Java OOP: Static Members*

R.G. (Dick) Baldwin

Abstract

According to Baldwin, static members can be useful in some situations, but the overuse of static members can lead to problems similar to those experienced in languages that support global variables and global functions.

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

*Version 1.1: Jul 28, 2012 1:47 pm +0000
†http://creativecommons.org/licenses/by/3.0/

2.1.2 Listings

3 Preview

Static members

There is another aspect of OOP in Java that I have avoided up to this point in the discussion: ***static*** variables and ***static*** methods.

Tends to complicate ...

I have avoided this topic because, while not particularly difficult, the existence of **static** members tends to break up the simple structures that I have discussed in previous modules in this collection.

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

Avoid overuse of static members

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.

When to use static members

I will discuss the use of **static** members in this module, and will provide some guidelines for their use.

The class named Class

I will also introduce the class named **Class** and discuss how it enters into the use of **static** variables and methods.

Instance members versus class members

I will describe the differences between *instance* members and *class* members with particular emphasis being placed on their accessibility.

Three kinds of objects

From a conceptual viewpoint, there are at least three kinds of objects involved in a Java program:

- Ordinary objects
- Array objects
- Class objects

Ordinary objects

All *(or at least most)* of the discussion up to this point in the collection deals with what I have referred to in the above list as *ordinary objects* .

These are the objects that you instantiate in you code by applying the **new** operator to a constructor for a class in order to create a new instance *(object)* of that class. *(There are also a couple of other ways to create ordinary objects, but I'm not going to get into that at this time.)*

Array objects

I haven't discussed *array objects* thus far in this collection. *(I will discuss them in a future module.)*

Suffice it for now to say that array objects are objects whose purpose is to encapsulate a one-dimensional array structure that can contain either primitive values, or references to other objects *(including other array objects).*

I will discuss Class objects in this module.

4 Discussion and sample code

Class objects

Let me emphasize at the beginning that the following discussion is ***conceptual*** in nature. In this discussion, I will describe how the Java system behaves, not necessarily how it is implemented. In other words, however it is implemented, it behaves as though it is implemented in the manner described below.

The class named Class

There is a class whose name is **Class** . The purpose of this class is to encapsulate information about some other class *(actually, it can also be used to encapsulate information about primitive types as well as class types).*

Here is part of what Sun has to say about this class:

"Instances of the class ***Class*** *represent classes and interfaces in a running Java application. ...*

Class *has no public constructor. Instead* ***Class*** *objects are constructed automatically by the Java Virtual Machine as classes are loaded ..."*

What does this mean?

As a practical matter, when one or more objects are instantiated from a given class, an extra object of the **Class** class is also instantiated automatically. This object contains information about the class from which the objects were instantiated. *(Note that it is also possible to cause a* ***Class*** *object that describes a specific class to be created in the absence of objects of that class, but that is a topic that will be reserved for more advanced modules.)*

A real-world analogy

Here is an attempt to describe a real-world analogy. Remember that a class definition contains the blueprint for objects instantiated from that class.

A certain large construction company is in the business of building condominium projects. This contractor builds condos of many different sizes, types, and price ranges. However, each different condo project contains condos of only two or three different types or price ranges.

A library of blueprints

There is a large library of blueprints at the contractor's central office. This library contains blueprints for all of the different types of condos that the contractor has built or is building. *(This library is analogous to the class libraries available to the Java programmer.)*

A subset from the blueprint library

When a condo project begins, the contractor delivers copies of several sets of blueprints to the construction site. The blueprints delivered to that site describe only the types of condos being constructed on that site.

Condo is analogous to an object

Each condo unit is analogous to an *ordinary Java object* .

Each set of blueprints delivered to the construction site is roughly analogous to an *object of the class named* ***Class*** . In other words, each set of blueprints describes one or more condo units constructed from that set of blueprints.

When construction is complete

When the construction project is complete, the contractor delivers a set of blueprints for each type of condo unit to the management firm that has been hired to manage the condo complex. Each set of blueprints

continues to be analogous to an object of the class named **Class** . The blueprints remain at the site of the condo units.

RTTI

Thus, information regarding the construction, wiring, plumbing, air conditioning, etc., for each condo unit *(object)* continues to be available at the site even after the construction has been completed. *(This is somewhat analogous to something called runtime type information and often abbreviated as RTTI. A Class object contains RTTI for objects instantiated from that class.)*

What are those analogies again?

