

Inner Classes

7

Exam Objectives

- Write code to construct instances of any concrete class, including normal top-level classes, inner classes, static inner classes and anonymous inner classes.

Supplementary Objectives

- State the kinds of nested classes and interfaces that can be defined.
- Identify the context in which a nested class or interface can be defined.
- State which accessibility modifiers are allowed for each category of nested classes.
- State which nested classes create instances that are associated with instances of the outer context.
- State the access rules that govern accessing entities in the outer context of nested classes, and write code that uses the augmented syntax involving the `this` keyword for this purpose.
- State whether a definition of a nested class can contain static and non-static members.
- Write code to instantiate nested classes using the augmented syntax of the `new` operator.
- Write code to show how nested classes can be imported and used.
- Distinguish between the inheritance and containment hierarchy of any nested class or interface.
- Write code to implement anonymous classes by extending an existing class and by implementing an interface.

7.1 Overview of Nested Classes

An ordinary class or interface is defined at the *top level*, which is the package level. These classes and interfaces (also called *top-level package member classes or interfaces*) are grouped into packages. In addition to these top-level package member classes and interfaces, there are four categories of *nested classes* and one of *nested interfaces* defined by the context these classes and interfaces are declared in:

- Top-level Nested Classes and Interfaces
- Non-static Inner Classes
- Local Classes
- Anonymous Classes

Top-level nested classes and interfaces are also considered to be at the top level. The last three categories are collectively known as *inner classes*. These differ from top-level classes in that they can use simple non-static names from the *enclosing context*, i.e. an instance of an inner class is not limited to directly accessing only its own instance variables. In particular, an instance of an inner class may be associated with an instance of the enclosing class and may access its members.

A *top-level nested class or interface* is defined as a static member in a top-level (possibly nested) class or interface. Such a nested class can be instantiated like any ordinary top-level class, using its full name. No instance of the enclosing class is required to instantiate a top-level nested class. *Non-static inner classes* are defined as instance members of other classes, just like instance variable and method members are defined in a class. An instance of a non-static inner class always has an instance of the enclosing class associated with it. *Local classes* can be defined in a block of code as in a method body or a local block, just as local variables can be defined in a method body or a local block. *Anonymous classes* can be defined and instantiated “on the fly” in expressions. Local and anonymous classes can be either *static* or *non-static*, where being non-static means that an instance of such a class is associated with an instance of the enclosing class.

Table 7.1 presents a summary of various aspects relating to classes and interfaces. The Entity column lists the different kinds of classes and interfaces that can be defined. The Declaration Context column lists the lexical context in which the class or interface can be defined. The Accessibility Modifiers column indicates what accessibility can be specified for the class or interface. The Outer Instance column specifies whether an instance of the enclosing context is associated with an instance of the class. The Direct Access to Enclosing Context column lists what is directly accessible in the enclosing context from within the class. The Defines Static or Non-static Members column refers to whether the class can define static or non-static members, or both. Subsequent sections on each nested class elaborate on the summary presented in Table 7.1. (N/A in the table means not applicable.)

Nested classes can be regarded as a form of encapsulation, enforcing relationships between classes by greater proximity. Used judiciously, they can be beneficial, but unrestrained nesting can easily result in incomprehensible code.

Table 7.1 *Overview of Classes and Interfaces*

Entity	Declaration Context	Access-ibility Modifiers	Outer Instance	Direct Access to Enclosing Context	Defines Static or Non-static Members
Package-level class	As package member	public or default	No	N/A	Both static and non-static
Top-level nested class (static)	As static class member	all	No	Static members in enclosing context	Both static and non-static
Non-static inner class	As non-static class member	all	Yes	All members in enclosing context	Only non-static
Local class (non-static)	In block with non-static context	none	Yes	All members in enclosing context + local final variables	Only non-static
Local class (static)	In block with static context	none	No	Static members in enclosing context + local final variables	Only non-static
Anonymous class (non-static)	As expression in non-static context	none	Yes	All members in enclosing context + local final variables	Only non-static
Anonymous class (static)	As expression in static context	none	No	Static members in enclosing context + local final variables	Only non-static
Interface	As package member or static class member	public only	N/A	N/A	Static variables + non-static method prototypes

7.2 Top-level Nested Classes and Interfaces

A *top-level nested class or interface* is similar to a top-level package member class or interface, but it is defined as a static member of an enclosing top-level class or interface. Top-level nested classes and interfaces can be nested to any depth, but only within other static top-level classes and interfaces.

Interfaces are implicitly static. Nested interfaces can optionally be prefixed with the keyword `static` and have any accessibility. There are no non-static inner, local or anonymous interfaces – only (possibly nested) static top-level interfaces.

Example 7.1 *Top-level Nested Classes and Interfaces*

```

.....
// Filename: TopLevelClass.java
public class TopLevelClass {                               // (1)
    // ...
    static class NestedTopLevelClass {                     // (2)
        // ...
        interface NestedTopLevelInterface1 {              // (3)
            // ...
        }
        static class NestedTopLevelClass1
            implements NestedTopLevelInterface1 {         // (4)
            // ...
        }
    }
}
.....

```

In Example 7.1, the top-level package member class `TopLevelClass` at (1) contains a nested top-level class `NestedTopLevelClass` at (2), which in turn defines a nested top-level interface `NestedTopLevelInterface1` at (3), which is implemented by the nested top-level class `NestedTopLevelClass1` at (4). Note that each nested top-level class is defined as `static`, just like `static` variables and methods in a class.

The full name of a nested top-level class or interface includes the name of the class it is defined in. For example, the full name of the nested top-level class `NestedTopLevelClass1` at (4) is `TopLevelClass.NestedTopLevelClass.NestedTopLevelClass1`. The full name of the nested top-level interface `NestedTopLevelInterface1` at (3) is `TopLevelClass.NestedTopLevelClass.NestedTopLevelInterface1`. Note that each nested top-level class or interface is uniquely identified by this naming convention, which is a generalization of the fully qualified naming scheme for package members. The full class name can be used in a program, like any other class or interface name. Note that a nested class cannot have the same name as an enclosing class or package.

