Java OOP: The this and super Keywords*

Richard Baldwin

Abstract

Baldwin explains the use of the keywords this and super, and provides sample programs to illustrate the use these keywords for several purposes.

1 Table of Contents

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. 3) . The extends keyword.

*Version 1.2: Aug 8, 2012 10:31 am -0500
†http://creativecommons.org/licenses/by/3.0/

2.1.2 Listings

3 Preview

This module explains the use of the keywords ***this*** and ***super*** . Short sample programs illustrate how you can use these keywords for several purposes.

I will discuss and illustrate the use of the **this** keyword in the following situations:

- To bypass local variables or parameters that hide member variables having the same name, in order to access the member variable.
- To make it possible for one overloaded constructor to call another overloaded constructor in the same class.
- To pass a reference to the current object to a method belonging to a different object *(as in implementing callbacks, for example)*.

I will also discuss and illustrate the use of the **super** keyword in the following situations:

- To bypass the overridden version of a method in a subclass and execute the version in the superclass.
- To bypass a member variable in a subclass in order to access a member variable having the same name in a superclass.
- To cause a constructor in a subclass to call a parameterized constructor in the immediate superclass.

4 Discussion and sample code

You already know quite a lot about OOP

By now you know that an *object* is an *instance of a class* . You know that all variables and methods in Java must be contained in a class or an object. You know that the three primary characteristics of an object-oriented programming language are:

- *encapsulation*
- *inheritance*
- *polymorphism* .

If you have been studying this series of modules on the Essence of OOP in Java, you already know quite a lot about OOP in general, and the implementation of OOP in Java in particular.

A few more important OOP/Java concepts

However, there are a few more important concepts that I haven't previously discussed in this series of modules. In this module, I will explain the use of the keywords ***this*** and ***super*** .

Data and methods

The class provides the plan from which objects are built. This plan defines the *data* that is to be stored in an object, and the *methods* for manipulating that data. The data is variously referred to as *data members, fields* , and *variables* , depending on which book you are reading.

Non-static and static

The data can be further sub-divided into *non-static* and *static* , often referred to as i *nstance variables* and *class variables* respectively.

The methods are also often referred to as *member methods* , and they can also be *static* or *non-static* . Static methods are often referred to as *class methods* while non-static methods are often referred to as *instance methods* .

Instance variables and instance methods

The class body contains the *declarations* for, and possibly the *initialization* of all data members *(both class variables and instance variables)* as well as the full definition of all *methods* .

In this module, we will be particularly interested in instance variables and instance methods.

Every class is a subclass of Object

By default, every class in Java extends *(either directly or indirectly)* the class named **Object** . A new class may either extend **Object** , or extend another class that extends **Object** , or extend another class further down the inheritance hierarchy.

The immediate parent class of a new class is known as its *superclass* , and the new class is known as the *subclass* .

(Sometimes we use the word superclass to indicate the collection of classes in the inheritance hierarchy from which a specific class is derived.)

If you do not specify the *superclass* for a new class, it will extend **Object** by default.

The *extends* keyword

The keyword **extends** is used in the class declaration to specify the immediate *superclass* of the new class using the syntax shown in Figure 1 (p. 3) .

The extends keyword.

```
    class NewClass extends SuperClassName{
  //body of class
}//end class definition
```

Figure 1: The extends keyword.

Inheritance

A class inherits the variables and methods of its superclass, and of the superclass of that class, etc., all the way back up the family tree to the single class **Object** , which is the root of all inheritance.

Thus, an object that is instantiated from a class contains all the instance variables and all the instance methods defined by that that class and defined by all its ancestors.

However, the methods may have been *overridden* one or more times along the way. Also, access to those variables and methods may have been restricted through the use of the **public** , **private** , and **protected** keywords.

(There is another access level, often referred to as ***package private*** *, which is what you get when you don't use an access keyword.)*

The this keyword

Every instance method in every object in Java receives a reference named **this** when the method is called. The reference named **this** is a reference to the object on which the method was called. It can be used for any purpose for which such a reference is needed.

Three common situations

There are at least three common situations where such a reference is needed:

- To bypass local variables or parameters that hide member variables having the same name, in order to access the member variable.
- To make it possible for one overloaded constructor to call another overloaded constructor in the same class.
- To pass a reference to the current object to a method belonging to a different object *(as in implementing callbacks, for example).*

Normally, instance methods belonging to an object have direct access to the instance variables belonging to that object, and to the class variables belonging to the class from which that object was instantiated.

