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


```
public Set<K> keySet()
{
   Set<K> ks = keySet;
   if (ks == null)
   {
      ks = new KeySet();
      keySet = ks;
   }
   return ks;
}
```

Notice that the enclosed class's (`KeySet`'s) constructor is called from within the enclosing class's (`HashSet`'s) `keySet()` instance method. This illustrates a common practice, especially because prefixing the `new` operator with an enclosing class reference is rare.

Inner classes, type 2: Local classes

It's occasionally helpful to declare a class in a block, such as a method body or sub-block. For example, you might declare a class that describes an iterator in a method that returns an instance of this class. Such classes are known as *local classes* because (as with local variables) they are local to the methods in which they are declared. Here is an example:

```
interface I
{
   // members
}

class C
{
   I m() // or even static I m()
   {
      class D implements I
      {
         // members
      }
      return new D();
   }
}
```

Top-level class `C` declares instance method `m()`, which returns an instance of local class `D`, which is declared in this method. Notice that `m()`'s return type is interface `I`, which `D` implements. The interface is necessary because giving `m()` return type `D` would result in a compiler error--`D` isn't accessible outside of `m()`'s body.

Illegal access modifiers in local class declaration

The compiler will report an error if a local class declaration contains any of the access modifiers `private`, `public`, or `protected`; or the modifier `static`.

A local class can be associated with an instance of its enclosing class, but only when used in a non-static context. Also, a local class can be declared anywhere that a local variable can be declared, and has the same scope as a local variable. It can access the surrounding scope's local variables and parameters, which must be declared `final`. Consider Listing 5.

Listing 5. Declaring a local class within an enclosing class instance method (EnclosingClass.java, version 3)

```
class EnclosingClass
{
   void m(final int x)
   {
      final int y = x * 3;
      class LClass
      {
         int m = x;
         int n = y;
      }
      LClass lc = new LClass();
      System.out.println(lc.m);
      System.out.println(lc.n);
   }
}
```

Listing 5 declares `EnclosingClass` with instance method `m()`, declaring a local class named `LClass`. `LClass` declares a pair of instance fields (`m` and `n`). When `LClass` is instantiated, the instance fields are initialized to the values of final parameter `x` and final local variable `y`, as shown in Listing 6.

Listing 6. A local class declares and initializes a pair of instance fields (LCDemo.java)

```
public class LCDemo
{
   public static void main(String[] args)
   {
      EnclosingClass ec = new EnclosingClass();
      ec.m(5);
   }
}
```

Listing 6's `main()` method first instantiates `EnclosingClass`. It then invokes `m(5)` on this instance. The called `m()` method multiplies this argument by 3, instantiates `LClass`, whose `<init>()` method assigns the argument and the tripled value to its pair of instance fields and outputs `LClass`'s instance fields. (Note that in this case the local class uses the `<init>()` method instead of a constructor to interact with its instance fields.)

Compile Listings 5 and 6 as follows:

```
javac *.java
```

When you compile a class whose method contains a local class, the compiler creates a class file for the local class whose name consists of its enclosing class's name, a dollar-sign character, a 1-based integer, and the local class's name. In this case, compiling results in `EnclosingClass$1LClass.class` and `EnclosingClass.class`.

A note about local class file name

When generating a name for a local class's class file, the compiler adds an integer to the generated name. This integer is probably generated to distinguish a local class's class file from a non-static member class's class file. If two local classes have the same name, the compiler increments the integer to avoid conflicts. Consider the following example:

```
public class EnclosingClass
{
    public void m1()
    {
       class LClass
       {
       }
    }

    public void m2()
    {
       class LClass
       {
       }
    }

    public void m3()
    {
       class LClass2
       {
       }
    }
}
```

`EnclosingClass` declares three instance methods that each declare a local class. The first two methods generate two different local classes with the same name. The compiler generates the following class files:

```
EnclosingClass$1LClass.class
EnclosingClass$1LClass2.class
EnclosingClass$2LClass.class
EnclosingClass.class
```

Run the application as follows:

```
java LCDemo
```

You should observe the following output:

```
5
15
```

Example: Using local classes in regular expressions

The standard class library includes examples of local class usage. For example, the `Matcher` class, in `java.util.regex`, provides a `results()` method that returns a stream of match results. This method declares a `MatchResultIterator` class for iterating over these results:

```
public Stream<MatchResult> results()
{
   class MatchResultIterator implements Iterator<MatchResult>
   {
      // members
   }
   return StreamSupport.
          stream(Spliterators.spliteratorUnknownSize(new MatchResultIterator(),
                                                     Spliterator.ORDERED |
                                                     Spliterator.NONNULL),
                 false);
}
```

Note the instantiation of `MatchResultIterator()` following the class declaration. Don't worry about parts of the code that you don't understand; instead, think about the usefulness in being able to declare classes in the appropriate scopes (such as a method body) to better organize your code.