In the above scenario, each condo unit is *(roughly)* analogous to an object instantiated from a specific class *(set of blueprints)*.

Each set of blueprints remaining onsite after construction is complete is roughly analogous to a **Class** object that describes the characteristics of one or more condo units.

What do you care?

Until you get involved in such advanced topics as *reflection* and *introspection* , you don't usually have much involvement or much interest in **Class** objects. They are created automatically, and are primarily used by the Java virtual machine during runtime to help it do the things that it needs to do.

An exception to that rule

However, there is one area where you will be interested in the use of these **Class** objects from early on. You will be interested whenever variables or methods in the class definition are declared to be **static** .

Class variables and class methods

According to the current jargon, declaring a member variable to be **static** causes it to be a *class variable* . *(Note that local variables cannot be declared* ***static*** *. Only member variables can be declared* ***static*** *.)* Similarly, declaring a method to be **static** causes it to be a *class method.*

Instance variables and methods

On the other hand, according to the current jargon, not declaring a variable to be **static** causes it to be an *instance variable* , and not declaring a method to be **static** causes it to be an *instance method* .

In general, we can refer to them as *class members* and *instance members* .

What is the difference?

Here are some of the differences between *class* and *instance* members insofar as this discussion is concerned.

How many copies of member variables exist?

Every object instantiated from a given class has its own copy of each *instance variable* defined in the class. *(Instance variables are not shared among objects.)* However, every object instantiated from a given class shares the same copy of each *class variable* defined in the class. *(It is as though the class variable belongs to the single* ***Class*** *object and not to the individual objects instantiated from that class.)*

Access to an instance variable

Every object has its own copy of each instance variable *(the object owns the instance variable)*. Therefore, the only way that you can access an instance variable is to use that object's reference to send a message to the object requesting access to the variable *(even then, you may not be given access, depending on access modifiers)*.

Why call it an instance variable?

According to the current jargon, ***an object is an instance of a class*** . *(I probably told you that somewhere before in this collection.)* Each object has its own copy of each non-static variable. Hence, they are often called instance variables. *(Every instance of the class owns one and they are not implicitly shared among instances.)*

Access to a class variable

You can also send a message to an object requesting access to a class variable that the object shares with other objects instantiated from the same class. *(Again, you may or may not gain access, depending the access modifiers)*.

Access using the Class object

More importantly, you can also access a class variable without a requirement to go through an object instantiated from the class. *(In fact, a class variable can be accessed in the total absence of objects of that class.)* *(Remember, this discussion is conceptual in nature, and may not represent an actual implementation.)*

Assuming that a class variable is otherwise accessible, you can access the class variable by sending an access request message to the **Class** object to which the variable belongs.

One way to think of this

To help you keep track of things in a message-passing sense, you can pretend that there is a global reference variable whose name is the same as the name of a class.

This *(hypothetical)* reference variable contains a reference to the **Class** object that owns the class variable. Using standard Java message-passing syntax, you can access the class variable by joining the name of the reference variable to the name of the class variable with a period. Example syntax is shown below:

```
ReferenceVariableName.ClassVariableName
```

As a result of the hypothetical substitution process that I described above, this is equivalent to the following:

```
ClassName.ClassVariableName
```

We will see an example of this in the sample program that I will discuss later.

Be careful with this thought process

While this thought process may be useful when thinking about ***static*** *variables and methods, I want to point out, that the thought process breaks down very quickly when dealing with* ***Class*** *objects in a deeper sense.*

For example, when calling the ***getName*** *method on a* ***Class*** *object, an actual reference of type* ***Class*** *is required to access the members of the* ***Class*** *object. The name of the class will not suffice.*

If this discussion of a global reference variable whose name matches the name of the class is confusing to you, just forget it. Simply remember that you can access class variables by joining the name of the class to the name of the class variable using a period as the joining operator.

Characteristics of class methods

I'm not going to talk very much about instance methods and class methods in this module. However, there are a couple of characteristics of class methods that deserve a brief discussion in this context.

Cannot access instance members

First, the code in a class method has direct access only to other **static** members of the class. *(A class method does not have direct access to instance variables or instance methods of the class.)* This is sort of like saying that a class method has access to the methods and variables belonging to the **Class** object, but does not have access to the methods and variables belonging to the *ordinary objects* instantiated from the class described by the **Class** object.