(Class methods never have access to instance variables or instance methods.)

Name can be duplicated

However, the name of a method parameter or constructor parameter can be the same as the name of an instance variable belonging to the object or a class variable belonging to the class. It is also allowable for the name of a local variable to be the same as the name of an instance variable or a class variable. In this case, the local variable or the parameter is said to *hide* the member variable having the same name.

Reference named *this* is passed to instance methods

As mentioned above, whenever an instance method is called on an object, a hidden reference named **this** is always passed to the method. The **this** reference always refers to the object on which the method was called. This makes it possible for the code in the method to refer back to the object on which the method was called.

The reference named **this** can be used to access the member variables hidden by the local variables or parameters having of the same name.

The sample program named This01

The sample program shown in Listing 1 (p. 4) illustrates the use of the **this** reference to access a hidden instance variable named **myVar** and a hidden class variable named **yourVar** .

Listing 1: The program named This01.

```
    /*File This01.java
Copyright 2002, R.G.Baldwin
Illustrates use of this keyword to
access hidden member variables.

Tested using JDK 1.4.0 under Win2000

The output from this program is:

myVar parameter = 20
local yourVar variable = 1
Instance variable myVar = 5
Class variable yourVar = 10
************************************/

class This01 {
  int myVar = 0;
  static int yourVar = 0;

  //Constructor with parameters named
```

```
  // myVar and yourVar
  public This01(int myVar,int yourVar){
    this.myVar = myVar;
    this.yourVar = yourVar;
  }//end constructor
  //---------------------------------//

  //Method with parameter named myVar
  // and local variable named yourVar
  void myMethod(int myVar){
    int yourVar = 1;
    System.out.println(
         "myVar parameter = " + myVar);
    System.out.println(
            "local yourVar variable = "
                            + yourVar);
    System.out.println(
           "Instance variable myVar = "
                         + this.myVar);
    System.out.println(
            "Class variable yourVar = "
                       + this.yourVar);
  }//end myMethod
  //---------------------------------//

  public static void main(
                        String[] args){
    This01 obj = new This01(5,10);
    obj.myMethod(20);
  }//end main method
}//End This01 class definition.
```

The key points

The key points to observe in the program is Listing 1 (p. 4) are:

- When the code refers to **myVar** or **yourVar** , the reference resolves to either an incoming parameter or to a local variable having that name.
- When the code refers to **this.myVar** or **this.yourVar** , the reference resolves to the corresponding instance variable and class variable having that name.

To summarize this situation, every time an instance method is called, it receives a hidden reference named **this** . That is a reference to the object on which the method was called.

The code in the method can use that reference to access any instance member of the object on which it was called, or any class member of the class from which the object was instantiated.

However, when class methods are called, they do not receive such a hidden reference, and therefore, they cannot refer to any instance members of any object instantiated from the class. They can only access class members of the same class.

Calling other constructors of the same class

Now I am going to discuss and illustrate the second common situation listed earlier.

A class can define two or more overloaded constructors having the same name and different argument lists. Sometimes it is useful for one overloaded constructor to call another overloaded constructor in the

same class. When this is done, the constructor being called is referred to as though it were a method whose name is **this** , and whose argument list matches the argument list of the constructor being called.

The sample program named This02

This situation is illustrated in the program named **This02** shown in Listing 2 (p. 6) .

Listing 2: The program named This02.

```
    /*File This02.java
Copyright 2002, R.G.Baldwin
Illustrates use of this keyword for one
overloaded constructor to access
another overloaded constructor of the
same class.

Tested using JDK 1.4.0 under Win2000

The output from this program is:

Instance variable myVar = 15
************************************/

class This02 {
  int myVar = 0;

  public static void main(
                       String[] args){
    This02 obj = new This02();
    obj.myMethod();
  }//end main method
  //---------------------------------//

  //Constructor with no parameters
  public This02(){
    //Call parameterized constructor
    this(15);
  }//end constructor
  //---------------------------------//

  //Constructor with one parameter
  public This02(int var){
    myVar = var;
  }//end constructor
  //---------------------------------//

  //Method to display member variable
  // named myVar
  void myMethod(){
    System.out.println(
           "Instance variable myVar = "
                              + myVar);
  }//end myMethod
```

```
}//End This02 class definition.
```

Calling a *noarg* constructor

The **main** method in Listing 2 (p. 6) instantiates a new object by applying the **new** operator to the *noarg* constructor for the class named **This02** .

(The common jargon for a constructor that doesn't take any parameters is a noarg constructor.)

