The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Classes and Objects
Section: Nested Classes

Local Classes

Local classes are classes that are defined in a *block*, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method.

This section covers the following topics:

Declaring Local Classes

You can define a local class inside any block (see Expressions, Statements, and Blocks for more information). For example, you can define a local class in a method body, a `for` loop, or an `if` clause.

The following example, `LocalClassExample`, validates two phone numbers. It defines the local class `PhoneNumber` in the method `validatePhoneNumber`:

```
public class LocalClassExample {

    static String regularExpression = "[^0-9]";

    public static void validatePhoneNumber(
        String phoneNumber1, String phoneNumber2) {

        final int numberLength = 10;

        // Valid in JDK 8 and later:

        // int numberLength = 10;

        class PhoneNumber {

            String formattedPhoneNumber = null;

            PhoneNumber(String phoneNumber){
                // numberLength = 7;
                String currentNumber = phoneNumber.replaceAll(
                  regularExpression, "");
                if (currentNumber.length() == numberLength)
                    formattedPhoneNumber = currentNumber;
                else
                    formattedPhoneNumber = null;
            }

            public String getNumber() {
                return formattedPhoneNumber;
            }

            // Valid in JDK 8 and later:

//            public void printOriginalNumbers() {
//                System.out.println("Original numbers are " + phoneNumber1 +
//                    " and " + phoneNumber2);
//            }
        }

        PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
        PhoneNumber myNumber2 = new PhoneNumber(phoneNumber2);

        // Valid in JDK 8 and later:

//        myNumber1.printOriginalNumbers();
```

```
            if (myNumber1.getNumber() == null)
                System.out.println("First number is invalid");
            else
                System.out.println("First number is " + myNumber1.getNumber());
            if (myNumber2.getNumber() == null)
                System.out.println("Second number is invalid");
            else
                System.out.println("Second number is " + myNumber2.getNumber());

        }

        public static void main(String... args) {
            validatePhoneNumber("123-456-7890", "456-7890");
        }
    }
```

The example validates a phone number by first removing all characters from the phone number except the digits 0 through 9. After, it checks whether the phone number contains exactly ten digits (the length of a phone number in North America). This example prints the following:

```
First number is 1234567890
Second number is invalid
```

Accessing Members of an Enclosing Class

A local class has access to the members of its enclosing class. In the previous example, the `PhoneNumber` constructor accesses the member `LocalClassExample.regularExpression`.

In addition, a local class has access to local variables. However, a local class can only access local variables that are declared final. When a local class accesses a local variable or parameter of the enclosing block, it *captures* that variable or parameter. For example, the `PhoneNumber` constructor can access the local variable `numberLength` because it is declared final; `numberLength` is a *captured variable*.

However, starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or *effectively final*. A variable or parameter whose value is never changed after it is initialized is effectively final. For example, suppose that the variable `numberLength` is not declared final, and you add the highlighted assignment statement in the `PhoneNumber` constructor:

```
PhoneNumber(String phoneNumber) {
    numberLength = 7;
    String currentNumber = phoneNumber.replaceAll(
        regularExpression, "");
    if (currentNumber.length() == numberLength)
        formattedPhoneNumber = currentNumber;
    else
        formattedPhoneNumber = null;
}
```

Because of this assignment statement, the variable `numberLength` is not effectively final anymore. As a result, the Java compiler generates an error message similar to "local variables referenced from an inner class must be final or effectively final" where the inner class `PhoneNumber` tries to access the `numberLength` variable:

```
if (currentNumber.length() == numberLength)
```

Starting in Java SE 8, if you declare the local class in a method, it can access the method's parameters. For example, you can define the following method in the `PhoneNumber` local class:

```
public void printOriginalNumbers() {
    System.out.println("Original numbers are " + phoneNumber1 +
        " and " + phoneNumber2);
}
```

The method `printOriginalNumbers` accesses the parameters `phoneNumber1` and `phoneNumber2` of the method `validatePhoneNumber`.

Shadowing and Local Classes

Declarations of a type (such as a variable) in a local class shadow declarations in the enclosing scope that have the same name. See Shadowing for more information.

Local Classes Are Similar To Inner Classes

不能定义或声明任何静态成员

Local classes are similar to inner classes because they cannot define or declare any static members. Local classes in static methods, such as the class `PhoneNumber`, which is defined in the static method `validatePhoneNumber`, can only refer to static members of the enclosing class. For example, if you do not define the member variable `regularExpression` as static, then the Java compiler generates an error similar to "non-static variable `regularExpression` cannot be referenced from a static context."

Local classes are non-static because they have access to instance members of the enclosing block. Consequently, they cannot contain most kinds of static declarations.

You cannot declare an interface inside a block; interfaces are inherently static. For example, the following code excerpt does not compile because the interface

`HelloThere` is defined inside the body of the method `greetInEnglish`:

```
public void greetInEnglish() {
    interface HelloThere {
       public void greet();
    }
    class EnglishHelloThere implements HelloThere {
        public void greet() {
            System.out.println("Hello " + name);
        }
    }
    HelloThere myGreeting = new EnglishHelloThere();
    myGreeting.greet();
}
```

You cannot declare static initializers or member interfaces in a local class. The following code excerpt does not compile because the method `EnglishGoodbye.sayGoodbye` is declared `static`. The compiler generates an error similar to "modifier 'static' is only allowed in constant variable declaration" when it encounters this method definition:

```
public void sayGoodbyeInEnglish() {
    class EnglishGoodbye {
        public static void sayGoodbye() {
            System.out.println("Bye bye");
        }
    }
    EnglishGoodbye.sayGoodbye();
}
```

常量变量

A local class can have static members provided that they are constant variables. (A *constant variable* is a variable of primitive type or type `String` that is declared final and initialized with a compile-time constant expression. A compile-time constant expression is typically a string or an arithmetic expression that can be evaluated at compile time. See Understanding Class Members for more information.) The following code excerpt compiles because the static member `EnglishGoodbye.farewell` is a constant variable:

```
public void sayGoodbyeInEnglish() {
    class EnglishGoodbye {
        public static final String farewell = "Bye bye";
        public void sayGoodbye() {
            System.out.println(farewell);
        }
    }
    EnglishGoodbye myEnglishGoodbye = new EnglishGoodbye();
    myEnglishGoodbye.sayGoodbye();
}
```

Previous page: Inner Class Example
Next page: Anonymous Classes