Once again, be careful

Once again, this thinking breaks down very quickly once you get beyond ***static*** *members. A* ***Class*** *object also has instance methods, such as* ***getName*** *, which can only be accessed using an actual reference to the* ***Class*** *object.*

Now you are probably beginning to understand why I deferred this discussion until after I finished discussing the easy stuff.

No object required

Another important characteristic is that a class method can be accessed without a requirement for an object of the class to exist.

As with class variables, class methods can be accessed by joining the name of the class to the name of the method with a period.

I will illustrate much of this with a sample program named **MyClass01** .

Discuss in fragments

I will discuss the program in fragments. You will find a complete listing of the program in Listing 13 (p. 16) near the end of the module.

Listing 1 (p. 6) shows the beginning of the class definition.

Listing 1: Beginning of the class named MyClass01.

```
class MyClass01{
static Date v1 = new Date();
Date v2 = new Date();
```

Two member variables

The code in Listing 1 (p. 6) declares two member variables, named **v1** and **v2** , and initializes each of those variables with a reference to a new object of the **Date** class. *(When instantiated using the constructor with no arguments, the new* ***Date*** *object encapsulates the current date and time from the system clock.)*

Note the static keyword

The important thing to note here is the use of the **static** keyword when declaring the variable named **v1** . This causes **v1** to be a *class variable* , exhibiting the characteristics of class variables described earlier.

An instance variable

On the other hand, the variable named **v2** is not declared **static** . This causes it to be an *instance variable* , as described above.

The main method is a class method

Listing 2 (p. 6) shows the signature for the **main** method.

Listing 2: Signature of the main method.

```
public static void main(String[] args){
```

The important thing to note here is that the **main** method is declared **static** . That causes it to be a *class method* .

As a result, the **main** method can be called without a requirement for an object of the class to exist. *(Also, the main method has direct access only to other* ***static*** *members.)*

How a Java application starts running

In fact, that is how the Java Virtual Machine starts an application running.

First the JVM finds the specified file having an extension of ***.class.*** Then it examines that file to see if it has a **main** method with the correct signature. If not, an error occurs.

If the JVM finds a **main** method with the correct signature, it calls that method without instantiating an object of the class. That is how the Java Virtual Machine causes a Java application to start running.

A side note regarding applets

For those of you who are familiar with Java applets, you should know that this is not the case for an applet. An applet does not use a ***main*** *method. When an applet is started, an object of the controlling class is instantiated by the browser, by the* ***appletviewer*** *program, or by whatever program is being used to control the execution of the applet.*

A poor programming technique

Basically, this entire sample program is coded inside the **main** method. As a practical manner, this is a very poor programming technique, but it works well for this example.

Display some text

The code in Listing 3 (p. 6) , which is the first executable statement in the **main** method, causes the words **Static variable** to appear on the computer screen. I will come back and discuss the details of this and similar statements later in the module.

Listing 3: Display some text.

```
System.out.println("Static variable");
```

Display date information

Continuing with the code in the **main** method, the code in Listing 4 (p. 7) causes the current contents of the **Date** object referred to by the contents of the class variable named **v1** to be displayed on the computer screen.

Listing 4: Display date information.

```
System.out.println(MyClass01.v1);
```

No object required

For the moment, concentrate on the text inside the parentheses in the statement in Listing 4 (p. 7) .

Because the variable named **v1** is a class variable, it's value is accessed by joining the name of the class to the name of the variable with a period.

What was the output?

I will discuss the remaining portion of statements of this sort later. For now, just be aware that the code in Listing 4 (p. 7) caused the output shown in Figure 1 (p. 7) to be displayed on my computer screen when I ran the program.

Output date and time.

```
Mon Sep 17 09:52:27 CDT 2001
```

Figure 1: Output date and time.

Displays date and time

Obviously, the date and time displayed will depend on when you run the program. As you can see, I first wrote this module and ran this program in 2001.

Pay particular attention to the seconds portion of the time. I will refer back to this later.

A five-second delay

The code in Listing 5 (p. 7) *(still in the* ***main*** *method)* causes the main thread of the program to go to sleep for five seconds. Don't worry about it if you don't understand this code. The only reason that I included it in the program was to force a five-second delay in the execution of the program.

Listing 5: A five-second delay.

```
 try{
Thread.currentThread().sleep(5000);
}catch(InterruptedException e){}
```

Instantiate a new object

Having caused the program to sleep for five seconds, the code in Listing 6 (p. 8) instantiates a new object of the class named **MyClass01** . This code stores the new object's reference in the reference variable named **ref1** .