The *noarg* constructor calls a parameterized constructor

The code in the *noarg* constructor uses the **this** keyword to call the other overloaded constructor, passing an **int** value of 15 as a parameter.

That constructor stores the value of the incoming parameter (15) in the instance variable named **myVar** . Then control returns to the *noarg* constructor, which in turn returns control to the **main** method. When control returns to the **main** method, the new object has been constructed, and the instance variable named **myVar** belonging to that object contains the value 15.

Display the value of the instance variable

The next statement in the **main** method calls the method named **myMethod** on the object, which causes the value stored in the instance variable (15) to be displayed on the screen.

The most important statement

For purposes of this discussion, the most important statement in the program is the statement that reads:

```
this(15);
```

This is the statement used by one overloaded constructor to call another overloaded constructor.

Callbacks

An extremely important concept in programming is the third situation mentioned in the earlier list (p. 4) . This is a situation where a method in one object calls a method in another object and passes a reference to itself as a parameter.

(This is sometimes referred to as registration. That is to say, one object registers itself on another object.)

The method in the second object saves the reference that it receives as an incoming parameter. This makes it possible for a method in the second object to make a callback to the first object sometime later. This is illustrated in the program named **This03** , shown in Listing 3 (p. 7) .

Listing 3: The program named This03 .

```
    /*File This03.java
Copyright 2002, R.G.Baldwin
Illustrates using the this keyword in
a callback scenario.

Tested using JDK 1.4.0 under Win2000

The output from this program is:

Instance variable myVar = 15
************************************/

class This03 {
  public static void main(
                        String[] args){
    ClassA objA = new ClassA();
    ClassB objB = new ClassB();
```

```
    objA.goRegister(objB);
    objB.callHimBack();
    objA.showData();
  }//end main method
}//End This03 class definition.
//===================================//

class ClassA{
  int myVar;

  void goRegister(ClassB refToObj){
    refToObj.registerMe(this);

  }//end goRegister
  //---------------------------------//

  void callMeBack(int var){
    myVar = var;
  }//end callMeBack
  //---------------------------------//

  void showData(){
    System.out.println(
           "Instance variable myVar = "
                              + myVar);
  }//end showData
}//end ClassA
//===================================//

class ClassB{
  ClassA ref;

  void registerMe(ClassA var){
    ref = var;
  }//end registerMe
  //---------------------------------//

  void callHimBack(){
    ref.callMeBack(15);
  }//end callHimBack

}//End ClassB class definition
```

Not intended to be useful

Note that the program in Listing 3 (p. 7) is intended solely to illustrate the concept of a callback, and is not intended to do anything useful. This is a rather long and convoluted explanation, so please bear with me.

The **main** method begins by instantiating two objects, one each from the classes named **ClassA** and **ClassB** .

Go register yourself

Then the **main** method sends a message to **objA** telling it to go register itself on **objB** . A reference

to **objB** is passed as a parameter to the method named **goRegister** belonging to **objA** .

The code in **objA** uses this reference to call the method named **registerMe** on **objB** , passing **this** as a parameter. In other words, the code in **objA** calls a method belonging to **objB** passing a reference to itself as a parameter. The code in **objB** saves that reference in an instance variable for later use.

Make a callback

Then the **main** method sends a message to **objB** asking it to use the saved reference to make a callback to **objA** . The code in the method named **callHimBack** uses the reference to **objA** saved earlier to call the method named **callMeBack** on **objA** , passing 15 as a parameter. The method named **callMeBack** belonging to **objA** saves that value in an instance variable.

Show the data

Finally, the **main** method calls the **showData** method on **objA** to cause the value stored in the instance variable belonging to **objA** to be displayed on the computer screen.

Callbacks are important

Again, this program is provided solely to illustrate the concept of a callback using the **this** keyword. In practice, callbacks are used throughout Java, but they are implemented in a somewhat more elegant way, making use of interfaces.

For example, interfaces with names like **Observer** and **MouseListener** are commonly used to register *observer* objects on *observable* objects *(sometimes referred to as listeners and sources).* Then later in the program, when something of interest happens on the *observable* object *(the source),* all registered *observer* objects *(the listeners),* are notified of the event.

The main point regarding the *this* reference

The main point of this discussion is that the **this** reference is available to all instance methods belonging to an object, and can be used whenever there is a need for a reference to the object on which the method is called.

To disambiguate something

At least one prominent author uses the word *disambiguate* to describe the process described by the first item in the earlier list (p. 4) , where the **this** keyword is used to bypass one variable in favor of a different variable having the same name. I will also use that terminology in the following discussion.

