ADVANCED JAVA LANGUAGE FEATURES

Static classes and inner classes in Java

Learn how to use the four types of nested classes in your Java code

Nested classes are classes that are declared as members of other classes or scopes. Nesting classes is one way to better organize your code. For example, say you have a non-nested class (also known as a *top-level class*) that stores objects in a resizable array, followed by an iterator class that returns each object. Rather than pollute the top-level class's namespace, you could declare the iterator class as a member of the resizable array collection class. This works because the two are closely related.

In Java, nested classes are categorized as either *static member classes* or *inner classes*. Inner classes are non-static member classes, local classes, or anonymous classes. In this tutorial you'll learn how to work with static member classes and the three types of inner classes in your Java code.

Avoid memory leaks in nested classes

Also see the Java tip associated with this tutorial, where you'll learn why nested classes are vulnerable to memory leaks.

Static classes in Java

In my **Java 101** tutorial Classes and objects in Java, you learned how to declare static fields and static methods as members of a class. In Class and object initialization in Java, you learned how to declare static initializers as members of a class. Now you'll learn how to

declare *static classes*. Formally known as *static member classes*, these are nested classes that you declare at the same level as these other static entities, using the `static` keyword. Here's an example of a static member class declaration:

```
class C
{
   static int f;

   static void m() {}

   static
   {
      f = 2;
   }

   static class D
   {
      // members
   }
}
```

This example introduces top-level class `C` with static field `f`, static method `m()`, a static initializer, and static member class `D`. Notice that `D` is a member of `C`. The static field `f`, static method `m()`, and the static initializer are also members of `C`. Since all of these elements belong to class `C`, it is known as the *enclosing class*. Class `D` is known as the *enclosed class*.

Enclosure and access rules

Although it is enclosed, a static member class cannot access the enclosing class's instance fields and invoke its instance methods. However, it can access the enclosing class's static fields and invoke its static methods, even those members that are declared `private`. To demonstrate, Listing 1 declares an `EnclosingClass` with a nested `SMClass`.

Listing 1. Declaring a static member class (EnclosingClass.java, version 1)

```
class EnclosingClass
{
   private static String s;

   private static void m1()
   {
      System.out.println(s);
   }

   static void m2()
   {
      SMClass.accessEnclosingClass();
   }

   static class SMClass
   {
      static void accessEnclosingClass()
      {
         s = "Called from SMClass's accessEnclosingClass() method";
         m1();
      }

      void accessEnclosingClass2()
      {
         m2();
      }
   }
}
```

Listing 1 declares a top-level class named `EnclosingClass` with class field `s`, class methods `m1()` and `m2()`, and static member class `SMClass`. `SMClass` declares class method `accessEnclosingClass()` and instance method `accessEnclosingClass2()`. Note the following:

- `m2()`'s invocation of `SMClass`'s `accessEnclosingClass()` method requires the `SMClass.` prefix because `accessEnclosingClass()` is declared `static`.
- `accessEnclosingClass()` is able to access `EnclosingClass`'s `s` field and call its `m1()` method, even though both have been declared `private`.

Listing 2 presents the source code to an `SMCDemo` application class that demonstrates how to invoke `SMClass`'s `accessEnclosingClass()` method. It also demonstrates how to instantiate `SMClass` and invoke its `accessEnclosingClass2()` instance method.

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

Listing 2. Invoking a static member class's methods (SMCDemo.java)

```
public class SMCDemo
{
   public static void main(String[] args)
   {
      EnclosingClass.SMClass.accessEnclosingClass();
      EnclosingClass.SMClass smc = new EnclosingClass.SMClass();
      smc.accessEnclosingClass2();
   }
}
```

As shown in Listing 2, if you want to invoke a top-level class's method from within an enclosed class, you must prefix the enclosed class's name with the name of its enclosing class. Likewise, in order to instantiate an enclosed class you must prefix the name of that class with the name of its enclosing class. You can then invoke the instance method in the normal manner.