Listing 6: Instantiate a new object .

```
MyClass01 ref1 = new MyClass01();
```

A new Date object also

Recall from Listing 1 (p. 6) above that the class declares an instance variable named **v2** of the type **Date** .

When the new object is instantiated by the code in Listing 6 (p. 8) , a new **Date** object is also instantiated. A reference to that object is stored in the instance variable named **v2** . *(In other words, the new object of the class* ***MyClass01*** *owns a reference to a new object of the class* ***Date*** *. That reference is stored in an instance variable named* ***v2*** *in the new* ***MyClass01*** *object.)*

Display the new Date object

The code in Listing 7 (p. 8) causes a textual representation of the new **Date** object referred to by the reference variable named **v2** belonging to the object referred to by the reference variable named **ref1** , to be displayed on the standard output device.

Listing 7: Display the new Date object .

```
System.out.println(ref1.v2);
```

Five seconds later

This code caused the date and time shown in Figure 2 (p. 8) to appear on the computer screen when I ran the program:

Five seconds later.

```
Mon Sep 17 09:52:32 CDT 2001
```

Figure 2: Five seconds later.

The date and time shown in Figure 2 (p. 8) is five seconds later than the time reflected in the **Date** object referred to by the *class variable* named **v1** *(see Figure 1 (p. 7))* . That time was displayed by the code in Listing 4 (p. 7) earlier.

So, what does this mean?

It means that the **Date** object referred to by the **static** reference variable named **v1** was created five seconds earlier than the **Date** object referred to by the instance variable named **v2** .

When is a class variable created?

I can't tell you precisely when a class variable comes into existence. All I can say is that the virtual machine brings it into existence as soon as it is needed.

My guess is that it comes into existence at the first mention *(in the program)* of the class to which it belongs.

When is an instance variable created?

An instance variable doesn't come into existence until the object to which it belongs is created. *(An instance variable cannot exist until the object to which it belongs exists.)*

If the instance variable is initialized with a reference to a new object *(such as a new* ***Date*** *object in this sample program)*, that new object comes into existence when the object to which it belongs comes into existence.

A five-second delay

In this program, I purposely inserted a five-second delay between the first mention of the class named **MyClass01** in Listing 4 (p. 7) , and the instantiation of the object of the class named **MyClass01** in Listing 6 (p. 8) .

As a result, the **Date** object referred to by the instance variable named **v2** was created about five seconds later than the **Date** object referred to by the class variable named **v1** .

This is reflected in the date and time values displayed and discussed above.

Accessing class variable via an object

While it is possible to access a class variable using the name of the class joined to the name of the variable, it is also possible to access a class variable using a reference to any object instantiated from the class.

(As mentioned earlier, if two or more objects are instantiated from the same class, they share the same class variable.)

The code in parentheses in Listing 8 (p. 9) uses the reference variable named **ref1** to access the class variable named **v1** , and to cause the contents of the **Date** object referred to by the class variable to be displayed.

Listing 8: Accessing class variable via an object.

```
System.out.println(ref1.v1);
```

The output

This caused the date and time shown in Figure 3 (p. 9) to be displayed on my computer screen.

Same date and time as before.

```
Mon Sep 17 09:52:27 CDT 2001
```

Figure 3: Same date and time as before.

Same date and time as before

As you have probably already surmised, this is the same date and time shown earlier in Figure 1 (p. 7) . This is because the code in Listing 8 (p. 9) refers to the same class variable as the code in Listing 4 (p. 7) .

Nothing has caused the contents of that class variable to change, so both Figure 1 (p. 7) and Figure 3 (p. 9) display the contents of the same **Date** object.

(Only one class variable exists and it doesn't matter how you access it. Either way, you gain access to the same ***Date*** *object whose reference is stored in the class variable. Thus, the same date and time is shown in both cases.)*

Another new object

If you examine the code in Listing 13 (p. 16) near the end of the program, you will see that an additional five-second delay is introduced at this point in the program.

Following that delay, the code in Listing 9 (p. 10) instantiates another new object of the class named **MyClass01** , and stores the object's reference in a new reference variable named **ref2** .