If the file `TopLevelClass.java` containing the definitions in Example 7.1 is compiled, it will result in the generation of the following class files, where each file corresponds to a class or interface definition:

```
TopLevelClass$NestedTopLevelClass$NestedTopLevelClass1.class
TopLevelClass$NestedTopLevelClass$NestedTopLevelInterface1.class
TopLevelClass$NestedTopLevelClass.class
TopLevelClass.class
```

Note how the full class name corresponds to the file name (minus the extension) with the dollar sign (\$) replaced by the dot sign (.).

A client can use the `import` statement to provide a shortcut for the names of nested top-level classes and interfaces. Here are some variations on usage of the `import` statement for nested top-level classes and interfaces:

```
// Filename: Client1.java
import TopLevelClass.*;                               // (1)

public class Client {
    NestedTopLevelClass.NestedTopLevelClass1 objRef1 =
        new NestedTopLevelClass.NestedTopLevelClass1(); // (2)
}

.....

// Filename: Client2.java
import TopLevelClass.NestedTopLevelClass.*;           // (3)

public class Client2 {
    NestedTopLevelClass1 objRef2 = new NestedTopLevelClass1(); // (4)
}

class SomeClass implements
    TopLevelClass.NestedTopLevelClass.NestedTopLevelInterface1 { // (5)
    /* ... */
}
```

In the file `Client1.java`, the `import` statement at (1) allows the nested top-level class `NestedTopLevelClass1` to be referenced as `NestedTopLevelClass.NestedTopLevelClass1` as at (2), whereas in the file `Client2.java`, the `import` statement at (3) will allow the same class to be referenced using the simple name as at (4). At (5), the full name of the nested top-level interface is used in an `implements` clause.

For all intents and purposes, a top-level nested class or interface is very much like any other top-level package member class or interface. Static variables and methods belong to a class, and not to instances of the class. The same is true for nested top-level classes. A (static) nested top-level class can be instantiated without any reference to any instance of the enclosing context, i.e. objects of a nested top-level class can be created without regard to its nesting. Examples of creating instances of nested top-level classes are shown above at (2) and (4) using the `new` operator.

Static methods do not have a `this` reference and can therefore only access other static methods and variables directly in the class. This also applies to methods in

a nested top-level class. A method in a nested top-level class can only directly access static members in the enclosing class or interface, but not instance members in the enclosing context. Since nested top-level classes are static, their methods do not have any (outer) instance of the enclosing context.

Figure 7.1 is a class diagram that illustrates top-level nested classes and interfaces. These are shown as members of the enclosing context, with the {static} tag to indicate that they are static, i.e. they can be instantiated without regard to any outer object of the enclosing context. Since they are members of a class or an interface, their accessibility can be specified exactly like that of any other member of a class or interface. The classes from the diagram are implemented in Example 7.2.

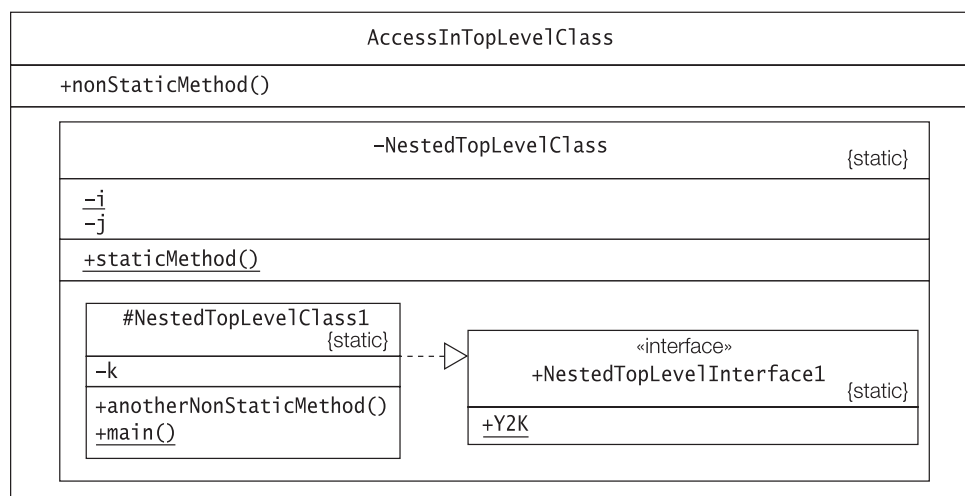


Figure 7.1 *Top-level Nested Classes and Interfaces*

Example 7.2 *Access in Top-level Nested Classes and Interfaces*

```

// Filename: AccessInTopLevelClass.java
public class AccessInTopLevelClass { // (1)
    public void nonStaticMethod() { // (2)
        System.out.println("nonstaticMethod in AccessInTopLevelClass");
    }

    private static class NestedTopLevelClass { // (3)
        private static int i; // (4)
        private int j; // (5)

        public static void staticMethod() { // (6)
            System.out.println("staticMethod in NestedTopLevelClass");
        }
    }

    interface NestedTopLevelInterface1 { int Y2K = 2000; } // (7)

    protected static class NestedTopLevelClass1
        implements NestedTopLevelInterface1 { // (8)
        private int k = Y2K; // (9)
    }
}
  
```

```

        public void anotherNonStaticMethod() {           // (10)
        // int jj = j;           // (11) Not OK.
        int ii = i;           // (12)
        int kk = k;           // (13)

        // nonStaticMethod();   // (14) Not OK.
        staticMethod();       // (15)
        }

        public static void main (String args[]) {
        int ii = i;           // (16)
        // int kk = k;           // (17) Not OK.
        staticMethod();       // (18)
        }
    }
}

```

Output from the program:

```
staticMethod in NestedTopLevelClass
```

Example 7.2 demonstrates accessing members directly in the enclosing context of class `NestedTopLevelClass1` defined at (8). The initialization at (9) is valid, since the variable `Y2K`, defined in the outer interface `NestedTopLevelInterface1` at (7), is implicitly static. The compiler will flag an error at (11) and (14) in method `anotherNonStaticMethod()`, because direct access to non-static members in the enclosing class is not permitted by *any* method in a nested top-level class. It will also flag an error at (17) in method `main()`, because a static method cannot access directly other non-static variables in its own class. Statements at (16) and (18) only access static members in the enclosing context. The references in these statements can also be specified using full names:

```
int ii = AccessInTopLevelClass.NestedTopLevelClass.i;
AccessInTopLevelClass.NestedTopLevelClass.staticMethod();
```

Note that a top-level nested class can define both static and instance members, like any other package-level class. However, its code can only directly access static members in its enclosing context.