Inner classes, type 3: Anonymous classes

Static member classes, non-static member classes, and local classes have names. In contrast, *anonymous classes* are unnamed nested classes. You introduce them in the context of expressions that involve the `new` operator and the name of either a base class or an interface that is implemented by the anonymous class:

```
// subclass the base class

abstract class Base
{
   // members
}

class A
{
   void m()
   {
      Base b = new Base()
               {
                 // members
               };
   }
}

// implement the interface

interface I
{
   // members
}

class B
{
   void m()
   {
      I i = new I()
            {
               // members
            };
   }
}
```

The first example demonstrates an anonymous class extending a base class. Expression `new Base()` is followed by a pair of brace characters that signify the anonymous class. The second example demonstrates an anonymous class implementing an interface. Expression `new I()` is followed by a pair of brace characters that signify the anonymous class.

Constructing anonymous class instances

An anonymous class cannot have an explicit constructor because a constructor must be named after the class and anonymous classes are unnamed. Instead, you can use an object initialization block ({}) as a constructor.

Anonymous classes are useful for expressing functionality that's passed to a method as its argument. For example, consider a method for sorting an array of integers. You want to sort the array in ascending or descending order, based on comparisons between pairs of array elements. You might duplicate the sorting code, with one version using the less than (<) operator for one order, and the other version using the greater than (>) operator for the opposite order. Alternatively, as shown below, you could design the sorting code to invoke a comparison method, then pass an object containing this method as an argument to the sorting method.

Listing 7. Using an anonymous class to pass functionality as a method argument (Comparer.java)

```
public abstract class Comparer
{
   public abstract int compare(int x, int y);
}
```

The `compare()` method is invoked with two integer array elements and returns one of three values: a negative value if `x` is less than `y`, 0 if both values are the same, and a positive value if `x` is greater than `y`. Listing 8 presents an application whose `sort()` method invokes `compare()` to perform the comparisons.

Listing 8. Sorting an array of integers with the Bubble Sort algorithm (ACDemo.java)

```
public class ACDemo
{
   public static void main(String[] args)
   {
      int[] a = { 10, 30, 5, 0, -2, 100, -9 };
      dump(a);
      sort(a, new Comparer()
                  {
                     public int compare(int x, int y)
                     {
                        return x - y;
                     }
                  });
      dump(a);
      int[] b = { 10, 30, 5, 0, -2, 100, -9 };
      sort(b, new Comparer()
                  {
                     public int compare(int x, int y)
                     {
                        return y - x;
                     }
                  });
      dump(b);
   }

   static void dump(int[] x)
   {
      for (int i = 0; i < x.length; i++)
         System.out.print(x[i] + " ");
      System.out.println();
   }

   static void sort(int[] x, Comparer c)
   {
      for (int pass = 0; pass < x.length - 1; pass++)
         for (int i = x.length - 1; i > pass; i--)
            if (c.compare(x[i], x[pass]) < 0)
            {
               int temp = x[i];
               x[i] = x[pass];
               x[pass] = temp;
            }
   }
}
```

The `main()` method reveals two calls to its companion `sort()` method, which sorts an array of integers via the Bubble Sort algorithm. Each call receives an integer array as its first argument, and a reference to an object created from an anonymous `Comparer` subclass as its second argument. The first call achieves an ascending order sort by subtracting `y` from `x`; the second call achieves a descending order sort by subtracting `x` from `y`.

Migrating from anonymous classes to lambdas

As you can see in Listing 8, passing anonymous class-based functionality can lead to verbose syntax. Starting with Java 8, you can use lambdas for more concise code. See the **Java 101** tutorial Get started with lambda expressions in Java to learn more about using lambdas in your Java code.

Compile Listings 7 and 8 as follows:

```
javac *.java
```

When you compile a class whose method contains an anonymous class, the compiler creates a class file for the anonymous class whose name consists of its enclosing class's name, a dollar-sign character, and an integer that uniquely identifies the anonymous class. In this case, compiling results in `ACDemo$1.class` and `ACDemo$2.class` in addition to `ACDemo.class`.

Run the application as follows:

```
java ACDemo
```

You should observe the following output:

```
10 30 5 0 -2 100 -9
-9 -2 0 5 10 30 100
100 30 10 5 0 -2 -9
```

Example: Using anonymous classes with an AWT event handler

Anonymous classes can be used with many packages in the standard class library. For this example, we'll use an anonymous class as an event handler in the Abstract Windowing Toolkit or Swing Windowing Toolkit. The following code fragment registers an event handler with Swing's `JButton` class, which is located in the `javax.swing` package. `JButton` describes a button that performs an action (in this case printing a message) when clicked.

```
JButton btnClose = new JButton("close");
btnClose.addActionListener(new ActionListener()
                                {
                                   public void actionPerformed(ActionEvent ae)
                                   {
                                      System.out.println("close button clicked");
                                   }
                                });
```

The first line instantiates `JButton`, passing `close` as the button label to `JButton`'s constructor. The second line registers an action listener with the button. The action listener's `actionPerformed()` method is invoked whenever the button is clicked. The object passed to `addActionListener()` is instantiated from an anonymous class that implements the `java.awt.event.ActionListener` interface.

Nested interfaces and classes
Java also lets you nest interfaces in interfaces and classes in interfaces. Check out the companion article on nested interfaces to learn about this strange but occasionally useful Java language feature.

Conclusion

Java's nesting capabilities help you organize non-top-level reference types. For top-level reference types, Java provides packages. The next **Java 101** tutorial introduces you to packages and the related topic of static imports.

Jeff Friesen teaches Java technology (including Android) to everyone.

Follow