(The object referred to by ***ref1*** *is a different object than the object referred to by* ***ref2*** *. Each object has its own instance variable named* ***v2*** *, and in this case, each instance variable is initialized to instantiate and refer to a new* ***Date*** *object when the new* ***MyClass01*** *object is instantiated.)*

Listing 9: Another new object .

```
MyClass01 ref2 = new MyClass01();
```

Display the date and time

Then, the code in Listing 10 (p. 10) causes the contents of the **Date** object referred to by the instance variable named **v2** in the second object of the class named **MyClass01** to be displayed.

Listing 10: Display the date and time .

```
System.out.println(ref2.v2);
```

This caused the output shown in Figure 4 (p. 10) to be displayed on my computer screen when I ran the program.

(Once again, you will get different results if you compile and run the program because the date and time shown is the date and time that you run the program.)

A new date and time.

```
Mon Sep 17 09:52:37 CDT 2001
```

Figure 4: A new date and time.

Five seconds later

As you have probably figured out by now, the time encapsulated in this **Date** object is five seconds later than the time encapsulated in the **Date** object displayed in Figure 2 (p. 8) . This is because the program was put to sleep for five seconds between the instantiation of the two objects referred to by **ref1** and **ref2** .

Every object has one

Every object instantiated from a given class has its own copy of each instance variable declared in the class definition. There is no sharing of instance variables among objects.

Each instance variable comes into existence when the object to which it belongs comes into existence, and ceases to exist when the object to which it belongs ceases to exist.

Eligible for garbage collection

If the instance variables are reference variables holding references to other objects, as is the case here, and if there are no other reference variables holding references to those same objects, the secondary objects cease to exist when the primary objects cease to exist.

Technically, the objects may not actually cease to exist. Technically they become eligible for garbage collection, which means that the memory that they occupy becomes eligible for reuse. However, as a practical matter, they cease to exist insofar as the program is concerned because they are no longer accessible.

A five-second difference in the time of creation

Since the two objects referred to by **ref1** and **ref2** came into existence with a five-second delay, the **Date** objects belonging to those two object reflect a five-second difference in the time encapsulated in the objects.

Only one copy of class variable exists

Also remember that if a variable is a class variable, only one copy of the variable exists, and all objects instantiated from the class share that one copy.

This is illustrated by the code in Listing 11 (p. 11) , which uses the reference to the second object instantiated from the class named **MyClass01** , to cause the contents of the class variable named **v1** to be displayed.

Listing 11: Display date information.

```
    System.out.println(ref2.v1);
}//end main
```

The output produced by the code in Listing 11 (p. 11) is shown in Figure 5 (p. 11) .

Same date and time as before.

```
Mon Sep 17 09:52:27 CDT 2001
```

Figure 5: Same date and time as before.

Same output as before

As you can see, this is the same as the output shown in Figure 1 (p. 7) and Figure 3 (p. 9) earlier.

Accessing the same physical class variable

Since only one class variable named **v1** exists, and all objects instantiated from the class named **MyClass01** share that single copy, it doesn't matter whether you access the class variable using the name

of the class, or access it using a reference to either of the objects instantiated from the class. In all three cases, you are accessing the same physical class variable.

Since nothing was done to cause the contents of the class variable to change after it came into existence and was initialized, Figure 1 (p. 7) , Figure 3 (p. 9) , and Figure 5 (p. 11) are simply three different displays of the date and time encapsulated in the same **Date** object whose reference is stored in the class variable.

Let's revisit System.out.println...

Now, I want to revisit the statement originally shown in Listing 8 (p. 9) and repeated in Listing 12 (p. 12) for viewing convenience.

Listing 12: Revisiting System.out.println.

```
System.out.println(ref1.v1);
```

Java programmer wanted

I sometimes tell my students that if I were out in industry interviewing prospective Java programmers, my first question would be to ask the prospective employee to tell me everything that she knows about the statement in Listing 12 (p. 12) .

Covers a lot of Java OOP technology

This is not because there is a great demand for the use of this statement in real-world problems. *(In fact, in a GUI-driven software product world, there is probably very little demand for the use of this statement.)* Rather, it is because a lot of Java object-oriented technology is embodied in this single statement.

In that scenario, I would expect to receive a verbal dissertation of fifteen to twenty minutes in length to cover all the important points.