A top-level nested class, being a member of the enclosing class or interface, can have any accessibility (`public`, `protected`, `package/default`, `private`), like any other members of a class. The class `NestedTopLevelClass` at (3) has `private` accessibility, whereas its nested class `NestedTopLevelClass1` at (8) has `protected` accessibility. The class `NestedTopLevelClass1` defines the method `main()`, which can be executed by the command:

```
java AccessInTopLevelClass$NestedTopLevelClass$NestedTopLevelClass1
```

Note that the class `NestedTopLevelClass1` is specified using the full name of the class file, minus the extension.

7.3 Non-static Inner Classes

Non-static inner classes are defined without the keyword `static`, as members of an enclosing class, and can also be nested to any depth. Non-static inner classes are on par with other non-static members defined in a class. The following aspects about non-static inner classes should be noted:

- An instance of a non-static inner class can only exist with an instance of its enclosing class. This means that an instance of a non-static inner class must be created in the context of an instance of the enclosing class. This also means that a non-static inner class cannot have static members. In other words, the class does not provide any services, only instances of the class do.
- Methods of a non-static inner class can directly refer to any member (including classes) of any enclosing class, including private members. No explicit reference is required.
- Since a non-static inner class is a member of an enclosing class, it can have any accessibility: `public`, `package/default`, `protected` or `private`.

Example 7.3 Defining Non-static Inner Classes

```

class ToplevelClass { // (1)
    private String msg = "Shine the inner light."; // (2)
    public NonStaticInnerClass makeInstance() { // (3)
        return new NonStaticInnerClass(); // (4)
    }
    public class NonStaticInnerClass { // (5)
        // private static int staticVar; // (6) Not OK.
        private String string; // (7)
        public NonStaticInnerClass() { string = msg; } // (8)
        public void printMsg() { System.out.println(string); } // (9)
    }
}

public class Client { // (10)
    public static void main(String args[]) { // (11)
        ToplevelClass topRef = new ToplevelClass(); // (12)
        ToplevelClass.NonStaticInnerClass innerRef1 =
            topRef.makeInstance(); // (13)
        innerRef1.printMsg(); // (14)
        // ToplevelClass.NonStaticInnerClass innerRef2 =
        //     new ToplevelClass.NonStaticInnerClass(); // (15) Not OK.
        ToplevelClass.NonStaticInnerClass innerRef3 =
            topRef.new NonStaticInnerClass(); // (16)
    }
}

```

Output from the program:

```
Shine the inner light.
```

In Example 7.3, the class `TopLevelClass` at (1) defines a non-static inner class at (5). Declaration of a static variable in class `NonStaticInnerClass` would be flagged as a compile time error, as a non-static inner class cannot define static members.

The non-static method `makeInstance()` at (3) in the class `TopLevelClass` creates an instance of the `NonStaticInnerClass` using the `new` operator, as shown at (4). This creates an instance of a non-static inner class in the context of the instance of the enclosing class on which the `makeInstance()` method is invoked. The `makeInstance()` method is called at (13). The reference to an object of the non-static inner class can then be used in the normal way to access its members, as shown at (14). An attempt to create an instance of the non-static inner class, without an outer instance, using the `new` operator with the full name of the inner class, as shown at (15), results in a compile time error. A special form of the `new` operator must be used, which is illustrated at (16):

```
topRef.new NonStaticInnerClass();           // (16)
```

The expression `<enclosing object reference>` in the syntax

```
<enclosing object reference>.new
```

evaluates to an instance of the enclosing class in which the designated non-static inner class is defined. It is an error to specify the full name of the inner class, as the enclosing context is already given by `<enclosing object reference>`. The reference `topRef` denotes an object of class `TopLevelClass`. After the execution of the statement at (16), the `TopLevelClass` object has two instances of the inner class `NonStaticInnerClass` associated with it. This is depicted in Figure 7.2, where the outer object (denoted by `topRef`) of class `TopLevelClass` is shown with its two associated inner objects (denoted by `innerRef1` and `innerRef3` respectively) right after the execution of the statement at (16). In other words, multiple objects of the inner classes can be associated with an object of an enclosing class at runtime.

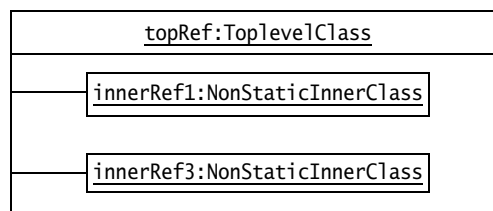


Figure 7.2 Outer Object with Associated Inner Objects

An implicit reference to the enclosing object is always available in every method and constructor of a non-static inner class. A method can explicitly use this reference with a special form of the `this` construct, as explained below.

From within a non-static inner class, it is possible to refer to members in the enclosing class directly. An example is shown at (8), where the instance variable `msg` from the enclosing class is accessed in the non-static inner class. It is also possible to

explicitly refer to members in the enclosing class, but this requires special usage of the `this` reference. One might be tempted to define the constructor at (8) as follows:

```
public NonStaticInnerClass() { this.string = this.msg; }
```

The reference `this.string` is correct, because the instance variable `string` certainly belongs to the current object (denoted by `this`) of `NonStaticInnerClass`, but `this.msg` cannot possibly work, as the current object (indicated by `this`) of `NonStaticInnerClass` has no instance variable `msg`. The correct syntax is the following:

```
public NonStaticInnerClass() { this.string = ToplevelClass.this.msg; }
```

The expression

```
<enclosing class name>.this
```

evaluates to a reference that denotes the enclosing object (of class `<enclosing class name>`) of the current instance of a non-static inner class.

Accessing Shadowed Members from Non-static Inner Classes

As non-static inner classes can be nested, names of instance members in enclosing classes can become *shadowed*. The special form of the `this` syntax can be used to access members in the enclosing context.

Example 7.4 Special Form of `this` and new Constructs in Non-static Inner Classes

```
.....
// Filename: Client2.java
class TlClassA {                                     // (1) Top-level Class
    private String msg = "TlClassA object ";
    public TlClassA(String objNo) { msg = msg + objNo; }
    public void printMessage() { System.out.println(msg); }

    class InnerB {                                   // (2) Non-static Inner Class
        private String msg = "InnerB object ";
        public InnerB(String objNo) { msg = msg + objNo; }
        public void printMessage() { System.out.println(msg); }

        class InnerC {                             // (3) Non-static Inner Class
            private String msg = "InnerC object ";
            public InnerC(String objNo) { msg = msg + objNo; }
            public void printMessage() {
                System.out.println(msg);           // (4)
                System.out.println(this.msg);      // (5)
                System.out.println(InnerC.this.msg); // (6)
                System.out.println(InnerB.this.msg); // (7)
                InnerB.this.printMessage();        // (8)
                System.out.println(TlClassA.this.msg); // (9)
                TlClassA.this.printMessage();      // (10)
            }
        }
    }
}
```

```

public class Client2 { // (11)
    public static void main(String args[]) { // (12)
        TlClassA a = new TlClassA("1"); // (13)
        TlClassA.InnerB b = a.new InnerB("1"); // (14)
        TlClassA.InnerB.InnerC c = b.new InnerC("1"); // (15)
        c.printMessage(); // (16)
        TlClassA.InnerB bb = new TlClassA("2").new InnerB("2");// (17)
        TlClassA.InnerB.InnerC cc = bb.new InnerC("2"); // (18)
        cc.printMessage(); // (19)
        TlClassA.InnerB.InnerC ccc =
            new TlClassA("3").new InnerB("3").new InnerC("3"); // (20)
    }
}

```

Output from the program:

```

InnerC object 1
InnerC object 1
InnerC object 1
InnerB object 1
InnerB object 1
TlClassA object 1
TlClassA object 1
InnerC object 2
InnerC object 2
InnerC object 2
InnerB object 2
InnerB object 2
TlClassA object 2
TlClassA object 2

```

Example 7.4 illustrates the special form of the `this` construct to access members in the enclosing context, and also demonstrates the special form of the `new` construct to create instances of non-static inner classes. The example shows the class `InnerC`, defined at (3), which is nested in the class `InnerB` defined at (2), which in turn is nested in the top-level class `TlClassA` at (1). All three classes have a private non-static `String` variable `msg` and a non-static method `printMessage()`. These members are *not* overridden in the inner classes, as no inheritance is involved. Like any other class member, they have class scope.

The `main()` method at (12) uses the additional syntax of the `new` operator to create an instance of `InnerC` (denoted by `c`) at (15) in the context of an instance of class `InnerB` (denoted by `b`) at (14), which in turn is created in the context of an instance of class `TlClassA` (denoted by `a`) at (13).

The reference `c` is used at (16) to invoke the method `printMessage()` from class `InnerC`. This method uses the standard `this` reference to access members of the object on which it is invoked, as shown at (5). It also uses the special form of the `this` construct, in conjunction with the class name, to access members in (outer) objects which are associated with the current object, as shown in the statements from (6) through (10).

When the intervening references to a non-static inner class are of no interest, the `new` operator can be chained, as shown at (17), (18) and (20).

Note that the (outer) objects associated with the references `c`, `cc` and `ccc` are distinct, as evident from the program output.

Compiling and Importing Non-static Inner Classes

If the file `Client2.java` containing the definitions from Example 7.4 is compiled, it will result in the generation of the following class files, where each file corresponds to a class definition:

```
TLClassA$InnerB$InnerC.class
TLClassA$InnerB.class
TLClassA.class
Client2.class
```

Clients can use the `import` statement to provide a shortcut for the names of non-static inner classes. Example 7.5, based on Example 7.4, shows how nested classes can be imported and used. Note the specification of the class name in the `import` statement. It “imports” the *immediate* nested classes as shown at (14’). For deeply nested inner classes the intervening class names must be specified, as shown in the declarations at (15’), (18’) and (20’). If the class `TLClassA` belonged to a named package, then the package name must be prepended to the class’s name in the usual way in the `import` statement.

Example 7.5 *Importing Inner Classes*

```
// Filename: Client3.java
import TLClassA.*;

// Uses classes from Example 7.4.

public class Client3 {
    public static void main(String args[]) { // (12)
        TLClassA a = new TLClassA("1"); // (13)
        InnerB b = a.new InnerB("1"); // (14')
        InnerB.InnerC c = b.new InnerC("1"); // (15')
        InnerB bb = new TLClassA("2").new InnerB("2"); // (17')
        InnerB.InnerC cc = bb.new InnerC("2"); // (18')
        InnerB.InnerC ccc =
            new TLClassA("3").new InnerB("3").new InnerC("3"); // (20')
        ccc.printMessage();
    }
}
```

Output from the program:

```
InnerC object 3
InnerC object 3
InnerC object 3
InnerB object 3
```

```

InnerB object 3
TLClassA object 3
TLClassA object 3

```

Inheritance and Containment Hierarchy of Non-static Inner Classes

Non-static inner classes can extend other classes and can themselves be extended. Therefore both the inheritance and containment hierarchy must be considered when dealing with member access. Imagine a subclass C that is derived from superclass B, and class C is also a non-static inner class in an enclosing class A. In the absence of name conflicts there is no problem, but what if both the superclass B and the enclosing class A had a member with the name x? If a name conflict arises, the inherited member shadows the member with the same name in the enclosing class. The compiler, however, requires that explicit references be used.