Three uses of the *super* keyword

Here are three common uses of the **super** keyword:

- If your class *overrides* a method in a superclass, you can use the **super** keyword to bypass the overridden version in the class and execute the version in the superclass.
- If a local variable in your method or a member variable in your class hides a member variable in the superclass *(having the same name),* you can use the **super** keyword to access the member variable in the superclass.
- You can also use **super** in a constructor of your class to call a parameterized constructor in the superclass.

The program named Super3

The program in Listing 4 (p. 9) uses **super** to call a parameterized constructor in the superclass from the subclass constructor. This is an important use of **super** .

The program also uses **this** and **super** to disambiguate a local variable, an instance variable of the subclass, and an instance variable of the superclass. All three variables have the same name.

Listing 4: The program named Super3.

```
    /*File Super3.java
Copyright 2002, R.G.Baldwin
Illustrates use of super reference to
```

```
access constructor in superclass.  Also
illustrates use of super to
disambiguate instance variable in
subclass from instance variable in
superclass.  Illustrates use of this
to disambiguate local variable from
instance variable in subclass.

Tested using JDK 1.4.0 under Win2000

The output from this program is:

In SuperClass constructor.
Setting superclass instance var to 500

In subclass constructor.
Setting subclass instance var to 400

In main
Subclass instance var = 400

In method myMeth
Local var = 300
Subclass instance var = 400
SuperClass instance var = 500
************************************/
class SuperClass{
  int data;

  //Parameterized superclass
  // constructor
  public SuperClass(int val){
    System.out.println(
        "In SuperClass constructor. ");
    System.out.println(
         "Setting superclass instance "
                   + "var to " + val);
    data = val;
    System.out.println();//blank line
  }//end SuperClass constructor
}//end SuperClass class definition
//===================================//

class Super3 extends SuperClass{
  //Instance var in subclass has same
  // name as instance var in superclass
  int data;

  //Subclass constructor
  public Super3(){
    //Call parameterized SuperClass
```

```
    // constructor
    super(500);
    System.out.println(
           "In subclass constructor.");
    System.out.println(
       "Setting subclass instance var "
                            + "to 400");
    data = 400;
    System.out.println();//blank line
  }//end subclass constructor
  //---------------------------------//

  //Method illustrates use of this and
  // super to disambiguate local
  // variable, instance variable of
  // subclass, and instance variable
  // of superclass.  All three
  // variables have the same name.
  void myMeth(){
    int data = 300;//local variable
    System.out.println(
                   "In method myMeth");
    System.out.println("Local var = "
                              + data);
    System.out.println(
             "Subclass instance var = "
                           + this.data);
    System.out.println(
           "SuperClass instance var = "
                          + super.data);
  }//end method myMeth
  //---------------------------------//

  public static void main(
                       String[] args){
    Super3 obj = new Super3();
    System.out.println("In main");
    System.out.println(
             "Subclass instance var = "
                            + obj.data);
    System.out.println();//blank line
    obj.myMeth();
  }//end main method
}//End Super3 class definition.
```

The keyword **super** is used twice in the program in Listing 4 (p. 9) .

Call a parameterized constructor

The first usage of the keyword **super** appears as the first executable statement in the *noarg* constructor for the class named **Super3** . This statement reads as follows:

```
    super(500);
```

This statement causes the parameterized constructor for the immediate superclass *(the class named* ***SuperClass*** *)* of the class named **Super3** , to be executed before the remaining code in the constructor for **Super3** is executed.

This is the mechanism by which you can cause a parameterized constructor in the immediate superclass to be executed.

What if you don't do this?

If you don't do this, an attempt will always be made to call a *noarg* constructor on the superclass before executing the remaining code in the constructor for your class.

(That is why you should almost always make certain that the classes that you define have a noarg constructor in addition to any parameterized constructors that you may define.)

First executable statement in constructor

When **super(parameters)** is used to call the superclass constructor, it must always be the first executable statement in the constructor.

Whenever you call the constructor of a class to instantiate an object, if your constructor doesn't have a call to **super** as the first executable statement in the constructor, the call to the *noarg* constructor in the superclass is made automatically.

In other words, in order to construct an object of a class, it is necessary to first construct that part of the object attributable to the superclass. That normally happens automatically, making use of the superclass constructor that doesn't take any parameters.

Calling a parameterized constructor

If you want to use a version of the superclass constructor that takes parameters, you can make your own call to **super(parameters)** as the first executable statement in your constructor *(as was done in this program).*

Accessing a superclass member variable

The second use of the **super** keyword in the program shown in Listing 4 (p. 9) uses the keyword to bypass an instance variable named **data** in the class named **Super3** , to access and display the value of an instance variable named **data** in the superclass named **SuperClass** .

Note that in that same section of code, the **this** keyword is used to bypass a local variable named **data** in order to display the value of an instance variable named **data** in the class named **Super3** .

Similarly, a statement without the use of either **this** or **super** is used to display the value of a local variable named **data** .

To disambiguate

Therefore, as stated earlier, the program uses **this** and **super** to disambiguate a local variable, an instance variable of the subclass, and an instance variable of the superclass, where all three variables have the same name.

Accessing overridden superclass method

As mentioned earlier (p. 9) , if your method *overrides* a method in its superclass, you can use the keyword **super** to call the overridden version in the superclass, possibly completely bypassing the overridden version in the subclass.

The program named Super4

This is illustrated by the program in Listing 5 (p. 12) . This program contains an overridden version of a superclass method named **meth** . The subclass version uses the value of an incoming parameter to decide whether to call the superclass version and then to call some of its own code, or to execute its own code exclusively.

Listing 5: The program named Super4.

```
    /*File Super4.java
Copyright 2002, R.G.Baldwin
Illustrates calling the superclass
version of an overridden method from
```

```
code in the subclass version.

Tested using JDK 1.4.0 under Win 2000.

The output from this program is:

In main
Entering overridden method in subclass
Incoming parameter is false
Subclass version only is called
Back in or still in subclass version
Goodbye from subclass version

Entering overridden method in subclass
Incoming parameter is true
SuperClass method called
Back in or still in subclass version
Goodbye from subclass version

Back in main
************************************/
class SuperClass{
  //Following method is overridden in
  // the subclass.
  void meth(boolean par){
    System.out.println(
       "Incoming parameter is " + par);
    System.out.println(
          "SuperClass method called");
  }//end meth
}//end SuperClass class definition
//===================================//

class Super4 extends SuperClass{
  //Following method overrides method
  // in the superclass
  void meth(boolean par){
    System.out.println(
          "Entering overridden method "
                      + "in subclass");
    //Decide whether to call
    // superclass version
    if(par)
      //Call superclass version
      super.meth(par);
    else{
      //Don't call superclass version
      System.out.println(
       "Incoming parameter is " + par);
      System.out.println(
            "Subclass version only is "
```

```
                        + "called");
    }//end else
    //Execute some additional code
    System.out.println(
        "Back in or still in subclass "
                        + "version");
    System.out.println(
      "Goodbye from subclass version");
    System.out.println();//blank line

  }//end overridden meth()
  //---------------------------------//

  public static void main(
                       String[] args){
    //instantiate an object of
    // this type
    Super4 obj = new Super4();
    System.out.println("In main");
    //Call overridden version of
    // method
    obj.meth(false);
    //Call superclass version of
    // method
    obj.meth(true);
    System.out.println("Back in main");
  }//end main method
}//End Super4 class definition.
```

Only one statement contains super

The **super** keyword is used in only one statement in the program in Listing 5 (p. 12) . That statement appears in the subclass version of an overridden method, and is as follows:

```
super.meth(par);
```

This statement is inside the body of an **if** statement. If the value of **par** is true, then this statement is executed, causing the superclass version of the method named **meth** to be executed *(passing the value of* **par** *as a parameter to the superclass method).* When the method returns, the remaining code in the subclass version of the method is executed.

If the value of **par** is false, the above statement is bypassed, and the superclass version of the method doesn't get executed. In this case, only the code in the subclass version is executed.

5 Summary

I have discussed and illustrated the use of the **this** keyword in the following common situations:

- To bypass local variables or parameters that hide member variables having the same name, in order to access the member variable.
- To make it possible for one overloaded constructor to call another overloaded constructor in the same class.
- To pass a reference to the current object to a method belonging to a different object *(as in implementing callbacks, for example).*

I have also discussed and illustrated the use of the **super** keyword in the following situations:

- To bypass the overridden version of a method in a subclass and execute the version in the superclass.
- To bypass a member variable in a subclass in order to access a member variable having the same name in a superclass.
- To cause a constructor in a subclass to call a parameterized constructor in the immediate superclass.

6 What's next?

The next module in this collection will teach you how to use exception handling in Java.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: The this and super Keywords
- File: Java1628.htm
- Published: August 8, 2012
- Revised: –

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.

-end-