The short version

Let me give you the short version. There is a class named **System** . The **System** class declares three **static** *(class)* variables having the following types, names, and modifiers:

- public static final PrintStream **out**
- public static final InputStream **in**
- public static final PrintStream **err**

(Note that these class variables are also declared ***final*** *, causing them to behave as constants.)*

A **ccess the out variable without an object**

Because **out** is a class variable, **System.out** returns the contents of the class variable named **out** *(an object of the* ***System*** *class is not required in order to access a class variable of the* ***System*** *class).*

In general, *(ignoring the possibility of subclasses and interfaces)* because **out** is a reference variable of type **PrintStream** , the returned value must either be **null** *(no object reference)* or a reference to a valid **PrintStream** object.

Object of the PrintStream class

When the Java Virtual Machine starts an application running, it automatically instantiates an object of the **PrintStream** class and connects it to the **standard output device** . *(By default, the standard output device is typically the computer screen, but it can be redirected at the operating system level to be some other device. The following discussion assumes that the screen is the standard output device.)*

Assign object's reference to out variable

When the **PrintStream** object is instantiated by the virtual machine, the object's reference is assigned to the class variable of the **System** class named **out** . *(Because the variable named* ***out*** *is final, the contents of the variable cannot be modified later.)*

Reference to a PrintStream object

Therefore, the expression **System.out** returns a reference to the **PrintStream** object, which is connected to the standard output device.

Many instance methods

An object of the **PrintStream** class contains many instance methods. This includes numerous overloaded versions of a method named **println** . The signature of one of those overloaded **versions of the println method follows** :

```
public void println(Object x)
```

Textual representation of an object

The purpose of this overloaded version of the **println** method is to:

- Create a textual representation of the object referred to by the incoming parameter of type **Object** *(because **Object** is a totally generic type, this version of the **println** method can accept an incoming parameter that is a reference to any type of object)*
- Send that textual representation to the output device

In general...

A new **PrintStream** object can be connected to a variety of output devices when it is instantiated. However, in the special case of the **PrintStream** object instantiated by the virtual machine when the program starts, whose reference is stored in the class variable named **out** of the **System** class, the purpose of the object is to provide a display path to the standard output device.

Our old friend, the toString method

To accomplish this, the code in the version of the **println** method shown above (p. 13) calls the **toString** method on the incoming reference. *(I discussed the **toString** method in detail in earlier modules in this collection.)* The **toString** method may, or may not, have been overridden in the definition of the class from which the object was instantiated, or in some superclass of the class from which the object was instantiated.

Default version of toString

If not overridden, the default version of the **toString** method defined in the **Object** class is used to produce a textual representation of the object. As we learned in an earlier module, that textual representation looks something like the following:

```
ClassName@HexHashCode
```

Overridden version of toString method

If the class from which the object was instantiated *(or some superclass of that class)* contains an overridden version of the **toString** method, runtime polymorphism kicks in and the overridden version of the method is executed to produce the textual representation of the object.

The Date class overrides toString

In the case of this sample program, the object was instantiated from the **Date** class. The **Date** class does override the **toString** method.

When the overridden **toString** method is called on a **Date** object's reference, the **String** returned by the method looks something like that shown in Figure 6 (p. 14) .

Output from overridden toString method in Date class.

```
Mon Sep 17 09:52:27 CDT 2001
```

Figure 6: Output from overridden toString method in Date class.

You will recall that this is the output that was produced by the code shown in Listing 8 (p. 9) and Listing 12 (p. 12) .

More than you ever wanted to know ...

And that is probably more than you ever wanted to know about the expression **System.out.println...** .

It is also probably more than you ever wanted to know about class variables, class methods, instance variables, and instance methods.

Some cautions

Before leaving this topic, I do want to express some cautions. Basically, I want to suggest that you use ***static*** members very sparingly, if at all.

Static variables

To begin with, don't ever use **static** variables without declaring them **final** unless you understand exactly why you are declaring them **static** .

*(**static** **final** variables, or constants, are often very appropriate. See the fields in the **Color** class for example.)*

I can only think of only a very a few situations in which the use of a non-final **static** variable might be appropriate.

(One appropriate use might be to count the number of objects instantiated from a specific class.)

Static methods

Don't declare methods **static** if there is any requirement for the method to remember anything from one call to the next.

There are many appropriate uses for **static** methods, but in most cases, the purpose of the method will be to completely perform some action with no requirement to remember anything from that call to the next.

The method should probably also be self-contained. By this I mean that all information that the method needs to do its job should either come from incoming parameters or from **final** **static** member variables *(constants)*. The method probably should not depend on the values stored in non-final **static** member variables, which are subject to change over time.