In Example 7.6, the compiler would flag an error at (3) as the reference x is ambiguous. The standard form of the this reference can be used to access the inherited member, as shown at (4). The keyword super would be another alternative. To access the member from the enclosing context, the special form of the this reference together with the enclosing class name is used, as shown at (5).

Example 7.6 *Inheritance and Containment Hierarchy*

```

class B {
    protected double x = 2.17;
}

class A {
    private double x = 3.14;
    class C extends B {
        // private double w = x;
        private double y = this.x;
        private double z = A.this.x;
        public void printX() {
            System.out.println("this.x: " + y);
            System.out.println("A.this.x: " + z);
        }
    }
}

public class Client4 {
    public static void main(String args[]) {
        A.C ref = new A().new C();
        ref.printX();
    }
}

```

// (1) Top-level Class
// (2) Non-static inner Class
// (3) Compile time error
// (4) x from superclass
// (5) x from enclosing class
// (6)
// (7)

Output from the program:

```
this.x: 2.17
A.this.x: 3.14
```



Review questions

7.1 What will be the result of attempting to compile and run the following code?

```
public class MyClass {
    public static void main(String args[]) {
        Outer objRef = new Outer();
        System.out.println(objRef.createInner().getSecret());
    }
}

class Outer {
    private int secret;
    Outer() { secret = 123; }

    class Inner {
        int getSecret() { return secret; }
    }

    Inner createInner() { return new Inner(); }
}
```

Select the one right answer.

- (a) The code will fail to compile, since the class Inner cannot be declared within the class Outer.
- (b) The code will fail to compile, since the method createInner() cannot be allowed to pass objects of the inner class Inner to methods outside of the class Outer.
- (c) The code will fail to compile, since the secret variable is not accessible from the method getSecret().
- (d) The code will fail to compile, since the method getSecret() is not visible from the main() method in the class MyClass.
- (e) The code will compile without error and will print 123 when run.

7.2 Which of these statements concerning nested classes are true?

Select all valid answers.

- (a) An instance of a top-level nested class has an inherent outer instance.
- (b) A top-level nested class can contain non-static member variables.
- (c) A top-level nested interface can contain non-static member variables.
- (d) A top-level nested interface has an inherent outer instance.
- (e) For each instance of the outer class, there can exist many instances of a non-static inner class.

7.3 What will be the result of attempting to compile and run the following code?

```
public class MyClass {
    public static void main(String args[]) {
        State st = new State();
        System.out.println(st.getValue());
        State.Memento mem = st.memento();
        st.alterValue();
        System.out.println(st.getValue());
        mem.restore();
        System.out.println(st.getValue());
    }

    public static class State {
        protected int val = 11;

        int getValue() { return val; }
        void alterValue() { val = (val + 7) % 31; }
        Memento memento() { return new Memento(); }

        class Memento {
            int val;

            Memento() { this.val = State.this.val; }
            void restore() { ((State) this).val = this.val; }
        }
    }
}
```

Select the one right answer.

- (a) The code will fail to compile, since the static `main()` method tries to create a new instance of the inner class `State`.
- (b) The code will fail to compile, since the class declaration of `State.Memento` is not visible from the `main()` method.
- (c) The code will fail to compile, since the inner class `Memento` declares a variable with the same name as a variable in the outer class `State`.
- (d) The code will fail to compile, since the `Memento` constructor tries an invalid access through the `State.this.val` expression.
- (e) The code will fail to compile, since the `Memento` method `restore()` tries an invalid access through the `((State) this).val` expression.
- (f) The program compiles without errors and prints 11, 18 and 11 when run.

7.4 What will be the result of attempting to compile and run the following program?

```
public class Nesting {
    public static void main(String args[]) {
        B.C obj = new B().new C();
    }
}

class A {
    int val;
    A(int v) { val = v; }
}
```

```

class B extends A {
    int val = 1;
    B() { super(2); }

    class C extends A {
        int val = 3;
        C() {
            super(4);
            System.out.println(B.this.val);
            System.out.println(C.this.val);
            System.out.println(super.val);
        }
    }
}

```

Select all valid answers.

- (a) The program will fail to compile.
- (b) The program will compile without error, and print 2, 3 and 4 in that order when run.
- (c) The program will compile without error, and print 1, 4 and 2 in that order when run.
- (d) The program will compile without error, and print 1, 3 and 4 in that order when run.
- (e) The program will compile without error, and print 3, 2 and 1 in that order when run.

7.4 Local Classes

A local class is a class that is defined in a block. This could be a method body, a constructor, a local block, a static initializer or an instance initializer. Such a local class is only visible within the context of the block, i.e. the name of the class is only valid in the context of the block in which it is defined. A local class cannot be specified with the keyword `static`. However, if the context is static (i.e. a static method or a static initializer) then the local class is implicitly static. Otherwise, the local class is non-static.

Like non-static inner classes, an instance of a non-static local class is passed a hidden reference designating an instance of its enclosing class in its constructors, and this gives non-static local classes much of the same capability as non-static inner classes. Some restrictions which apply to local classes are:

- Local classes cannot have static members, as they cannot provide class-specific services.
- Local classes cannot have any accessibility. This restriction applies to local variables, and is also enforced for local classes.

Example 7.7 *Access in Local Classes*

```

class SuperB {
    protected double x;
    protected static int n;
}

class SuperC {
    protected double y;
    protected static int m;
}

class TopLevelA extends SuperC {           // Top-level Class
    private double z;
    private static int p;

    void nonStaticMethod(final int i) {    // Non-static Method
        final int j = 10;
        int k;
        class NonStaticLocalD extends SuperB { // Non-static local class
            // static double d; // (1) Not OK. Only non-static members allowed.
            int ii = i; // (2) final from enclosing method.
            int jj = j; // (3) final from enclosing method.
            // double kk = k; // (4) Not OK. Only finals from enclosing method.
            double zz = z; // (5) non-static from enclosing class.
            int pp = p; // (6) static from enclosing class.
            double yy = y; // (7) inherited by enclosing class.
            int mm = m; // (8) static from enclosing class.
            double xx = x; // (9) non-static inherited from superclass
            int nn = n; // (10) static from superclass
        }
    }

    static void staticMethod(final int i) { // Static Method
        final int j = 10;
        int k;
        class StaticLocalE extends SuperB { // Static local class
            // static double d; // (11) Not OK. Only non-static members allowed.
            int ii = i; // (12) final from enclosing method.
            int jj = j; // (13) final from enclosing method.
            // double kk = k; // (14) Not OK. Non-final from enclosing method.
            // double zz = z; // (15) Not OK. Non-static member.
            int pp = p; // (16) static from enclosing class.
            // double yy = y; // (17) Not OK. Non-static member.
            int mm = m; // (18) static from enclosing class.
            double xx = x; // (19) non-static inherited from superclass
            int nn = n; // (20) static from superclass
        }
    }
}

```

Access Rules for Local Classes

Example 7.7 illustrates the access rules for local classes, which are stated below.

- A local class can access members defined within the class. This should not come as a surprise.
- A local class can access `final` local variables, `final` method parameters and `final` catch-block parameters in the scope of the local context. Such `final` variables are also read-only in the local class. This situation is shown at (2) and (3), where the `final` parameter `i` and the `final` local variable `j` of the method `nonStaticMethod()` in the non-static local class `NonStaticLocalD` are accessed. This also applies to static local classes, as shown at (12) and (13). Access to non-final local variables is not permitted from local classes, as shown at (4) and (14).
- A non-static local class can access members defined in the enclosing class. This situation is shown at (5) and (6), where the instance variable `z` and static variable `p` defined in the enclosing class `TopLevelA` are accessed, respectively. The special form of the `this` construct can be used for *explicit* referencing of members defined in the enclosing class:

```
double zz = TopLevelA.this.z;
```

However, a static local class can only directly access static members defined in the enclosing class, as shown at (16), but not non-static members, as shown at (15).

- A non-static local class can directly access members inherited by the enclosing class. This situation is shown at (7) and (8), where the instance variable `y` and the static variable `m` are inherited by the enclosing class `TopLevelA` from the superclass `SuperC`. The special form of the `this` construct can also be used in the local class for *explicit* referencing of members inherited by the enclosing class:

```
double yy = TopLevelA.this.y;
```

However, a static local class can only directly access static members that are inherited by the enclosing class, as shown at (18), but not non-static members, as shown at (17).

- A local class can access members inherited from its superclass in the usual way. The instance variables `x` and `n` in the superclass `SuperB` are inherited by the local subclass `NonStaticLocalD`. These variables are accessed in the local class `NonStaticLocalD` as shown at (9) and (10). The standard `this` reference (or the `super` keyword) can be used for referencing members inherited by the local class:

```
double xx = this.x;
```

Note that this also applies for static local classes. This is shown at (19) and (20).

Instantiating Local Classes

Clients outside the context of a local class cannot create or access these classes directly, because they are after all local. A local class can be instantiated in the block

in which it is defined. A method can return an instance of the local class. The local class type must then be assignable to the return type of the method. It cannot be the same as the local class type, since this type is not accessible outside of the method. Often a supertype of the local class is specified as the return type.

Example 7.8 *Instantiating Local Classes*

```

interface IDrawable { // (1)
    void draw();
}
class Shape implements IDrawable { // (2)
    public void draw() { System.out.println("Drawing a Shape."); }
}
class Painter { // (3) Top-level Class
    public Shape createCircle(final double radius) { // (4) Non-static Method
        class Circle extends Shape { // (5) Non-static local class
            public void draw() {
                System.out.println("Drawing a Circle of radius: " + radius);
            }
        }
        return new Circle(); // (6) Object of non-static local class
    }

    public static IDrawable createMap() { // (7) Static Method
        class Map implements IDrawable { // (8) Static local class
            public void draw() { System.out.println("Drawing a Map."); }
        }
        return new Map(); // (9) Object of static local class
    }
}

public class Client {
    public static void main(String args[]) {
        IDrawable[] drawables = { // (10)
            new Painter().createCircle(5), // (11) Object of non-static local class
            Painter.createMap(), // (12) Object of static local class
            new Painter().createMap() // (13) Object of static local class
        };
        for (int i = 0; i < drawables.length; i++) // (14)
            drawables[i].draw();

        System.out.println("Local Class Names:");
        System.out.println(drawables[0].getClass()); // (15)
        System.out.println(drawables[1].getClass()); // (16)
    }
}

```

Output from the program:

```

Drawing a Circle of radius: 5.0
Drawing a Map.
Drawing a Map.

```

```
Local Class Names:
class Painter$1$Circle
class Painter$1$Map
```

.....

Example 7.8 illustrates how clients can instantiate local classes. The non-static local class `Circle` at (5) is defined in the non-static method `createCircle()` at (4), which has the return type `Shape`. The static local class `Map` at (8) is defined in the static method `createMap()` at (7), which has the return type `IDrawable`. The inheritance hierarchy of the local classes and their supertypes `Shape` and `IDrawable` is depicted in Figure 6.5. The `main()` method creates a polymorphic array `drawables` of type `IDrawable` at (10), which is initialized with instances of the local classes:

```
IDrawable[] drawables = {           // (10)
    new Painter().createCircle(5), // (11) Object of non-static local class
    Painter.createMap(),           // (12) Object of static local class
    new Painter().createMap()      // (13) Object of static local class
};
```

Creating an instance of a non-static local class requires an instance of the enclosing class. The non-static method `createCircle()` is invoked on the instance of the enclosing class to create an instance of the non-static local class, as shown at (11). In the non-static method, the reference to the instance of the enclosing context is passed implicitly in the constructor call of the non-static local class at (6).

A static method can be invoked either through the class name or through an instance of the class. An instance of a static local class can be created in either way, by calling the `createMap()` method as shown at (12) and (13). As might be expected, no outer object is involved.

As references to a local class cannot be declared outside of the local context, the functionality of the class is only available through supertype references. The method `draw()` is invoked on objects in the array at (14). The program output indicates which objects were created. In particular, note that the `final` parameter `radius` of the method `createCircle()` at (4) is accessed by the `draw()` method of the local class `Circle` at (5). An instance of the local class `Circle` is created at (11) by a call to the method `createCircle()`. The `draw()` method is invoked on this instance of the local class `Circle` in the loop at (14). The value of the `final` parameter `radius` is still accessible to the `draw()` method invoked on this instance, although the call to the method `createCircle()`, which created the instance in the first place, has completed. Values of `final` local variables continue to be available to instances of local classes whenever these values are needed.

The output also shows the actual names of the local classes. In fact, the local class names are reflected in the class filenames.

7.5 Anonymous Classes

Classes are usually first defined and then instantiated using the `new` operator. Anonymous classes combine the process of definition and instantiation into a single step. Anonymous classes are defined at the location they are instantiated, using additional syntax with the `new` operator. As these classes do not have a name, an instance of the class can only be created together with the definition.

An anonymous class can be defined and instantiated in contexts where a reference can be used, i.e. as expressions that evaluate to a reference denoting an object. Anonymous classes are typically used for creating objects “on the fly” in contexts such as the return value of a method, or as an argument in a method call, or in initialization of variables. Anonymous classes can also be used to extend *adapter classes* (Section 14.4, p. 429).

The context determines whether the anonymous class is static, and the keyword `static` is not used explicitly. For example, an anonymous class as the return value of a static method would be static, as it would be if it was used to initialize a static member variable.

Extending an Existing Class

The following syntax can be used for defining and instantiating an anonymous class that extends an existing class specified by `<superclass name>`:

```
new <superclass name> (<optional argument list>) { <class declarations> }
```

Optional arguments can be specified, which are passed to the superclass constructor. Thus, the superclass must provide a constructor corresponding to the arguments passed. Since an anonymous class cannot define constructors (as it does not have a name), an instance initializer can be used to achieve the same effect as a constructor. No `extends` clause is used in the construct.

Example 7.9 Defining Anonymous Classes

```
interface IDrawable {                               // (1)
    void draw();
}
class Shape implements IDrawable {                 // (2)
    public void draw() { System.out.println("Drawing a Shape."); }
}
class Painter {                                    // (3) Top-level Class
    public Shape createShape() {                   // (4) Non-static Method
        return new Shape() {                      // (5) Extends superclass
            public void draw() { System.out.println("Drawing a new Shape."); }
        };
    }
}
```

```

    public static IDrawable createIDrawable() { // (7) Static Method
        return new IDrawable(){              // (8) Implements interface
            public void draw() {
                System.out.println("Drawing a new IDrawable.");
            }
        };
    }
}

public class Client {
    public static void main(String args[]) { // (9)
        IDrawable[] drawables = {           // (10)
            new Painter().createShape(),     // (11) non-static anonymous class
            Painter.createIDrawable(),       // (12) static anonymous class
            new Painter().createIDrawable()  // (13) static anonymous class
        };
        for (int i = 0; i < drawables.length; i++) // (14)
            drawables[i].draw();

        System.out.println("Anonymous Class Names:");
        System.out.println(drawables[0].getClass()); // (15)
        System.out.println(drawables[1].getClass()); // (16)
    }
}

```

Output from the program:

```

Drawing a new Shape.
Drawing a new IDrawable.
Drawing a new IDrawable.
Anonymous Class Names:
class Painter$1
class Painter$2

```

.....

Class definitions from Example 7.9, which is an adaptation of Example 7.8 to anonymous classes, are shown below. The instance method `createShape()` at (4) defines a non-static anonymous class at (5), which extends the superclass `Shape`. The anonymous class overrides the inherited method `draw()`. As references to an anonymous class cannot be declared, the functionality of the class is only available through superclass references. Usually it makes sense to either override methods from the superclass or implement abstract methods from the superclass. Any other members in the definition of an anonymous class cannot be accessed.

```

// ...
class Shape implements IDrawable { // (2)
    public void draw() { System.out.println("Drawing a Shape."); }
}

class Painter { // (3) Top-level Class
    public Shape createShape() { // (4) Non-static Method
        return new Shape(){ // (5) Extends superclass
            public void draw() { System.out.println("Drawing a new Shape."); }
        };
    }
}

```

```

    }
    // ...

}
// ...

```

Implementing an Interface

The following syntax can be used for defining and instantiating an anonymous class that implements an interface specified by *<interface name>*:

```
new <interface name> () { <class declarations> }
```

An anonymous class provides a single interface implementation, and no arguments are passed. The anonymous class implicitly extends the `Object` class. Note that no `implements` clause is used in the construct.

An anonymous class implementing an interface is shown below. Details can be found in Example 7.9. The static method `createIDrawable()` at (7) defines a static anonymous class at (8), which implements the interface `IDrawable` by providing an implementation of the method `draw()`. The functionality of objects of an anonymous class which implements an interface is available through references of the interface type and the `Object` type.

```

interface IDrawable {                                // (1)
    void draw();
}
// ...
class Painter {                                     // (3) Top-level Class
    // ...
    public static IDrawable createIDrawable() { // (7) Static Method
        return new IDrawable(){                 // (8) Implements interface
            public void draw() {
                System.out.println("Drawing a new IDrawable.");
            }
        };
    }
}
// ...

```

Instantiating Anonymous Classes

The discussion on instantiating local classes (Example 7.8) is also valid for instantiating anonymous classes. The class `Client` in Example 7.9 creates one instance at (11) of the non-static anonymous class defined at (5), and two instances at (12) and (13) respectively of the static anonymous class defined at (8). The program output shows the polymorphic behavior and the runtime types of the objects. Similar to a non-static local class, an instance of a non-static anonymous class has an instance of its enclosing class at (11). An enclosing instance is not mandatory for creating objects of a static anonymous class, as shown at (12).

The names of the anonymous classes at runtime are also shown in the program output. They are also the names used to designate their respective class files. Anonymous classes are not so anonymous after all.