Compile Listings 1 and 2 as follows:

```
javac *.java
```

When you compile an enclosing class that contains a static member class, the compiler creates a class file for the static member class whose name consists of its enclosing class's name, a dollar-sign character, and the static member class's name. In this case, compiling results in `EnclosingClass$SMCClass.class` and `EnclosingClass.class`.

Run the application as follows:

```
java SMCDemo
```

You should observe the following output:

```
Called from SMClass's accessEnclosingClass() method
Called from SMClass's accessEnclosingClass() method
```

Example: Static classes and Java 2D

Java's *standard class library* is a runtime library of class files, which store compiled classes and other reference types. The library includes numerous examples of static member classes, some of which are found in the Java 2D geometric shape classes located in the `java.awt.geom` package. (You'll learn about packages in the next **Java 101** tutorial.)

The `Ellipse2D` class found in `java.awt.geom` describes an ellipse, which is defined by a framing rectangle in terms of an (x,y) upper-left corner along with width and height extents. The following code fragment shows that this class's architecture is based on `Float` and `Double` static member classes, which both subclass `Ellipse2D`:

```
public abstract class Ellipse2D extends RectangularShape
{
   public static class Float extends Ellipse2D implements Serializable
   {
      public float x, y, width, height;

      public Float()
      {
      }

      public Float(float x, float y, float w, float h)
      {
         setFrame(x, y, w, h);
      }

      public double getX()
      {
         return (double) x;
      }

      // additional instance methods
   }

   public static class Double extends Ellipse2D implements Serializable
   {
      public double x, y, width, height;

      public Double()
      {
      }

      public Double(double x, double y, double w, double h)
      {
         setFrame(x, y, w, h);
      }

      public double getX()
      {
         return x;
      }

      // additional instance methods
   }

   public boolean contains(double x, double y)
   {
```

```
        // ...
    }

    // additional instance methods shared by Float, Double, and other
    // Ellipse2D subclasses
}
```

The `Float` and `Double` classes extend `Ellipse2D`, providing floating-point and double precision floating-point `Ellipse2D` implementations. Developers use `Float` to reduce memory consumption, particularly because you might need thousands or more of these objects to construct a single 2D scene. We use `Double` when greater accuracy is required.

You cannot instantiate the abstract `Ellipse2D` class, but you can instantiate either `Float` or `Double`. You also can extend `Ellipse2D` to describe a custom shape that's based on an ellipse.

As an example, let's say you want to introduce a `Circle2D` class, which isn't present in the `java.awt.geom` package. The following code fragment shows how you would create an `Ellipse2D` object with a floating-point implementation:

```
Ellipse2D e2d = new Ellipse2D.Float(10.0f, 10.0f, 20.0f, 30.0f);
```

The next code fragment shows how you would create an `Ellipse2D` object with a double-precision floating-point implementation:

```
Ellipse2D e2d = new Ellipse2D.Double(10.0, 10.0, 20.0, 30.0);
```

You can now invoke any of the methods declared in `Float` or `Double` by invoking the method on the returned `Ellipse2D` reference (e.g., `e2d.getX()`). In the same manner, you could invoke any of the methods that are common to `Float` and `Double`, and which are declared in `Ellipse2D`. An example is:

```
e2d.contains(2.0, 3.0)
```

That completes the introduction to static member classes. Next we'll look at inner classes, which are non-static member classes, local classes, or anonymous classes. You'll learn how to work with all three inner class types.

Get the code

Download the source code for examples in this tutorial. *Created by Jeff Friesen for JavaWorld.*

Inner classes, type 1: Non-static member classes

You've learned previously in the **Java 101** series how to declare non-static (instance) fields, methods, and constructors as members of a class. You can also declare *non-static member classes*, which are nested non-static classes that you declare at the same level as instance fields, methods, and constructors. Consider this example:

```
class C
{
   int f;

   void m() {}

   C()
   {
      f = 2;
   }

   class D
   {
      // members
   }
}
```

Here, we introduce top-level class `C` with instance field `f`, instance method `m()`, a constructor, and non-static member class `D`. All of these entities are members of class `C`, which encloses them. However, unlike in the previous example, these instance entities are associated with *instances of* `C` and not with the `C` class itself.

