrate that for 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.

7 Miscellaneous

This section contains a variety of miscellaneous information.

> NOTE: **Housekeeping material**
>
> - Module name: OOP 101, Polymorphism, Type Conversion, Casting, etc.
> - File: Java1610.htm
> - Published: February 26, 2002
> - Revised: July 25, 2012
> - Keywords:
>   · assignment compatibility
>   · type conversion
>   · casting
>   · primitive types
>   · reference types

> 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, con-
verted 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 listings

A complete listing of the program is shown in Listing 11 (p. 9) below.

Listing 11: Complete program listing.

```
    /*File Poly02.java
Copyright 2002, R.G.Baldwin

This program illustrates downcasting

Program output is:

m in class B
**************************************/

class A extends Object{
  //this class is empty
}//end class A
//==================================//

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

class C extends Object{
  //this class is empty
}//end class C
//==================================//

public class Poly02{
  public static void main(String[] args){
    Object var = new B();
    //Following will not compile
    //var.m();
    //Following will not compile
    //((A)var).m();
```

```
    //Following will compile and run
    ((B)var).m();

    //Following will compile and run
    B v1 = (B)var;
    //Following will not execute
    //C v2 = (C)var;
    //Following will not compile
    //C v3 = (B)var;
  }//end main
}//end class Poly02
```

-end-