Access Rules for Anonymous Classes

Access rules for anonymous classes are the same as for local classes. Example 7.10 is an adaptation of Example 7.7, and illustrates the access rules for anonymous classes. Non-static anonymous classes can access all members in their enclosing context, and any final variables in their local scope. In fact, inside the definition of a non-static anonymous class, objects of the enclosing context can be referenced using the *<enclosing class name>.this* construct. As for static anonymous classes, they can only access static members in the enclosing context, and any final variables in their local scope.

Example 7.10 defines a non-static anonymous class (which extends the superclass SuperB) at (1), whose instance is assigned to the instance variable *b* when the class TopLevelA is instantiated. A static anonymous class (which also extends the superclass SuperB) is defined at (9) as the return value of the static method *staticMethod()*. The example illustrates all the significant cases involving access in anonymous classes.

Like local classes, anonymous classes cannot have static members, and they cannot specify any accessibility modifiers. This is shown at (2) and (10).

.....

Example 7.10 Access in Anonymous Classes

```

class SuperB {
    protected double x;
    protected static int n;
}

class SuperC {
    protected double y;
    protected static int m;
}

class TopLevelA extends SuperC { // Top-level Class
    private double z;
    private static int p;
    SuperB b = new SuperB() { // (1) Non-static anonymous class
        // static double d; // (2) Not OK. Only non-static members allowed.
        double zz = TopLevelA.this.z; // (3) non-static from enclosing class.
        int pp = p; // (4) static from enclosing class.
        double yy = y; // (5) inherited by enclosing class.
        int mm = m; // (6) static from enclosing class.
        double xx = this.x; // (7) non-static inherited from superclass.
        int nn = this.n; // (8) static from superclass.
    };
};

```

```

static SuperB staticMethod(final int i) { // Static Method
    final int j = 10;
    int k;
    return new SuperB() { // (9) Static anonymous class.
        // static double d; // (10) Not OK. Only non-static members allowed.
        int ii = i; // (11) final from enclosing method.
        int jj = j; // (12) final from enclosing method.
        // double kk = k; // (13) Not OK. Non-final from enclosing method.
        // double zz = TopLevelA.this.z; // (14) Not OK. Non-static member.
        int pp = p; // (15) static from enclosing class.
        // double yy = y; // (16) Not OK. Non-static member.
        int mm = m; // (17) static from enclosing class.
        double xx = this.x; // (18) non-static inherited from superclass.
        int nn = this.n; // (19) static from superclass.
    };
}
}

```



Review questions

7.5 Which of the following statements are true?

Select all valid answers.

- (a) Non-static inner classes must have either default or public accessibility.
- (b) All nested classes can contain other top-level nested classes.
- (c) Methods in all nested classes can be declared static.
- (d) All nested classes can be declared static.
- (e) Top-level nested classes can contain non-static methods.

7.6 Given the declaration

```
interface IntHolder { int getInt(); }
```

which of the following methods are valid?

```

//----(1)----
IntHolder makeIntHolder(int i) {
    return new IntHolder() {
        public int getInt() { return i; }
    };
}

//----(2)----
IntHolder makeIntHolder(final int i) {
    return new IntHolder {
        public int getInt() { return i; }
    };
}

//----(3)----
IntHolder makeIntHolder(int i) {
    class MyIH implements IntHolder {
        public int getInt() { return i; }
    }
}

```

```

        return new MyIH();
    }
//----(4)----
    IntHolder makeIntHolder(final int i) {
        class MyIH implements IntHolder {
            public int getInt() { return i; }
        }
        return new MyIH();
    }
//----(5)----
    IntHolder makeIntHolder(int i) {
        return new MyIH(i);
    }
    static class MyIH implements IntHolder {
        final int j;
        MyIH(int i) { j = i; }
        public int getInt() { return j; }
    }

```

Select all valid answers.

- (a) The method labeled (1)
- (b) The method labeled (2)
- (c) The method labeled (3)
- (d) The method labeled (4)
- (e) The method labeled (5)

7.7 Which of these statements are true?

Select all valid answers.

- (a) You cannot declare static members within a non-static inner class.
- (b) If a non-static inner class is nested within a class named `Outer`, then methods within the non-static inner class must use the prefix `Outer.this` to access the members of the class `Outer`.
- (c) All member variables in any nested class must be declared `final`.
- (d) Anonymous classes cannot have constructors.
- (e) If `objRef` is an instance of any nested class within the class `Outer`, then `(objRef instanceof Outer)` would yield `true`.

7.8 Which of the following statements are true?

Select all valid answers.

- (a) Package member classes can be declared `static`.
- (b) Classes declared as members of top-level classes can be declared `static`.
- (c) Local classes can be declared `static`.
- (d) Anonymous classes can be declared `static`.
- (e) No classes can be declared `static`.



Chapter summary

The following information was included in this chapter:

- Categories of nested classes: top-level nested classes and interfaces, non-static inner classes, local classes, anonymous classes.
- The following aspects pertaining to nested classes and interfaces are discussed:
 - The context in which they can be defined.
 - What accessibility modifiers are valid for such classes and interfaces.
 - Whether an instance of the outer context is associated with an instance of the nested class.
 - What entities in its outer context a nested class or interface can access.
 - Whether both static and non-static members can be defined in a nested class.
- Importing and using nested classes and interfaces.
- Accessing members in the outer context, using `<enclosing class name>.this` syntax.
- Instantiating instances of nested classes, using `<enclosing object reference>.new` syntax.
- Discussion of the inheritance and containment hierarchies of nested classes.
- Implementing anonymous classes by extending an existing class and by implementing an interface.



Programming exercise

- 7.1** Create a new program with a nested class named `PrintFunc` that extends the `Print` class in Exercise 6.2. In addition to just printing the value, `PrintFunc` should first apply a `Function` object on the value. The `PrintFunc` class should have a constructor that takes an instance of `Function` as a parameter. The `evaluate()` method of `PrintFunc` should use the `Function` object on its argument. The `evaluate()` method should print and return the result. The `evaluate()` method in superclass `Print` should be used to print the value.

Make the program behave just like the program in Exercise 6.2, but now using `PrintFunc` instead of `Print`.