*(A **static** method only has access to other **static** members of the class, so it cannot depend on instance variables defined in the class.)*

An appropriate example of a **static** method is the **sqrt** method of the **Math** class. This method computes and *"Returns the correctly rounded positive square root of a double"* where the double value is provided as a parameter to the method. Each time the method is called, it completes its task and doesn't attempt to save any values from that call to the next. Furthermore, it gets all the information that it needs to do its job from an incoming parameter.

5 Summary

Added complexity

The existence of **static** members tends to break up the simple OOP structures that I have discussed in previous modules in this collection.

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 class named Class

I discussed the class named **Class** and how a *conceptual* object of type **Class** exists in memory following a reference to a specific class in the program code.

The **Class** object represents the referenced class in memory, and contains the **static** variables and **static** methods belonging to that class. *(It contains some other information as well, such as the name of the superclass.)*

Class members and instance members

Class variables and *class methods* are declared **static** *(declaring a member **static** in the class definition causes to be called a class member)* .

Instance variables and instance methods are not declared **static** .

Each object has its own copy ...

Every object instantiated from a given class has its own copy of each instance variable declared in the class definition. *(Instance variables are not shared among objects.)*

Every object instantiated from a given class acts like it has its own copy of every instance method declared in the class definition. *(Although instance methods are actually shared among objects in order to reduce the amount of memory required, they are shared in such a way that they don't appear to be shared.)*

Every object shares ...

Every object instantiated from a given class shares the same single copy of each class variable declared in the class definition. Similarly, every object instantiated from a given class shares the same copy of each class method.

Accessing an instance member

An instance variable or an instance method can only be accessed by using a reference to the object that owns it. Even then, it may or may not be accessible depending on the access modifier assigned by the programmer.

Accessing a class member

The single shared copy of a class variable or a class method can be accessed in either of two ways:

- Via a reference to any object instantiated from the class.
- By simply joining the name of the class to the name of the class variable or the class method.

Again, the variable or method may or may not be accessible, depending on the access modifiers assigned by the programmer.

When to use class variables

It is very often appropriate to use **final** **static** variables, as constants in your programs. It is rarely, if ever, appropriate to use non-final **static** variables in your programs. The use of non-final **static** variables should definitely be minimized.

When to use static methods

It is often appropriate to use **static** methods in your programs, provided there is no requirement for the method to remember anything from one call to the next. **Static** methods should be self-contained.

6 What's next?

The next module in this collection will address the special case of Array Objects.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Static Members
- File: Java1620.htm
- Published: April 24, 2002
- Revised: July 28, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

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

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

8 Complete program listing

A complete listing of the sample program is shown in Listing 13 (p. 16) .

Listing 13: Complete program listing.

```
    /*File MyClass01.java
Copyright 2002, R.G.Baldwin

This program illustrates static
members of a class. Output is:

Static variable
Mon Sep 17 09:52:27 CDT 2001

Instance variable
Mon Sep 17 09:52:32 CDT 2001
Static variable
Mon Sep 17 09:52:27 CDT 2001


Instance variable
Mon Sep 17 09:52:37 CDT 2001
Static variable
Mon Sep 17 09:52:27 CDT 2001
```

```
************************************/
import java.util.Date;
class MyClass01{
 static Date v1 = new Date();
 Date v2 = new Date();

 public static void main(
 String[] args){
 //Display static variable
 System.out.println(
 "Static variable");
 System.out.println(MyClass01.v1);

 //Delay for five seconds
 try{
 Thread.currentThread().sleep(5000);
 }catch(InterruptedException e){}

 //Instantiate an object and
 // display instance variable
 MyClass01 ref1 = new MyClass01();
 System.out.println();//blank line
 System.out.println(
 "Instance variable");
 System.out.println(ref1.v2);

 //Now, display the static
 // variable using object reference
 System.out.println(
 "Static variable");
 System.out.println(ref1.v1);

 System.out.println();//blank line

 //Delay for five seconds
 try{
 Thread.currentThread().sleep(5000);
 }catch(InterruptedException e){}

 //Instantiate another object
 MyClass01 ref2 = new MyClass01();
 System.out.println();//blank line
 System.out.println(
 "Instance variable");
 System.out.println(ref2.v2);

 //Now, display the same static
 // variable using object reference
 System.out.println(
 "Static variable");
 System.out.println(ref2.v1);
```

```
 }//end main
}//end class MyClass01
```

-end-