Each instance of the non-static member class is implicitly associated with an instance of its enclosing class. The non-static member class's instance methods can call the enclosing class's instance methods and access its instance fields. To demonstrate this access, Listing 3 declares an `EnclosingClass` with a nested `NSMClass`.

Listing 3. Declare an enclosing class with a nested non-static member class (EnclosingClass.java, version 2)

```
class EnclosingClass
{
   private String s;

   private void m()
   {
      System.out.println(s);
   }

   class NSMClass
   {
      void accessEnclosingClass()
      {
         s = "Called from NSMClass's accessEnclosingClass() method";
         m();
      }
   }
}
```

Listing 3 declares a top-level class named `EnclosingClass` with instance field `s`, instance method `m()`, and non-static member class `NSMClass`. Furthermore, `NSMClass` declares instance method `accessEnclosingClass()`.

Because `accessEnclosingClass()` is non-static, `NSMClass` must be instantiated before this method can be called. This instantiation must take place via an instance of `EnclosingClass`, as shown in Listing 4.

Listing 4. NSMCDemo.java

```
public class NSMCDemo
{
   public static void main(String[] args)
   {
      EnclosingClass ec = new EnclosingClass();
      ec.new NSMClass().accessEnclosingClass();
   }
}
```

Listing 4's `main()` method first instantiates `EnclosingClass` and saves its reference in local variable `ec`. The `main()` method then uses the `EnclosingClass` reference as a prefix to the `new` operator, in order to instantiate `NSMClass`. The `NSMClass` reference is then used to call `accessEnclosingClass()`.

Should I use 'new' with a reference to the enclosing class?

Prefixing `new` with a reference to the enclosing class is rare. Instead, you will typically call an enclosed class's constructor from within a constructor or an instance method of its enclosing class.

Compile Listings 3 and 4 as follows:

```
javac *.java
```

When you compile an enclosing class that contains a non-static member class, the compiler creates a class file for the non-static member class whose name consists of its enclosing class's name, a dollar-sign character, and the non-static member class's name. In this case, compiling results in `EnclosingClass$NSMCClass.class` and `EnclosingClass.class`.

Run the application as follows:

```
java NSMCDemo
```

You should observe the following output:

```
Called from NSMClass's accessEnclosingClass() method
```

When (and how) to qualify 'this'

An enclosed class's code can obtain a reference to its enclosing-class instance by qualifying reserved word `this` with the enclosing class's name and the member access operator (`.`). For example, if code within `accessEnclosingClass()` needed to obtain a reference to its `EnclosingClass` instance, it would specify `EnclosingClass.this`. Because the compiler generates code to accomplish this task, specifying this prefix is rare.

Example: Non-static member classes in HashMap

The standard class library includes non-static member classes as well as static member classes. For this example, we'll look at the `HashMap` class, which is part of the Java Collections Framework in the `java.util` package. `HashMap`, which describes a hash table-based implementation of a map, includes several non-static member classes.

For example, the `KeySet` non-static member class describes a set-based *view* of the keys contained in the map. The following code fragment relates the enclosed `KeySet` class to its `HashMap` enclosing class:

```
public class HashMap<K,V> extends AbstractMap<K,V>
                          implements Map<K,V>, Cloneable, Serializable
{
   // various members

   final class KeySet extends AbstractSet<K>
   {
      // various members
   }

   // various members
}
```

The `<K,V>` and `<K>` syntaxes are examples of *generics*, a suite of related language features that help the compiler enforce type safety. I'll introduce generics in an upcoming **Java 101** tutorial. For now, you just need to know that these syntaxes help the compiler enforce the type of key objects that can be stored in the map and in the keyset, and the type of value objects that can be stored in the map.

`HashMap` provides a `keySet()` method that instantiates `KeySet` when necessary and returns this instance or a cached instance. Here's the complete method: