Nested Type Declarations 9

Chapter Topics

- Categorizing nested classes:
  - ❍ Static member type which can be a class, an enum type, a record class, or an interface
  - ❍ Inner class which can be a non-static member class, a local class, or an anonymous class
  - ❍ Static local type which can be an interface, an enum type, or a record class
- Understanding the salient aspects of nested types:
  - ❍ The context in which they can be defined
  - ❍ Which accessibility modifiers are valid for nested types
  - ❍ Whether an instance of the enclosing context is associated with an instance of the nested class
  - ❍ Which entities a nested type can access in its enclosing context and in its inheritance hierarchy
  - ❍ Whether both static and non-static members can be defined in a nested type
- Importing and using nested types
- Instantiating non-static member classes using the *enclosing_object_reference* `.new` syntax
- Accessing members in the enclosing context of inner classes using the *enclosing_class_name* `.this` syntax
- Implementing anonymous classes by extending an existing class or by implementing an interface

Java SE 17 Developer Exam Objectives	
[3.1] Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection ❍ *Only nested types are covered in this chapter.* ❍ *For object lifecycle and garbage collection, see* **Chapter 10**, **p. 531**.	**§9.1**, **p. 491** *to* **§9.6**, **p. 521**

Java SE 11 Developer Exam Objectives

[3.1] Declare and instantiate Java objects including nested class objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection) | §9.1, p. 491 to §9.6, p. 521

❍ *Only nested types are covered in this chapter.*

❍ *For object lifecycle and garbage collection, see* **Chapter 10**, **p. 531**.

This chapter covers the different kinds of nested type declarations—that is, type declarations that can be declared inside a language construct such as a class, an interface, or even a method. In particular, we look at declaring, instantiating, and using such types. Since they are nested, we consider the rules for accessing entities in their enclosing context and in their inheritance hierarchy. Please hold on, as we first introduce the terminology for these nested types.

9.1 Overview of Nested Type Declarations

A *type declaration* allows a *new reference type* to be defined. A type declaration can either be a *top-level type declaration* or a *nested type declaration*. **Figure 9.1** gives an overview of the different kinds of type declarations that can be defined in Java.

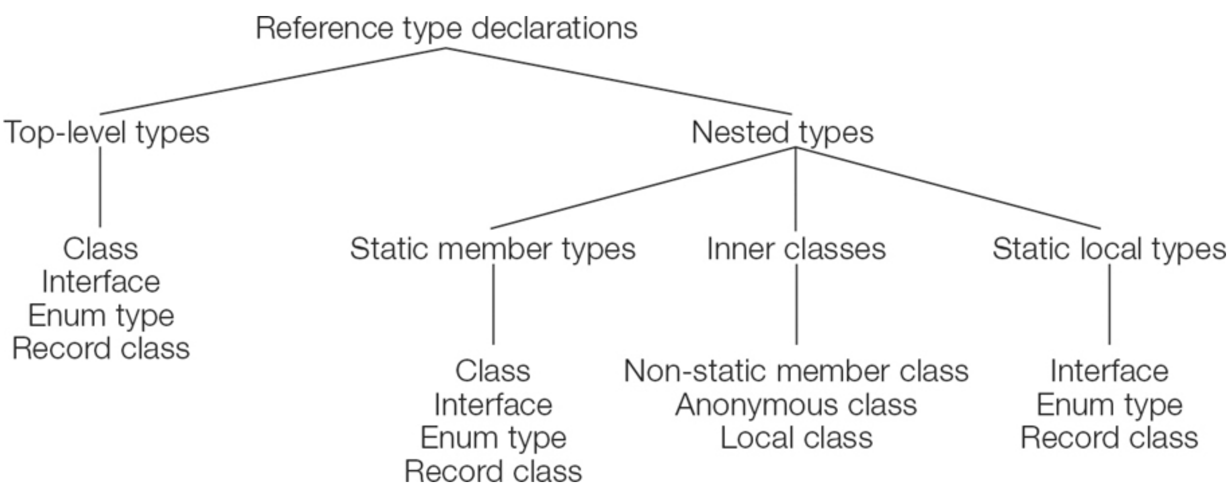

Figure 9.1 *Overview of Type Declarations*

A *top-level type declaration* is a type declaration that is *not* defined inside another type declaration. A top-level type declaration can be any one of the following: a *top-level class*, a *top-level interface*, a *top-level enum type*, or a *top-level record class*. We invariably use the shorter names *class*, *interface*, *enum type*, and *record class* when it is clear we are not referring to their nested counterparts.

A *nested type declaration* is a type declaration that is defined inside another declaration. There are three categories of nested types:

- Static member types (a.k.a. *static nested types*)

- Inner classes (a.k.a. *non-static nested classes*)
- Static local types (a.k.a. *static local nested types*)

As the name implies, *static member types* can be declared as a `static` *member* of either a top-level type or a nested type declaration. There are four kinds of static member types:

- Static member class (a.k.a. *static nested class*)
- Static member interface (a.k.a. *nested interface*)
- Static member enum type (a.k.a. *nested enum type*)
- Static member record class (a.k.a. *nested record class*)

A `static` member class or record class can be instantiated like any ordinary top-level class or record class, using its *qualified name* when calling the constructor with the `new` operator. It is not possible to instantiate an enum type or an interface.

Inner classes are non-`static` nested classes. There are three kinds of inner classes:

- Non-`static` member class
- Local class
- Anonymous class

Inner classes differ from `static` member classes in one important aspect: An instance of an inner class has an instance of the enclosing class (called the *immediately enclosing instance*) associated with it when the inner class is declared in a non-static context. An instance of an inner class can access the members of its enclosing instance by their simple names.

Non-static member classes are inner classes that are defined as instance members of other type declarations, just as fields and instance methods are defined in a class.

Local (normal) classes are non-`static` inner classes that can be defined in a block—a local block, a method body, an initializer block—both in static and non-static context, just as local variables can be defined in a block.

Static local types are defined in a block, but are *implicitly* `static` as opposed to local classes that are non-`static`. A static local type can be defined both in static and non-static context, just as local classes can be defined in a block. There are three kinds of static local types:

- Static local interface
- Static local enum type
- Static local record class

Anonymous classes are inner classes that can be defined as expressions and in expression statements, both in static and non-static context, and instantiated *on the fly*.

In **Figure 9.1** we see that there are four kinds of *nested classes* (static member classes, non-`static` member classes, local classes, anonymous classes), two kinds of *nested interfaces* (static member interfaces, static local interfaces), two kinds of *nested enum types* (static member enum types, static local enum types), and two kinds of *nested record classes* (static member record classes, static local record classes)—all defined by the context in which these nested types are declared.

Given the terminology introduced for nested types in **Figure 9.1**, a *member type declaration* can be any one of the following nested types: a `static` member type (class, interface, enum type, record class) or a non-`static` member class. Note that local classes, anonymous classes, local interfaces, local enum types, and local record classes are *not* member type declarations, as they *cannot* be declared as a *member* of a type declaration.

Example 9.1 *Overview of Type Declarations*

Click here to view code image

```
class TLC {                          // (1) Top-level class

  // Static member types:
  static class     SMC {}            // (2) Static member class
         interface SMI {}            // (3) Static member interface
         enum      SME {}            // (4) Static member enum
         record    SMR() {}          // (5) Static member record

  // Non-static member class:
  class NSMC {}                      // (6) Inner class

  // Local types in non-static context (analogous for static context):
  void nsm() {                       // Non-static method
    class     LC {}                  // (7) Local class (inner class)
    interface SLI {}                 // (8) Static local interface
    enum      SLE {}                 // (9) Static local enum
    record    SLR() {}               // (10) Static local record
  }

  // Anonymous classes (here defined as initializer expressions):
         SMC nsf = new SMC() {};  // (11) Inner class in non-static context
  static SMI  sf = new SMI() {};  // (12) Inner class in static context
}
```

Skeletal code for nested types is shown in Example 9.1. Table 9.1 presents a summary of various aspects relating to nested types. Subsequent sections on each nested type elaborate on the summary presented in this table. (*N/A* in the table means “*not applicable*”.)

- The *Type* column lists the different kinds of types that can be declared.
- The *Declaration context* column lists the lexical context in which a type can be declared.
- The *Access modifiers* column indicates what access can be specified for the type.
- The *Enclosing instance* column specifies whether an enclosing instance is associated with an instance of the type.
- The *Direct access to enclosing context* column lists what is directly accessible in the enclosing context from within the type.

Generic nested classes and interfaces are discussed in §11.13, p. 633. It is not possible to declare a generic enum type (§11.13, p. 635).

Locks on nested classes are discussed in §22.4, p. 1391.

Nested types can be regarded as a form of encapsulation, enforcing relationships between types by greater proximity. They allow structuring of types and a special binding relationship between a nested object and its enclosing instance. Used judiciously, they can be beneficial, but unrestrained use of nested types can easily result in unreadable code.

A word about the examples in this chapter: They are concocted to illustrate various aspects of nested types, and are not solutions to any well-defined or meaningful problems.

Table 9.1 *Various Aspects of Type Declarations*

Type	Declaration context	Access modifiers	Enclosing instance	Direct access to enclosing context
Top-level class, interface, enum type, or record class *(Top-level types)*	Package	`public` or package access	No	N/A

Type	Declaration context	Access modifiers	Enclosing instance	Direct access to enclosing context
Static member class, interface, enum type, or record class (*Static member types*)	As `static` member of a top-level type or a nested type	All, except when declared in interfaces whose member type declarations are implicitly `public`	No	Static members in enclosing context
Non-`static` member class (*Inner class*)	As non-`static` member of a top-level type or a nested type	All	Yes	All members in enclosing context
Local class (*Inner class*)	In block with non-static context	None	Yes	All members in enclosing context plus `final` or effectively `final` local variables
	In block with static context	None	No	Static members in enclosing context plus `final` or effectively `final` local variables
Anonymous class (*Inner class*)	As expression in non-static context	None	Yes	All members in enclosing con-

Type	Declaration context	Access modifiers	Enclosing instance	Direct access to enclosing context
	As expression in static context	None	No	Static members in enclosing context plus `final` or effectively `final` local variables
Local interface, enum type, or record class (*Static local types*)	In block with static and non-static context	None	No	Static members in enclosing context

9.2 Static Member Types

Declaring Static Member Types

Static member types can be declared in top-level type declarations, or within other nested types. For all intents and purposes, a `static` member type is very much like a top-level type.

A static member class, enum type, record class, or *interface* has the same declarations as those allowed in a top-level class, enum type, record class, or interface type, respectively. A `static` member class is declared with the keyword `static`, except when declared in an interface where a `static` member class is considered implicitly `static`. Static member enum types, record classes and interfaces are considered implicitly `static`, and the keyword `static` can be omitted.

Any access level (`public`, `protected`, package, `private`) can be specified for a `static` member type, except when declared in interfaces, where `public` access is implied for member type declarations.

Although the discussion in this section is primarily about static member classes and interfaces, it is also applicable to static member enum types and record classes.

In **Example 9.2**, the top-level class `ListPool` at (1) declares the `static` member class `MyLinkedList` at (2), which in turn defines a `static` member interface `ILink` at (3) and a `static` member class `BiNode` at (4). The `static` member class `BiNode` at (4) implements the `static` member interface `IBiLink` declared at (7). Note that each `static` member class is defined as `static`, just like `static` variables and methods in a top-level class.

In **Example 9.2**, an attempt to declare the `static` member class `Traversal` with `private` access at (8) in the interface `IBiLink` results in a compile-time error, as only `public` access is permitted for interface members. Since the class `BiTraversal` at (9) is defined in an interface; it is implicitly `public` and `static`, and *not* a non-`static` member class with package access.

The `static` member class `SortCriteria` at (11) in the non-`static` member class `SortedList` is allowed, as a `static` member type can be declared in non-static context.

Example 9.2 *Static Member Types*

Click here to view code image

```
// File: ListPool.java
package smc;

public class ListPool {                              // (1) Top-level class

  public static class MyLinkedList {                 // (2) Static member class

    private interface ILink { }                      // (3) Static member interface

    public static class BiNode                       // (4) Static member class
                  implements IBiLink {

      public static void printSimpleName() {         // (5) Static method
        System.out.println(BiNode.class.getSimpleName());
      }

      public void printName() {                      // (6) Instance method
        System.out.println(this.getClass().getName());
      }
    } // end BiNode
  } // end MyLinkedList

  interface IBiLink
            extends MyLinkedList.ILink {             // (7) Static member interface
//  private static class Traversal { }               // (8) Compile-time error!
```

```
                                                    //     Can only be public.

      class BiTraversal { }                         // (9) Class is public and static
    } // end IBiLink

    public class SortedList {                       // (10) Non-static member class
      private static class SortCriteria {}          // (11) Static member class
    }
  }
```

Click here to view code image

```
  // File: MyBiLinkedList.java
  package smc;

  public class MyBiLinkedList implements ListPool.IBiLink {       // (12)

    public static void main(String[] args) {
      ListPool.MyLinkedList.BiNode.printSimpleName();             // (13)
      ListPool.MyLinkedList.BiNode node1
               = new ListPool.MyLinkedList.BiNode();              // (14)
      node1.printName();                                          // (15)

  //  ListPool.MyLinkedList.ILink ref;                    // (16) Compile-time error!
    }
  }
```

Output from the program:

Click here to view code image

```
BiNode
smc.ListPool$MyLinkedList$BiNode
```

Using Qualified Name of Nested Types

The *qualified name* of a (static or non-`static`) member type includes the names of the enclosing types it is lexically nested in—that is, it associates the member type with its enclosing types. In **Example 9.2**, the qualified name of the `static` member class `BiNode` at (4) is `ListPool.MyLinkedList.BiNode`. The qualified name of the nested interface `IBiLink` at (7) is `ListPool.IBiLink`, determined by the lexical nesting of the types. Each member class or interface is uniquely identified by this naming syntax, which is a generalization of the naming scheme for packages. The qualified name can be used in exactly the same way as any other top-level class or interface name, as shown at (12) and (13). Such a member's *fully qualified name* is its qualified name prefixed by the name of its package. For example, the fully qualified name of the `static` member class at (4) is `smc.ListPool.MyLinkedList.BiNode`. Note that a nested member type cannot have the same name as an enclosing type.

If the source file `ListPool.java` containing the declarations in **Example 9.2** is compiled, it will result in the generation of the following class files in the package `smc`, where each class file corresponds to either a class or an interface declaration in the source file:

Click here to view code image

```
ListPool$IBiLink$BiTraversal.class
ListPool$IBiLink.class
ListPool$MyLinkedList$BiNode.class
ListPool$MyLinkedList$ILink.class
ListPool$MyLinkedList.class
ListPool$SortedList.class
ListPool.class
```

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

Within the scope of its top-level type, a `static` member type can be referenced regardless of its access modifier and lexical nesting, as shown at (7) in **Example 9.2**. Although the interface `MyLinkedList.ILink` has `private` access, it is accessible at (7), outside its enclosing class. Its access modifier (and that of the types making up its qualified name) comes into play when it is referenced by an external client. The declaration at (16) in **Example 9.2** will not compile because the member interface `ListPool.MyLinkedList.ILink` has `private` access.

Instantiating Static Member Classes

A `static` member class can be instantiated without first creating an instance of the enclosing class. **Example 9.2** shows a client creating an instance of a `static` member class at (14) using the `new` operator and the qualified name of the class. Not surprisingly, the compiler will flag an error if any of the types in the qualified name are not accessible by an external client.

Click here to view code image

```
ListPool.MyLinkedList.BiNode objRef1
         = new ListPool.MyLinkedList.BiNode();              // (14)
```

External clients must use the (fully) qualified name of a `static` member class in order to access such a class.

A `static` member class is loaded and initialized when the types in its enclosing context are loaded at runtime. Analogous to top-level classes, nested `static` members can always be accessed by the qualified name of the class, and no instance of the enclosing type is required, as shown at (13) where the full class name is used to invoke the `static` method `printSimpleName()` at (5) in the `static` member class `BiNode`. At (15), the reference `node1` is used to invoke the instance method `print-Name()` at (6) in an instance of the `static` member class `BiNode`. An instance of a `static` member class can exist independently of any instance of its enclosing class.

Importing Static Member Types

There is seldom any reason to import nested types from packages. It would undermine the encapsulation achieved by such types. However, a compilation unit can use the import facility to provide a shortcut for the names of member types. Note that type import and static import of `static` member types are equivalent. Type import can be used to import the `static` member type as a type name, and static import can be used to import the `static` member type as the name of a `static` member.

Usage of the (`static`) `import` declaration for `static` member classes is illustrated in **Example 9.3**. In the file `Client1.java`, the `import` statement at (1) allows the `static` member class `BiNode` to be referenced as `MyLinkedList.BiNode` at (2), whereas in the file `Client2.java`, the static import at (3) allows the same class to be referenced using its simple name, as at (4). At (5), the fully qualified name of the `static` member interface is used in an `implements` clause. However, in **Example 9.2** at (5), the interface `smc.ListPool.IBiLink` is declared with package access in its enclosing class `ListPool` in the package `smc`, and therefore is not visible in other packages, including the default package.

Example 9.3 *Importing Static Member Types*

Click here to view code image

```
// File: Client1.java
import smc.ListPool.MyLinkedList;                          // (1) Type import

public class Client1 {
  MyLinkedList.BiNode objRef1 = new MyLinkedList.BiNode(); // (2)
}
```

Click here to view code image

```
// File: Client2.java
import static smc.ListPool.MyLinkedList.BiNode;         // (3) Static import

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

//class BiListPool implements smc.ListPool.IBiLink { }  // (5) Compile-time error!
                                                         // Not accessible!
```

Accessing Members in Enclosing Context

Static member classes do not have a `this` reference, as they do not have any notion of an enclosing instance. This means that *any* code in a `static` member class can *only* directly access `static` members in its enclosing context. Trying to access any instance members directly in its enclosing context results in a compile-time error.

Figure 9.2 is a class diagram that illustrates static member classes and interfaces. These are shown as members of the enclosing context, with the `{static}` tag to indicate that they are `static` members. 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: Class members can be declared with any of the four access levels (`public`, `protected`, package, `private`), but interface members are always implicitly `public`. The classes from the diagram are implemented in **Example 9.4**.

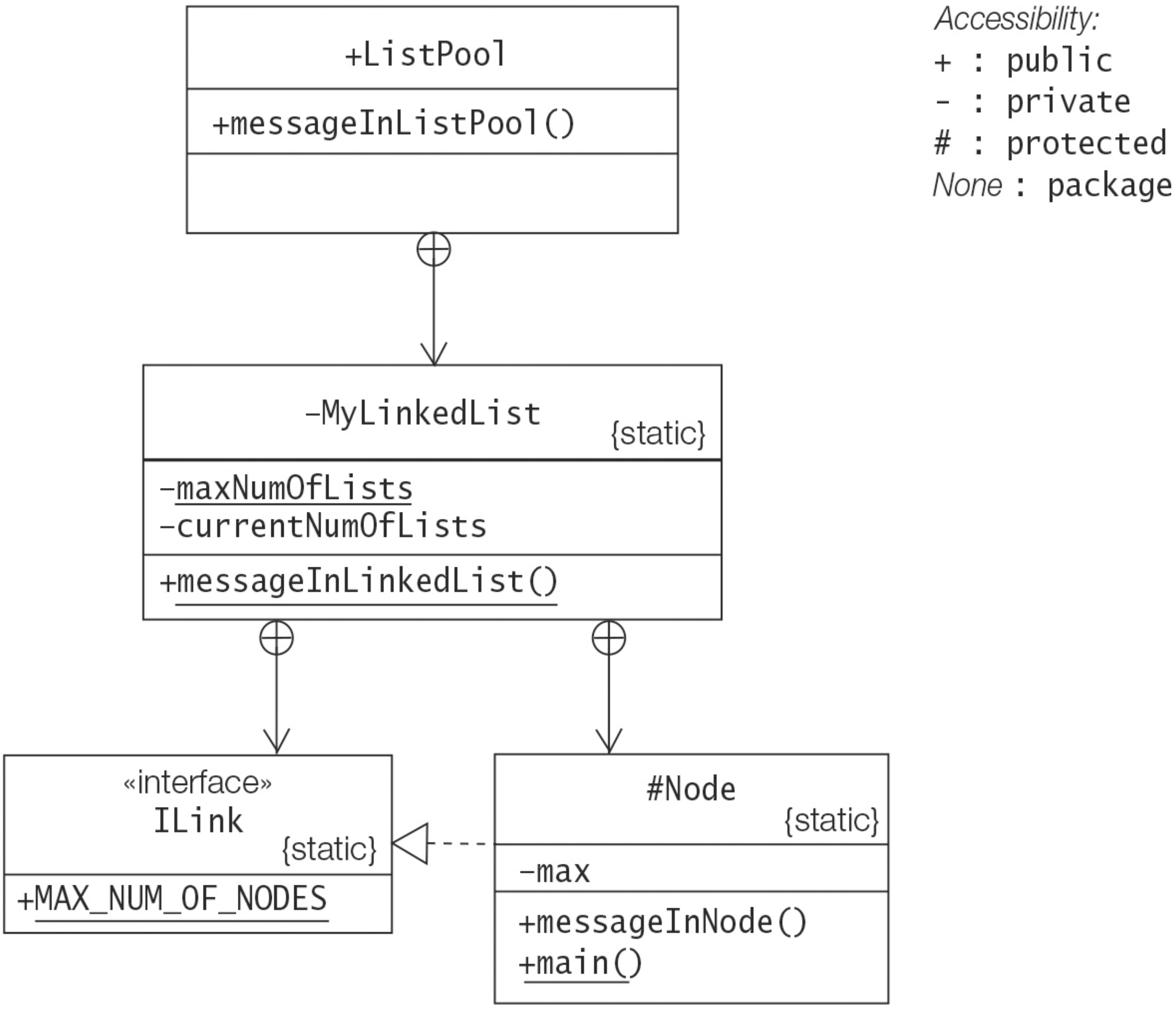

Figure 9.2 *Static Member Classes and Interfaces*

Example 9.4 *Accessing Members in Enclosing Context (Static Member Classes)*

Click here to view code image

```
// File: ListPool.java
public class ListPool {                              // Top-level class

  public void messageInListPool() {                  // Instance method
    System.out.println("This is a ListPool object.");
  }

  private static class MyLinkedList {                // (1) Static member class
    private static int maxNumOfLists = 100;          // Static field
    private int currentNumOfLists;                   // Instance field

    public static void messageInLinkedList() {       // Static method
      System.out.println("This is MyLinkedList class.");
    }

    interface ILink { int MAX_NUM_OF_NODES = 2000; }// (2) Static member interface

    protected static class Node implements ILink {  // (3) Static member class
```

```
      private int max = MAX_NUM_OF_NODES;          // (4) Instance field

      public void messageInNode() {                // Instance method
//      int currentLists = currentNumOfLists;// (5) Not OK. Access instance field
                                           //             in outer class

        int maxLists = maxNumOfLists;     // Access static field in outer class
        int maxNodes = max;               // Access instance field in member class

//      messageInListPool();    // (6) Not OK. Call instance method in outer class
        messageInLinkedList();  // (7) Call static method in outer class
      }

      public static void main(String[] args) {
        int maxLists = maxNumOfLists;    // (8) Access static field in outer class
//      int maxNodes = max;   // (9) Not OK. Access instance field in member class
        messageInLinkedList();// (10) Call static method in outer class

      }
    }  // Node
  }  // MyLinkedList
} // ListPool
```

Compiling and running the program:

Click here to view code image

```
>javac ListPool.java
>java ListPool$MyLinkedList$Node
This is MyLinkedList class.
```

Example 9.4 demonstrates direct access of members in the enclosing context of the `static` member class `Node` defined at (3). The class `Node` implements the `static` member interface `ILink` declared at (2). The initialization of the instance field `max` at (4) is valid, since the field `MAX_NUM_OF_NODES`, defined in the outer interface `ILink` at (2), is implicitly `static`. The compiler will flag an error at (5) and (6) in the instance method `messageInNode()` because direct access to instance members in the enclosing context is not permitted by *any* method in a `static` member class. It will also flag an error at (9) in the method `main()` because a `static` method cannot directly access instance fields in its own class. The statements at (8) and (10) can directly access `static` members in the enclosing context. The references in these two statements can also be specified using qualified names.

Click here to view code image

```
int maxLists = ListPool.MyLinkedList.maxNumOfLists;            // (8')
ListPool.MyLinkedList.messageInLinkedList();                   // (10')
```

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

Example 9.4 also illustrates that `static` member types when declared as class members can have any access level. In the class `ListPool`, the `static` member class `MyLinkedList` at (1) has `private` access, whereas its `static` member interface `ILink` at (2) has package access and its `static` member class `Node` at (3) has `protected` access.

The class `Node` defines the method `main()` which can be executed by the following command:

Click here to view code image

```
>java ListPool$MyLinkedList$Node
```

Note that the class `Node` in the command line is specified using the qualified name of the class file minus the extension.

9.3 Non-Static Member Classes

Declaring Non-Static Member Classes

Non-`static` member classes are *inner classes*—that is, non-`static` nested classes—that are defined without the keyword `static` as instance members of either a class, an enum type, or a record class. Non-`static` member classes cannot be declared as an instance member in an interface, as a class member in an interface is implicitly `static`. Non-`static` member classes are on par with other non-`static` members defined in a reference type.

Since a non-static member class can be an instance member of a class, an enum type, or a record class, it can have any accessibility: `public`, package, `protected`, or `private`.

The compiler generates separate class files for the non-`static` member classes defined in a top-level type declaration, as it does for static member classes.

A typical application of non-`static` member classes is implementing data structures. For example, a class for linked lists could define the nodes in the list with the help of a non-`static` member class which could be declared `private` so that it was not accessible outside the top-level class, but also nodes could not exist without the list object of the enclosing class. Nesting promotes encapsulation, and the close proximity allows classes to exploit each other's capabilities.

Example 9.5 illustrates nesting and use of non-`static` member classes, and is in no way meant to be a complete implementation for linked lists. The class `MyLinkedList` at (1) defines a non-`static` member class `Node` at (5). The class `Node` has `public` access.

Example 9.5 *Defining and Instantiating Non-Static Member Classes*

Click here to view code image

```
// File: ListClient.java
class MyLinkedList {                                  // (1)
  private String message = "Shine the light";        // (2)

  public Node makeNode(String info, Node next) {     // (3)
    return new Node(info, next);                      // (4)
  }

  public class Node {                                 // (5) Non-static member class
    // Static field:
    static int maxNumOfNodes = 100;                   // (6)

    // Instance fields:
    private String nodeInfo;                          // (7)
    private Node next;

    // Non-zero argument constructor:
    public Node(String nodeInfo, Node next) {         // (8)
      this.nodeInfo = nodeInfo;
      this.next = next;
    }

    // Instance methods:
    public Node getNext() { return next; }
    @Override
    public String toString() {
      return message + " in " + nodeInfo + " (" + maxNumOfNodes + ")"; // (9)
    }
  }
}
//______________________________________________________________________________
public class ListClient {                                       // (10)
  public static void main(String[] args) {                      // (11)
```

```
    MyLinkedList list = new MyLinkedList();                     // (12)
    MyLinkedList.Node node1 = list.makeNode("node1", null); // (13)
    MyLinkedList.Node node2 = list.new Node("node2", node1);// (14)
    for (MyLinkedList.Node node = node2;
         node!=null;
         node = node.getNext()) {                               // (15)
      System.out.println(node);
    }

//  MyLinkedList.Node nodeX
//                 = new MyLinkedList.Node("nodeX", node1);   // (16) Not OK.

  }
}
```

Output from the program:

Click here to view code image

```
Shine the light in node1 (100)
Shine the light in node2 (100)
```

Instantiating Non-Static Member Classes

An instance of a non-`static` member class can only exist when associated with an instance of its enclosing class. This means that an instance of a non-`static` member class must be created in the context of an instance of the enclosing class. In other words, the non-`static` member class does not provide any services; only instances of the class do.

A special form of the `new` operator (called the *qualified class instance creation expression*) is used to instantiate a non-`static` member class and associate it with the immediately enclosing object:

Click here to view code image

```
enclosing_object_reference.new non_static_member_class_constructor_call
```

The *enclosing object reference* in the instance creation expression evaluates to an instance of the immediately enclosing class in which the designated non-`static` member class is defined. A new instance of the non-`static` member class is created and associated with the indicated instance of the enclosing class. Note that the expression returns a reference value that denotes a new instance of the non-`static` member class.

It is illegal to specify the qualified name of the non-`static` member class in the constructor call, as the enclosing context is already given by the *enclosing object reference*.

In **Example 9.5**, the non-`static` method `makeNode()` at (3) in the class `MyLinkedList` illustrates how to instantiate a non-`static` member class in non-static context within the enclosing class. The non-`static` method `makeNode()` creates an instance of the non-`static` member class `Node` using the `new` operator, as shown at (4):

Click here to view code image

```
return new Node(info, next);                      // (4)
```

This creates an instance of the non-static member class `Node` in the context of the current instance of the enclosing class on which the `makeNode()` method is invoked. The `new` operator in the statement at (4) has an implicit `this` reference as the *enclosing object reference* that denotes this outer object. In the qualified class instance creation expression at (4') below, the `this` reference is explicitly specified to indicate the enclosing object:

Click here to view code image

```
return this.new Node(info, next);                 // (4')
```

The `makeNode()` method is called at (13). This method call associates an inner object of the `Node` class with the `MyLinkedList` object denoted by the reference `list`. This inner object is denoted by the reference `node1`. This reference can now be used in the normal way to access members of the inner object.

In **Example 9.5**, the declaration statement at (14) in the `main()` method illustrates how external clients can instantiate a non-`static` member class using the qualified class instance creation expression. The reference `list` at (12) denotes an object of the enclosing class `MyLinkedList`. This reference is specified in the qualified class instance creation expression, as shown at (14).

Click here to view code image

```
MyLinkedList.Node node2 = list.new Node("node2", node1); // (14)
```

After the execution of the statement at (14), the `MyLinkedList` object denoted by the `list` reference has two instances of the non-`static` member class `Node` associated with it. This is depicted in **Figure 9.3**, where the outer object (denoted by `list`) of the class `MyLinkedList` is shown with its two associated inner objects (denoted by the references `node1` and `node2`, respectively) right after the execution of the statement at (14). In other words, multiple objects of the non-`static` member classes can be associated with an object of the enclosing class at runtime.

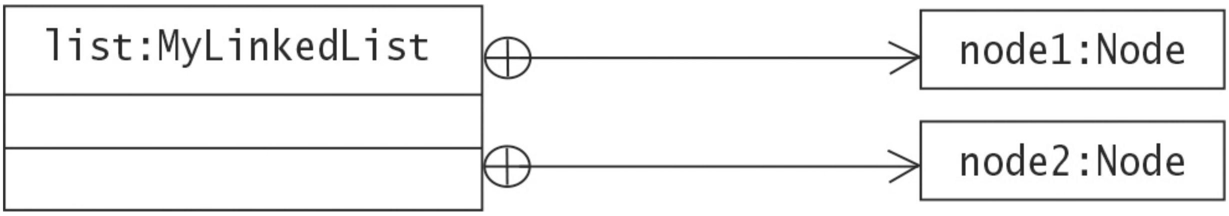

Figure 9.3 *Outer Object with Associated Inner Objects*

In **Example 9.5**, if the non-`static` method `makeNode()` at (3) in the class `MyLinkedList` is made `static`, the constructor call to the `Node` class at (4) will *not* compile. Static code in a class can only refer to other `static` members, and not to non-`static` members. A `static` method would have to provide an instance of the outer object, as would any other external client, seen here in the `static` version of the `makeNode()` method:

Click here to view code image

```
public static Node makeNode(String info, Node next) {       // (3') Static method
  return new MyLinkedList().new Node(info, next);   // (4') Explicit outer object
}
```

An example of using the inner objects is shown at (15) in the `for` loop. The print statement in the loop body calls the `toString()` method implicitly on each inner object to print its text representation.

An attempt to create an instance of the non-`static` member class using the `new` operator with the qualified name of the inner class, as shown at (16), results in a compile-time error. The full class name creation expression at (16) applies to creating instances of `static` member classes.

Accessing Members in Enclosing Context

An implicit reference to the enclosing object is always available in every method and constructor of a non-static member class. A method or constructor can explicitly specify this reference using a special form of the `this` construct and access its enclosing object, as explained in the next example.

From within a non-static member class, it is possible to refer to all members in the enclosing class directly, unless they are hidden. An example is shown at (9) in **Example 9.5**, where the instance field `message` in an object of the enclosing class is accessed by its simple name in the non-static member 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 write the statement at (9) as follows:

Click here to view code image

```
return this.message + " in " + this.nodeInfo +
                             " (" + this.maxNumOfNodes + ")";          // (9a) Not ok.
```

The reference `this.nodeInfo` is correct because the field `nodeInfo` certainly belongs to the current object (denoted by `this`) of the `Node` class, but `this.message` *cannot* possibly work, as the current object (indicated by `this`) of the `Node` class has no field named `message`. The correct syntax is the following:

Click here to view code image

```
return MyLinkedList.this.message + " in " + this.nodeInfo +
                                          " (" + this.maxNumOfNodes + ")"; // (9b)
```

The expression (called the *qualified* `this`)

```
enclosing_class_name.this
```

evaluates to a reference that denotes the enclosing object (of the specified class) that is associated with the current instance of a non-static member class.

Accessing Hidden Members

Fields and methods in the enclosing context can be *hidden* by fields and methods with the same names in the non-static member class. The qualified `this` can be used to access members in the enclosing context, somewhat analogous to using the keyword `super` in subclasses to access hidden superclass members.

Example 9.6 *Qualified* `this` *and Qualified Class Instance Creation Expression*

Click here to view code image

```
// File: OuterInstances.java
class TLClass {                                              // (1)  TLC
  private String id = "TLClass ";                            // (2)
  public TLClass(String objId) { id = id + objId; }          // (3)
  public void printId() {                                    // (4)
    System.out.println(id);
```

```
  }

  class InnerB {                                          // (5)  NSMC
    private String id = "InnerB ";                        // (6)
    public InnerB(String objId) { id = id + objId; }      // (7)
    public void printId() {                               // (8)
      System.out.print(TLClass.this.id + " : ");          // (9)  Refers to (2)
      System.out.println(id);                             // (10) Refers to (6)
    }

    class InnerC {                                        // (11) NSMC
      private String id = "InnerC ";                      // (12)
      public InnerC(String objId) { id = id + objId; }    // (13)
      public void printId() {                             // (14)
        System.out.print(TLClass.this.id + " : ");        // (15) Refers to (2)
        System.out.print(InnerB.this.id + " : ");         // (16) Refers to (6)
        System.out.println(id);                           // (17) Refers to (12)
      }
      public void printIndividualIds() {                  // (18)
        TLClass.this.printId();                           // (19) Calls (4)
        InnerB.this.printId();                            // (20) Calls (8)
        printId();                                        // (21) Calls (14)
      }
    } // InnerC
  } // InnerB
} // TLClass
//____________________________________________________________________________
public class OuterInstances {                               // (22)
  public static void main(String[] args) {                  // (23)
    TLClass a = new TLClass("a");                           // (24)
    TLClass.InnerB b = a.new InnerB("b");                   // (25) b  --> a
    TLClass.InnerB.InnerC c1 = b.new InnerC("c1");          // (26) c1 --> b
    TLClass.InnerB.InnerC c2 = b.new InnerC("c2");          // (27) c2 --> b
    b.printId();                                            // (28)
    c1.printId();                                           // (29)
    c2.printId();                                           // (30)
    System.out.println("------------");

    TLClass.InnerB bb = new TLClass("aa").new InnerB("bb");  // (31)
    TLClass.InnerB.InnerC cc = bb.new InnerC("cc");         // (32)
    bb.printId();                                           // (33)
    cc.printId();                                           // (34)
    System.out.println("------------");

    TLClass.InnerB.InnerC ccc =
      new TLClass("aaa").new InnerB("bbb").new InnerC("ccc");// (35)
    ccc.printId();                                          // (36)
    System.out.println("------------");
```

```
        ccc.printIndividualIds();                                    // (37)
    }
  }
```

Output from the program:

Click here to view code image

```
TLClass a : InnerB b
TLClass a : InnerB b : InnerC c1
TLClass a : InnerB b : InnerC c2
------------
TLClass aa : InnerB bb
TLClass aa : InnerB bb : InnerC cc
------------
TLClass aaa : InnerB bbb : InnerC ccc
------------
TLClass aaa
TLClass aaa : InnerB bbb
TLClass aaa : InnerB bbb : InnerC ccc
```

Example 9.6 illustrates the qualified `this` employed to access members in the enclosing context, and also demonstrates the qualified class instance creation expression employed to create instances of non-static member classes. The example shows the non-static member class `InnerC` at (11), which is nested in the non-static member class `InnerB` at (5), which in turn is nested in the top-level class `TLClass` at (1). All three classes have a `private` non-static `String` field named `id` and a non-`static` method named `printId`. The member name in the nested class *hides* the name in the enclosing context. These members are *not* overridden in the nested classes because no inheritance is involved. In order to refer to the hidden members, the nested class can use the qualified `this`, as shown at (9), (15), (16), (19), and (20).

Within the nested class `InnerC`, the three forms used in the following statements to access its field `id` are equivalent:

Click here to view code image

```
System.out.println(id);                    // (17)
System.out.println(this.id);               // (17a)
System.out.println(InnerC.this.id);        // (17b)
```

The `main()` method at (23) uses the special syntax of the `new` operator to create objects of non-static member classes and associate them with enclosing objects. An instance of class `InnerC` (denoted by `c1`) is created at (26) in the context of an instance of class `InnerB` (denoted by `b`), which was created at (25) in the context of an instance of class `TLClass` (denoted by `a`), which in turn was created at (24).

Click here to view code image

```
TLClass a = new TLClass("a");                         // (24)
TLClass.InnerB b = a.new InnerB("b");                 // (25) b  --> a
TLClass.InnerB.InnerC c1 = b.new InnerC("c1");        // (26) c1 --> b
```

The reference `c1` is used at (29) to invoke the method `printId()` declared at (14) in the nested class `InnerC`. This method prints the field `id` from all the objects associated with an instance of the nested class `InnerC`.

Click here to view code image

```
TLClass a : InnerB b : InnerC c1
```

When the intervening references to an instance of a non-static member class are of no interest—that is, if the reference values need not be stored in variables—the `new` operator can be chained as shown at (31) and (35).

Click here to view code image

```
TLClass.InnerB bb = new TLClass("aa").new InnerB("bb");  // (31)
...
TLClass.InnerB.InnerC ccc =
  new TLClass("aaa").new InnerB("bbb").new InnerC("ccc");// (35)
```

Note that the (outer) objects associated with the instances denoted by the references `c1`, `cc`, and `ccc` (at (26), (32), and 35), respectively) are distinct, as evident from the program output. However, the instances denoted by references `c1` and `c2` (at (26) and (27), respectively) have the same outer objects associated with them.

Inheritance Hierarchy and Enclosing Context

A non-`static` member class can extend another class and implement interfaces, as any normal class. An inherited field (or method) in a non-`static` member subclass can *hide* a field (or method) with the same name in the enclosing context. Using the simple name to access this member will access the inherited member, not the one in the enclosing context.

Example 9.7 illustrates the situation. In the inner subclass at (4), the field name `value` at (1) in the superclass hides the field with the same name in the enclosing class at (3).

In **Example 9.7**, the standard form of the `this` reference is used to access the inherited field `value`, as shown at (6). The simple name of the field would also work in this case, as would the keyword `super` with the simple name. The `super` keyword would be mandatory to access the superclass field if the inner subclass also declared a field with the same name. However, to access the member from the enclosing context, the qualified `this` must be used, as shown at (7).

Example 9.7 *Inheritance Hierarchy and Enclosing Context*

Click here to view code image

```
// File: HiddenAndInheritedAccess.java
class Superclass {
  protected int value = 3;                   // (1) Instance field in superclass
}
//________________________________________________________________________________
class TopLevelClass {                         // (2) Top-level Class
  private double value = 3.14;                // (3) Hidden by the instance field

                                              //     at (1) in the inner subclass
  class InnerSubclass extends Superclass {    // (4) Non-static member subclass
    public void printHidden() {               // (5)
      // (6) value from superclass:
      System.out.println("this.value: " + this.value);

      // (7) value from enclosing context:
      System.out.println("TopLevelClass.this.value: "
                         + TopLevelClass.this.value);
    }
  } // InnerSubclass
} // TopLevelClass
//________________________________________________________________________________
public class HiddenAndInheritedAccess {
  public static void main(String[] args) {
    TopLevelClass.InnerSubclass ref = new TopLevelClass().new InnerSubclass();
    ref.printHidden();
  }
}
```

Output from the program:

Click here to view code image

```
this.value: 3
TopLevelClass.this.value: 3.14
```

Some caution should be exercised when extending an inner class. Some of the subtleties involved are illustrated by **Example 9.8**. The nesting and the inheritance hierarchy of the classes involved are shown in **Figure 9.4**. The question that arises is how do we provide an *outer instance* when creating a *subclass instance* of a non-static member class—for example, when creating objects of the subclasses `SubInnerA` and `InnerB` in **Figure 9.4**.

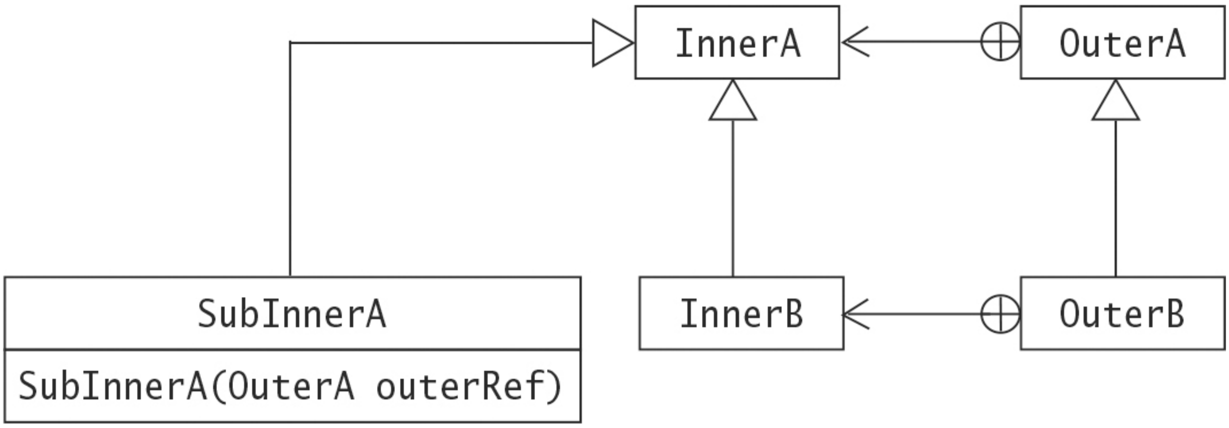

Figure 9.4 *Non-Static Member Classes and Inheritance*

Example 9.8 *Extending Inner Classes*

Click here to view code image

```
// File: Extending.java
class OuterA {                                  // (1)
  class InnerA { }                              // (2)
}
//______________________________________________________________________________
class SubInnerA extends OuterA.InnerA {         // (3) Extends NSMC at (2)

  // (4) Mandatory non-zero argument constructor:
  SubInnerA(OuterA outerRef) {
    outerRef.super();                           // (5) Explicit super() call
  }
}
//______________________________________________________________________________
class OuterB extends OuterA {                   // (6) Extends class at (1)
  class InnerB extends OuterA.InnerA { }        // (7) Extends NSMC at (2)
}
//______________________________________________________________________________
public class Extending {
  public static void main(String[] args) {

    // (8) Outer instance passed explicitly in constructor call:
    SubInnerA obj1 = new SubInnerA(new OuterA());
    System.out.println(obj1.getClass());
```

```
        // (9) No outer instance passed explicitly in constructor call to InnerB:
        OuterB.InnerB obj2 = new OuterB().new InnerB();
        System.out.println(obj2.getClass());
      }
    }
```

Output from the program:

```
class SubInnerA
class OuterB$InnerB
```

In **Example 9.8**, the non-static member class `InnerA`, declared at (2) in the class `OuterA` at (1), is extended by `SubInnerA` at (3). Note that `SubInnerA` and the class `OuterA` are not related in any way, and that the subclass `OuterB` inherits the class `InnerA` from its superclass `OuterA`. An instance of `SubInnerA` is created at (8). An instance of the class `OuterA` is explicitly passed as an argument in the constructor call to `SubInnerA`. The constructor at (4) for `SubInnerA` has a special `super()` call in its body at (5), called a *qualified superclass constructor invocation*. This call ensures that the constructor of the superclass `InnerA` has an outer object (denoted by the reference `outerRef`) to bind to. Using the standard `super()` call in the subclass constructor is not adequate because it does not provide an outer instance for the superclass constructor to bind to. The non-zero argument constructor at (4) and the `outer-Ref.super()` expression at (5) are mandatory to set up the relationships correctly between the objects involved.

The outer object problem mentioned above does not arise if the subclass that extends an inner class is also declared within an outer class that extends the outer class of the superclass. This situation is illustrated at (6) and (7): The classes `InnerB` and `OuterB` extend the classes `InnerA` and `OuterA`, respectively. The member class `InnerA` is inherited by the class `OuterB` from its superclass `OuterA`—and can be regarded as being nested in the class `OuterB`. Thus an object of class `OuterB` can act as an outer object for both an instance of class `InnerA` and that of class `InnerB`. The object creation expression `new OuterB().new InnerB()` at (9) creates an `OuterB` object and implicitly passes its reference to the default constructor of class `InnerB`. The default constructor of class `InnerB` invokes the default constructor of its superclass `InnerA` by calling `super()` and passing it the reference of the `OuterB` object, which the constructor of class `InnerA` can readily bind to.

It goes without saying that such convoluted inheritance and nesting relationships as those in **Example 9.8** hardly qualify as best coding practices.

Review Questions

9.1 Which statement is true about the following program?

Click here to view code image

```
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 correct answer.

a. The program will fail to compile because the class `Inner` cannot be declared within the class `Outer`.

b. The program will fail to compile because the method `createInner()` cannot be allowed to pass objects of the class `Inner` to methods outside the class `Outer`.

c. The program will fail to compile because the field `secret` is not accessible from the method `getSecret()`.

d. The program will fail to compile because the method `getSecret()` is not accessible from the `main()` method in the class `MyClass`.

e. The code will compile and print `123` at runtime.

9.2 Which of the following statements are true about nested classes? Select the two correct answers.

a. An instance of a static member class has an implicit outer instance.

b. A static member class can contain non- `static` fields.

c. A static member interface can contain non- `static` fields.

d. A static member interface has an implicit outer instance.

e. An instance of the outer class can be associated with many instances of a non-static member class.

9.3 Which statement is true about the following program?

Click here to view code image

```
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 the one correct answer.

a. The program will fail to compile.

b. The program will compile and print `2`, `3`, and `4,` in that order at runtime.

c. The program will compile and print `1`, `4`, and `2`, in that order at runtime.

d. The program will compile and print `1`, `3`, and `4`, in that order at runtime.

e. The program will compile and print `3`, `2`, and `1`, in that order at runtime.

9.4 Which of the following statements are true about the following program?

Click here to view code image

```
public class Outer {
  public void doIt() {}
  public class Inner {
    public void doIt() {}
  }

  public static void main(String[] args) {
    new Outer().new Inner().doIt();
  }
}
```

Select the two correct answers.

a. The `doIt()` method in the `Inner` class overrides the `doIt()` method in the `Outer` class.

b. The `doIt()` method in the `Inner` class overloads the `doIt()` method in the `Outer` class.

c. The `doIt()` method in the `Inner` class hides the `doIt()` method in the `Outer` class.

d. The qualified name of the `Inner` class is `Outer.Inner`.

e. The program will fail to compile.

9.4 Local Classes

Declaring Local Classes

A local class is an inner class that is defined in a block. This can be essentially any context where a local block or block body is allowed: a method, a constructor, an initializer block, a `try`-`catch`-`finally` construct, loop bodies, or an `if`-`else` statement. Example 9.9 shows declaration of the local class `StaticLocal` at (5) that is defined in the static context of the method `staticMethod()` at (1).

A local class cannot have any access modifier and cannot be declared `static`, as shown at (4) in Example 9.9. However, it can be declared `abstract` or `final`, as shown at (5). The declaration of the class is only accessible in the context of the block in which it is defined, subject to the same scope rules as for local variable declarations. In particular, it must be declared before use in the block. In Example 9.9, an attempt to create an object of class `StaticLocal` at (2) and use the class `Static-Local` at (3) fails, as the class has not been defined before use, but this is not a problem at (11), (12), (13), and (14).

A local class can declare members and constructors, shown from (6) to (10), as in a normal class. The members of the local class can have any access level, and are accessible in the enclosing block regardless of their access level. Even though the field `if1` at (7) is `private`, it is accessible in the enclosing method at (12).

Blocks in non-static context have a `this` reference available, which refers to an instance of the class containing the block. An instance of a local class, which is declared in such a non-`static` block, has an instance of the enclosing class associated with it. This gives such a non-`static` local class much of the same capability as a non-static member class.

However, if the block containing a local class declaration is defined in static context (i.e., a `static` method or a static initializer block), the local class is implicitly `static` in the sense that its instantiation does not require any outer object. This aspect of local classes is reminiscent of static member classes. However, note that a local class cannot be specified with the keyword `static`. The `static` method at (1) is called at (15). The local class `StaticLocal` can only be instantiated, as shown at (11), in the enclosing method `staticMethod()` and does not require any outer object of the enclosing class. Analogous to the value of a local variable, the object of the local class is not available to the caller of the method after the method completes execution, unless measures are taken to store it externally or if its reference value is returned by the call.

Example 9.9 *Declaring Local Classes*

Click here to view code image

```
// File: LocalClient1.java
class TLCWithSLClass {                                  // Top-level Class

  static void staticMethod(final int fp) {              // (1) Static Method
//   StaticLocal slRef = new StaticLocal(10);           // (2) Class cannot be resolved
//   System.out.println(StaticLocal.staticValue()); // (3) Class cannot be resolved

//   public static class StaticLocal { // (4) Not OK. Cannot be static,
                                       //               and no access modifier
```

```
    final class StaticLocal {                          // (5) Static local class
      public static final int sf1 = 10;                // (6) Static field
      private int if1;                                 // (7) Instance field
      public StaticLocal(int val) {                    // (8) Constructor
        this.if1 = val;
      }
      public int getValue() { return if1; }            // (9) Instance method
      public static int staticValue() { return sf1; }// (10) Static method
    } // end StaticLocal

    StaticLocal slRef2 = new StaticLocal(100);                           // (11)
    System.out.println("Instance field: " + slRef2.if1);                 // (12)
    System.out.println("Instance method call: " + slRef2.getValue());    // (13)
    System.out.println("Static method call: " + StaticLocal.staticValue());// (14)
  } // end staticMethod
}

public class LocalClient1 {
  public static void main(String[] args) {
    TLCWithSLClass.staticMethod(100);                                    // (15)
  }
}
```

Output from the program:

```
Instance field: 100
Instance method call: 100
Static method call: 10
```

Accessing Declarations in Enclosing Context

Declaring a local class in a static or a non-static block influences what the class can access in the enclosing context.

Accessing Local Declarations in the Enclosing Block

Example 9.10 illustrates how a local class can access declarations in its enclosing block. Example 9.10 shows declaration of the local class `NonStaticLocal` at (7) that is defined in the non-static context of the method `nonStaticMethod()` at (1).

A local class can access variables (local variables, method parameters, and `catch`-block parameters) that are declared `final` or *effectively* `final` in the scope of its local context. A variable whose value does not change after it is initialized is said to be *effectively* `final`. This situation is shown at (8) and (9) in the `NonStaticLocal` class, where the `final` parameter `fp` and the effectively `final` local variable `flv` of the method `nonStaticMethod()` are accessed. Access to local variables that are not `final` or effectively `final` is not permitted from local classes. The local variable `nfv1` at (4) is accessed at (10) in the local class, but this local variable is not effectively `final` as it is reassigned a new value at (6).

Accessing a local variable from the local context that has not been declared or has not been *definitely assigned* (§5.5, p. 232) results in a compile-time error, as shown at (11) and (12). The local variable `nflv2` accessed at (11) is not declared before use, as it is declared at (16). The local variable `nflv3` accessed at (12) is not initialized before use, as it is initialized at (17)—which means it is not definitely assigned at (12).

Declarations in the local class can *shadow* declarations in the enclosing block. The field `hlv` at (13) shadows the local variable by the same name at (3) in the enclosing method. There is no way for the local class to refer to shadowed declarations in the enclosing block.

The non-`static` method at (1) is called at (19) on an instance of its enclosing class. When the constructor at (15) in the non-`static` method is executed, the reference to this instance is passed implicitly to the constructor, thus this instance acts as the enclosing object of the local class instance.

Example 9.10 *Accessing Local Declarations in the Enclosing Block (Local Classes)*

Click here to view code image

```
// File: LocalClient2.java
class TLCWithNSLClass {                  // Top-level Class

  void nonStaticMethod(final int fp) { // (1) Non-static Method
    // Local variables:
    int flv = 10;                // (2) Effectively final local variable
    final int hlv = 20;          // (3) Final local variable (constant variable)
    int nflv1 = 30;              // (4) Non-final local variable
    int nflv3;                   // (5) Non-final local variable declaration

    nflv1 = 40;                  // (6) Not effectively final local variable

    // Non-static local class
    class NonStaticLocal {// (7)
      int f1 = fp;         // (8) Final param from enclosing method
      int f2 = flv;        // (9) Effectively final variable from enclosing method
//    int f3 = nflv1;      // (10) Not effectively final from enclosing method
//    int f4 = nflv2;      // (11) Name nflv2 cannot be resolved: use-before-decl
//    int f5 = nflv3;      // (12) Not definitely assigned
      int hlv;             // (13) Shadows local variable at (3)
```

```
      NonStaticLocal (int value) {
        hlv = value;
        System.out.println("Instance field: " + hlv);// (14) Prints value from (13)
      }
    } // end NonStaticLocal

    NonStaticLocal nslRef = new NonStaticLocal(200);// (15) Implicit outer object
    int nflv2 = 50;                                 // (16) Attempted use in (11)
    nflv3 = 60;                                     // (17) Initializes (4)
    System.out.println("Local variable: " + hlv);   // (18) Prints value from (3)
  } // end nonStaticMethod
}

public class LocalClient2 {
  public static void main(String[] args) {
    new TLCWithNSLClass().nonStaticMethod(1000);    // (19)
  }
}
```

Output from the program:

```
Instance field: 200
Local variable: 20
```

Accessing Members in the Enclosing Class

Example 9.11 illustrates how a local class can access members in its enclosing class. The top-level class `TLCWith2LCS` declares two methods: `nonStaticMethod()` and `staticMethod()`. Both methods define a local class each: `NonStaticLocal` at (1) in non-static context and `StaticLocal` at (8) in static context, both of which are subclasses of the superclass `Base`.

A local class can access members inherited from its superclass in the usual way. The field `nsf1` in the superclass `Base` is inherited by the local subclass `NonStaticLocal`. This inherited field is accessed in the `NonStaticLocal` class, as shown at (2), (3), and (4), by using the field's simple name, the standard `this` reference, and the `super` keyword, respectively. This also applies for static local classes, as shown at (9), (10), and (11).

Fields and methods in the enclosing class can be *hidden* by member declarations in the local class. The non-`static` field `nsf1`, inherited by the local classes, hides the field by the same name in the enclosing class `TLCWith2LCS`. The qualified `this` can be used in non-`static` local classes for *explicit* referencing of members in the enclosing class, regardless of whether these members are hidden or not.

[Click here to view code image]

```
double f4 = TLCWith2LCS.this.nsf1;      // (5) In enclosing object.
```

However, the special form of the `this` construct *cannot* be used in a local class that is declared in static context, as shown at (12), since it does not have any notion of an outer object. A local class in static context cannot refer to non-static members in the enclosing context.

A non-`static` local class can access both `static` and non-`static` members defined in the enclosing class. The non-`static` field `nsf2` and `static` field `sf` are defined in the enclosing class `TLCWith2LCS`. They are accessed in the `NonStaticLocal` class at (6) and (7), respectively. The special form of the `this` construct can also be used in non-`static` local classes, as previously mentioned.

However, a local class that is declared in static context can only directly access `static` members defined in the enclosing class. The `static` field `sf` in the class `TLCWith2LCS` is accessed in the `StaticLocal` class at (14), but the non-`static` field `nsf1` cannot be accessed, as shown at (13).

Example 9.11 *Accessing Members in the Enclosing Class (Local Classes)*

[Click here to view code image]

```
// File: LocalClient3.java
class Base { protected int nsf1; }     // Superclass
//______________________________________________________________________________
class TLCWith2LCS {                    // Top-level Class
  private int nsf1;                    // Non-static field
  private int nsf2;                    // Non-static field
  private static int sf;               // Static field

  void nonStaticMethod( int fp) {      // Non-static Method

    class NonStaticLocal extends Base {// (1) Non-static local subclass
      int f1 = nsf1;                   // (2) Inherited from superclass.
      int f2 = this.nsf1;              // (3) Inherited from superclass.
      int f3 = super.nsf1;             // (4) Inherited from superclass.
      int f4 = TLCWith2LCS.this.nsf1;  // (5) In enclosing object.
      int f5 = nsf2;                   // (6) Instance field in enclosing object.
      int f6 = sf;                     // (7) static field from enclosing class.
    } // NonStaticLocal

  } // nonStaticMethod
```

```
  static void staticMethod(final int fp) { // Static Method

    class StaticLocal extends Base {   // (8) Static local subclass
      int f1 = nsf1;                   // (9) Inherited from superclass.
      int f2 = this.nsf1;              // (10) Inherited from superclass.
      int f3 = super.nsf1;             // (11) Inherited from superclass.
//    int f4 = TLCWith2LCS.this.nsf1;  // (12) No enclosing object.
//    int f5 = nsf2;                   // (13) No enclosing object.
      int f6 = sf;                     // (14) static field from enclosing class.
    } // StaticLocal

  } // staticMethod
}

public class LocalClient3 {
  public static void main(String[] args) {
    TLCWith2LCS.staticMethod(200);          // (15)
    new TLCWith2LCS().nonStaticMethod(100); // (16)
  }
}
```

Instantiating Local Classes

Clients outside the scope of a local class cannot instantiate the class directly because such classes are, after all, local. A local class can be instantiated in the block in which it is defined. Like a local variable, a local class must be declared before being used in the block.

A method can return instances of any local class it declares. The local class type must then be assignable to the return type of the method. The return type cannot be the same as the local class type, since this type is not accessible outside the method. A supertype of the local class must be specified as the return type. This also means that, in order for the objects of the local class to be useful outside the method, a local class should implement an interface or override the behavior of its supertypes.

Example 9.12 illustrates how clients can instantiate local classes. The nesting and the inheritance hierarchy of the classes involved are shown in Figure 9.5. 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 `Graph` at (9) is defined in the `static` method `createGraph()` at (8), which has the return type `IDrawable`.

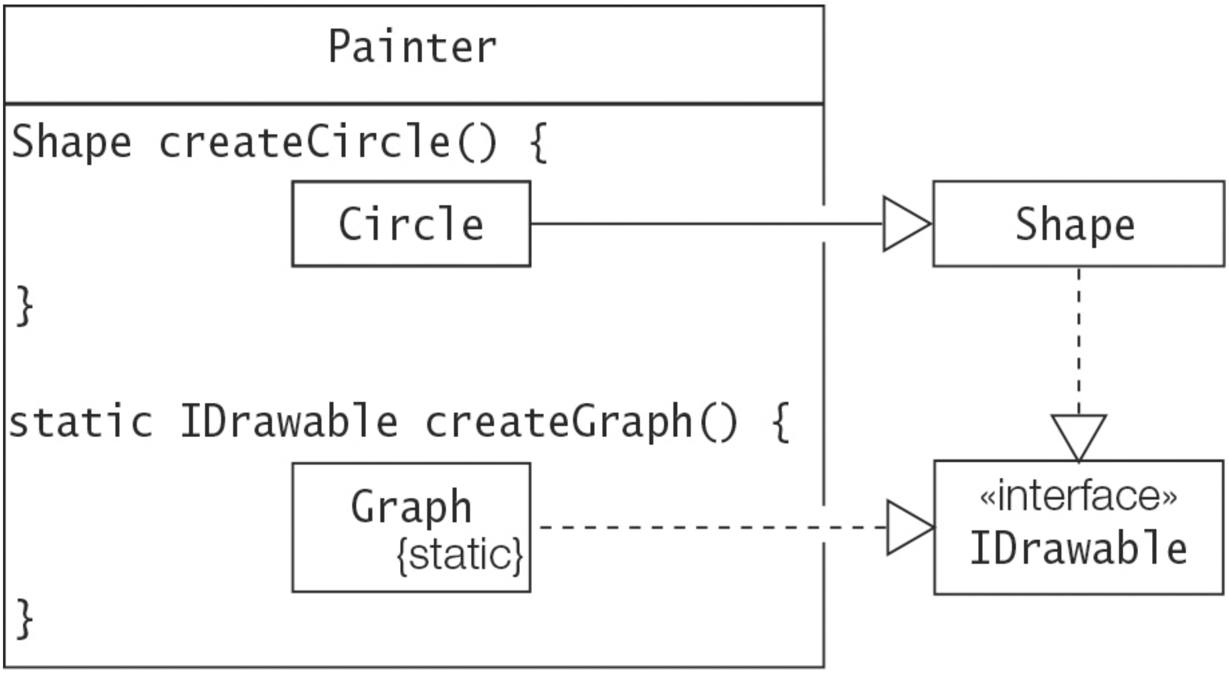

Figure 9.5 *Local Classes and Inheritance Hierarchy*

Example 9.12 *Instantiating Local Classes*

Click here to view code image

```
// File: LocalClassClient.java
interface IDrawable {                          // (1)
  void draw();
}
//_______________________________________________________________________________
class Shape implements IDrawable {             // (2)
  @Override
  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
      @Override
      public void draw() {
        System.out.println("Drawing a Circle of radius: " + radius); // (6)
      }
    }
    return new Circle();                       // (7) Passed enclosing object reference
  }

  public static IDrawable createGraph() { // (8) Static Method
    class Graph implements IDrawable {    // (9) Static local class
      @Override
      public void draw() { System.out.println("Drawing a Graph."); }
    }
    return new Graph();                        // (10) Object of static local class
```

```
    }
  }
  //____________________________________________________________________________
  public class LocalClassClient {
    public static void main(String[] args) {
      IDrawable[] drawables = {              // (11)
          new Painter().createCircle(5),     // (12) Object of non-static local class
          Painter.createGraph(),             // (13) Object of static local class
          new Painter().createGraph()        // (14) Object of static local class
      };
      for (IDrawable aDrawable : drawables) // (15)
        aDrawable.draw();

      System.out.println("Local Class Names:");
      System.out.println(drawables[0].getClass().getName());   // (16)
      System.out.println(drawables[1].getClass().getName());   // (17)
    }
  }
```

Output from the program:

Click here to view code image

```
Drawing a Circle of radius: 5.0
Drawing a Graph.
Drawing a Graph.
Local Class Names:
Painter$1$Circle
Painter$1$Graph
```

The `main()` method in **Example 9.12** creates a polymorphic array `drawables` of type `IDrawable[]` at (11), which is initialized at (12) through (14) with instances of the local classes.

Creating an instance of a non-`static` local class requires an instance of the enclosing class. In **Example 9.12**, the non-`static` method `createCircle()` is invoked on the instance of the enclosing class `Painter` to create an instance of the non-`static` local class `Circle` in the non-`static` method, as shown at (12). The reference to the instance of the enclosing class is passed implicitly in the constructor call at (7) to the non-`static` local class.

A `static` method can be invoked either through the class name or through a reference of the class type. An instance of a static local class can be created either way by calling the `createGraph()` method, as shown at (13) and (14). As might be expected in static context, no outer object is involved.

As references to a local class cannot be declared outside the local context, the functionality of the class is only available through supertype references. The method `draw()` is invoked on objects in the array `IDrawable` at (15). The program output indicates which objects were created. In particular, note that the `final` parameter `radius` of the method `createCircle()` at (4) is accessed in the `draw()` method of the local class `Circle` at (6). An instance of the local class `Circle` is created at (12) by a call to the method `createCircle()`. The `draw()` method is invoked on this instance of the local class `Circle` in the loop at (15). 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 (effectively) `final` local variables continue to be available to instances of local classes whenever these values are needed.

The output in **Example 9.12** also shows the actual names of the local classes. In fact, the local class names are reflected in the generated class file names. Because multiple local class declarations with the same name can be defined in the methods of the enclosing class, a numbering scheme (`$i`) is used to generate distinct class file names.

9.5 Static Local Types

It is possible to declare local interfaces, local enum types, and local record classes. However, these local nested types are implicitly `static`—as opposed to local classes that are never `static`. A local class declared in a static context is *not* the same as a static local type, and they are compared next.

Since these local nested types are implicitly `static`, this has implications for these types, as listed below. We illustrate these implications for a static local record class in **Example 9.13**, but they apply equally to local interfaces and local enum types as well.

- A local record class can be instantiated with the `new` operator without specifying an immediately enclosing instance. Interfaces and enum types cannot be instantiated with the `new` operator.
  At (10) in **Example 9.13**, an instance of the static local record class is created without passing a reference to the enclosing object.
- Static local types can only access `static` members in the enclosing context—this is the same as local classes declared in static context, whereas local classes declared in non-static context can access all members in the enclosing context.
  In **Example 9.13**, it is possible to access the `static` field in the enclosing class in the declaration of the local record class at (7) and (9), but it is not possible to access non-static fields as shown at (8).
- Static local types cannot access any local variables in the enclosing method—in contrast to local classes that can access (effectively) `final` local variables in the enclos-

ing method.

In **Example 9.13**, it is not possible to access any local variables, as shown at (5) and (6), in the enclosing method.

Example 9.13 *Defining Static Local Record Classes*

Click here to view code image

```
// File: LocalTypesClient.java
class LocalTypes {                        // Top-level Class
  private int nsf;                        // (1) Non-static field
  private static int sf;                  // (2) Static field

  void nonStaticMethod(final int fp) { // (3) Non-static Method. Final parameter.
    int lv = 20;                          // (4) Local variable

    record StaticLocalRecord(int val) { // Static local record
      // Cannot access local variables:
//    static int f1 = fp;  // (5) Cannot access final param from enclosing method.
//    static int f2 = lv;  // (6) Cannot access effectively final local variable
                           //     from enclosing method.

      // Can only access static fields in enclosing context:
      static int f3 = sf;         // (7) Access static field in enclosing context.

      void printFieldsFromEnclosingContext() {
//      System.out.println(nsf);  // (8) Cannot access non-static field
                                  //     in enclosing context.
        System.out.println(sf);   // (9) Access static field in enclosing context.
      }
    }

    // (10) Create local record. No enclosing instance passed to the constructor.
    StaticLocalRecord lrRef = new StaticLocalRecord(100);
    System.out.println("Value: " + lrRef.val());
  } // nonStaticMethod
}

public class LocalTypesClient {
  public static void main(String[] args) {
    new LocalTypes().nonStaticMethod(1000);    // (10)
  }
}
```

Output from the program:

```
Value: 100
```

9.6 Anonymous Classes

Declaring 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. Like local classes, anonymous classes are inner classes that can be defined in static and non-static context.

An anonymous class can be defined and instantiated in contexts where a reference value can be used—that is, as expressions that evaluate to a reference value denoting an object of the anonymous class. Anonymous classes are typically used for creating objects *on the fly* in contexts such as the value in a `return` statement, an argument in a method call, or in initialization of variables. The reference value of an anonymous class object can be assigned to any kind of variable (fields and local variables) whose type is a supertype of the anonymous class.

An anonymous class cannot be declared with an access modifier, nor can it be declared `static`, `final`, or `abstract`.

Typical uses of anonymous classes are to implement *event listeners* in GUI-based applications, threads for simple tasks (see examples in **Chapter 22**, **p. 1365**), and comparators for providing a total ordering for objects (see **Example 14.11**, **p. 772**).

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*:

Click here to view code image

```
new superclass_name<optional_type_arguments> (optional_constructor_arguments)
{
  member_declarations
}
```

Optional type arguments and constructor arguments can be specified, which are passed to the superclass constructor. Thus the superclass must provide a constructor corresponding to the arguments passed. No `extends` clause is used in the construct. 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 no-arg constructor.

Both static and non-static members can be declared in the class body. An anonymous class can override any instance methods accessible by their simple name from the superclass, but if it extends an `abstract` class, then it must provide implementation for all `abstract` methods from the superclass. The declaration is terminated by a semicolon (`;`), unless the reference value of the resulting object is immediately used to access a member of this object.

Example 9.14 *Defining Anonymous Classes*

Click here to view code image

```
// File: AnonClassClient.java
interface IDrawable {                              // (1)
  void draw();
}
//_______________________________________________________________________________
class Shape implements IDrawable {                 // (2)
  @Override
  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 at (2)
      @Override
      public void draw() { System.out.println("Drawing a new Shape."); }
    };
  }
  public static IDrawable createIDrawable() {      // (7) Static Method
    return new IDrawable() {                       // (8) Implements interface at (1)
      @Override
      public void draw() {
        System.out.println("Drawing a new IDrawable.");
      }
    };
  }
}
//_______________________________________________________________________________
public class AnonClassClient {
  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 (IDrawable aDrawable : drawables)       // (14)
      aDrawable.draw();

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

Output from the program:

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

Class declarations from **Example 9.12** are adapted to use anonymous classes in **Example 9.14**. The non-`static` method `createShape()` at (4) defines a non-`static` anonymous class at (5), which extends the superclass `Shape`. The anonymous class at (5) overrides the inherited method `draw()` from the superclass `Shape` at (2).

Click here to view code image

```
// ...
class Shape implements IDrawable {            // (2)
  @Override 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 at (2)
      @Override 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 the *interface name*:

Click here to view code image

```
new interface_name<optional_parameterized_types>() { member_declarations }
```

An anonymous class provides a *single* interface implementation. No arguments are passed, as an interface does not define a constructor. The anonymous class implicitly extends the `Object` class. Note that no `implements` clause is used in the construct. The class body must provide implementation for all `abstract` methods declared in the interface.

An anonymous class implementing an interface is shown below. Details can be found in Example 9.14. 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 that implements an interface is available through references of the interface type and the `Object` type—that is, its supertypes.

Click here to view code image

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

The following code is an example of a typical use of anonymous classes in building GUI applications. The anonymous class at (1) implements the j `ava.awt.event.-ActionListener` interface that has the method `actionPerformed()`. When the `addAc-tionListener()` method is called on the GUI button denoted by the reference `quit-`

Button, the anonymous class is instantiated and the reference value of the object is passed as a parameter to the method. The method addActionListener() of the GUI button registers the reference value, and when the user clicks the GUI button, it can invoke the actionPerformed() method on the ActionListener object.

Click here to view code image

```
quitButton.addActionListener(
    new ActionListener() {    // (1) Anonymous class implements an interface
      // Invoked when the user clicks the quit button.
      @Override public void actionPerformed(ActionEvent evt) {
        System.exit(0);       // (2) Terminates the program
      }
    }
);
```

Instantiating Anonymous Classes

The discussion on instantiating local classes (see **Example 9.12**) is also valid for instantiating anonymous classes. The class AnonClassClient in **Example 9.14** creates one instance at (11) of the non-static anonymous class defined at (5) by calling the non-static method createShape() on an instance of the class Painter, and two instances at (12) and (13) of the anonymous class that is defined in static context at (8) by calling the static method createDrawable() in the class Painter. 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 associated with it. An enclosing instance is not mandatory for creating objects of a static anonymous class.

The names of the anonymous classes at runtime are also shown in the program output in **Example 9.14**. A numbering scheme ($i) is used to designate the anonymous classes according to their declaration order inside the enclosing class. They are also the names used to designate their respective class files. Anonymous classes are not so anonymous after all.

Referencing Instances of an Anonymous Class

Each time an anonymous class declaration is executed it returns a reference value of a new instance of the anonymous class. Either an anonymous class extends an existing class or it implements an interface. Usually it makes sense to override (and implement) methods from its supertype. References of its supertype can refer to objects of the anonymous subclass. As we cannot declare references of an anonymous class, an instance of an anonymous class can only be manipulated by a supertype reference. Any subclass-specific members in an anonymous class cannot be accessed directly by external clients—the only functionality available is that provided by inherited methods and overridden methods which can be invoked by a supertype reference denoting an anonymous subclass instance.

Accessing Declarations in Enclosing Context

Member declarations and access rules for local classes (p. 512) also apply to anonymous classes. Example 9.15 is an adaptation of Example 9.9, Example 9.10, and Example 9.11. It illustrates what members can be declared in an anonymous class and what can be accessed in its enclosing context. The local classes from previous examples have been adapted to anonymous classes in Example 9.15.

The `TLCWithAnonClasses` class declares a non-`static` method at (3) in which a local variable (`baseRef`) is initialized with an instance of a non-`static` anonymous class at (9). The `baseRef` variable is used to invoke the `printValue()` method on this instance at (28).

The `TLCWithAnonClasses` class declares a `static` field `baseField` at (29) that is initialized with an instance of a `static` anonymous class. The field `baseField` is used to invoke the `printValue()` method on this instance at (36).

Accessing Local Declarations in the Enclosing Block

Being inner classes, there are many similarities between local classes and anonymous classes. An anonymous class can only access (effectively) `final` variables in its enclosing local context, shown both for static fields at (10), (11), and (12), and for instance fields at (13), (14), and (15). Local variables accessed in the anonymous class must be declared before use, and definitely assigned, as shown at (16) and (17), respectively. A field name in the anonymous class can shadow a local variable with the same name in the local context, as shown at (18). Note that the anonymous class declared at (29) does *not* have an enclosing local context, only an enclosing class, as it is defined in non-static context.

Accessing Members in the Enclosing Class

A member (either inherited or declared) in an anonymous class can hide a member with the same name in the enclosing object or class. The inherited field `nsf1` hides the field by the same name in the enclosing class, as shown at (19), (20), and (21).

Non-static anonymous classes can access any non-hidden members in the enclosing class by their simple names and the qualified `this`, as shown at (22), (23), and (24).

Static anonymous classes can only access non-hidden `static` members in the enclosing class by their simple names, as shown at (30), (31), and (32).

Example 9.15 *Accessing Declarations in Enclosing Context (Anonymous Classes)*

Click here to view code image

```
// File: AnonClient.java
abstract class Base {            // (1) Superclass
  protected int nsf1;
  abstract void printValue();
}
//_______________________________________________________________________________
class TLCWithAnonClasses {       // (2) Top level Class
  private int nsf1;              // Non-static field
  private int nsf2;              // Non-static field
  private static int sf = 5;     // Static field

  public void nonStaticMethod(final int fp) { // (3) Non-static Method
    // Local variables:
    int flv = 10;                // (4) Effectively final local variable
    final int hlv = 20;          // (5) Final local variable (constant variable)
    int nflv1 = 30;              // (6) Non-final local variable
    int nflv3;                   // (7) Non-final local variable declaration

    nflv1 = 40;                  // (8) Not effectively final local variable

    Base baseRef = new Base() { // (9) Non-static anonymous class
      // Static fields: Accessing local declarations in the enclosing block:
      static int sff1 = fp;    // (10) Final param from enclosing method
      static int sff2 = flv;   // (11) Effect. final variable from enclosing method
//    static int sf1 = nflv1; // (12) Not effect. final from enclosing method

      // Instance fields: Accessing local declarations in the enclosing block:
      int f1 = fp;        // (13) Final param from enclosing method
      int f2 = flv;       // (14) Effectively final variable from enclosing method
//    int f3 = nflv1;     // (15) Not effectively final from enclosing method

//    int f4 = nflv2;     // (16) nflv2 cannot be resolved: not decl-before-use
//    int f5 = nflv3;     // (17) Not definitely assigned: not initialized
      int hlv;            // (18) Shadows local variable at (5)

      // Accessing member declarations inherited from superclass:
      int f6 = nsf1;                // (19) Inherited from superclass
      int f7 = this.nsf1;           // (20) Inherited from superclass
      int f8 = super.nsf1;          // (21) Inherited from superclass

      // Accessing (hidden) member declarations in the enclosing class:
```

```
      int f9 = TLCWithAnonClasses.this.nsf1;          // (22) In enclosing object
      int f10 = nsf2;                  // (23) Instance field in enclosing object
      int f11 = sf;                    // (24) Static field from enclosing class

      { nsf1 = fp; }                   // (25) Non-static initializer block

      @Override void printValue() {                        // (26) Instance method
        System.out.println("Instance field nsf1: " + nsf1);// (27)
      }
    };

    int nflv2 = 70;
    nflv3 = 80;
    baseRef.printValue();              // (28) Invoke method on anonymous object
  }

  public static final Base baseField = new Base() { // (29) Static anonymous class
    // Accessing (hidden) member declarations in the enclosing class:
//  int f1 = TLCWithAnonClasses.this.nsf1; // (30) Not OK. No enclosing object
//  int f2 = nsf2;                         // (31) Not OK. No enclosing object
    { nsf1 = sf; }                         // (32) Non-static initializer block

    @Override void printValue() {                        // (33) Instance method
      System.out.println("Instance field nsf1: " + nsf1);// (34)
    }
  };
}
//______________________________________________________________________________
public class AnonClient {
  public static void main(String[] args) {
    new TLCWithAnonClasses().nonStaticMethod(100);                // (35)
    TLCWithAnonClasses.baseField.printValue();                    // (36)
  }
}
```

Output from the program:

Click here to view code image

```
Instance field nsf1: 100
Instance field nsf1: 5
```

Review Questions

9.5 Which statement is true about nested classes?

Select the one correct answer.

a. Non-static member classes must have either package or `public` access.

b. All nested classes cannot declare static member classes.

c. Methods in all nested classes cannot be declared `static`.

d. All nested classes can be declared `static`.

e. Static member classes can declare non-`static` methods.

9.6 Given the declaration:

Click here to view code image

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

Which of the following methods are valid?

Click here to view code image

```
//----(1)----
  IntHolder makeIntHolder(int i) {
    i = 10;
    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 the three correct answers.

a. The method at (1).

b. The method at (2).

c. The method at (3).

d. The method at (4).

e. The method at (5).

9.7 Which statement is true about nested classes?

Select the one correct answer.

a. No other static members, except `static final` fields declared as constant variables, can be declared within a non-static member class.

b. If a non-static member class is nested within a class named `Outer`, methods within the non-static member class must use the prefix `Outer.this` to access the members of the class `Outer`.

c. All fields in any nested class must be declared `final`.

d. Anonymous classes cannot have constructors.

e. If the reference `objRef` denotes an instance of any nested class within the class `Outer`, the expression `(objRef instanceof Outer)` will evaluate to `true`.

9.8 Which expression can be inserted independently at (1) so that compiling and running the program will print `LocalVar.str1`?

Click here to view code image

```
public class Access {
  final String str1 = "Access.str1";

  public static void main(final String args[]) {
    final String str1 = "LocalVar.str1";

    class Helper { String getStr1() { return str1; } }
    class Inner {
      String str1 = "Inner.str1";
      Inner() {
        System.out.println( /* (1) INSERT EXPRESSION HERE */ );
      }
    }
    Inner inner = new Inner();
  }
}
```

Select the one correct answer.

a. `str1`

b. `this.str1`

c. `Access.this.str1`

d. `new Helper().getStr1()`

e. `this.new Helper().getStr1()`

f. `Access.new Helper().getStr1()`

g. `new Access.Helper().getStr1()`

h. `new Access().new Helper().getStr1()`

9.9 Which statement is true about the following program?

Click here to view code image

```
public class Test {
  private char x = '=';
  public static void main(String[] args) {
    char x = '<';
    Test t = new Test() {
```

```
      private char x = '>';
      public String toString() {
        return this.x + super.toString() + x;
      }
    };
    System.out.println(t);
  }
  public String toString() {
    return x + "42";
  }
}
```

Select the one correct answer.

a. The program will print `<=42>`.

b. The program will print `<=42<`.

c. The program will print `>=42>`.

d. The program will print `>=42<`.

e. The program will compile, but it will throw an exception at runtime.

f. The program will fail to compile.

9.10 Which statement is true about the following program?

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    int x = 42;
    String s = new String() {
      int x = 24;
      public int hashCode() {
        return this.x;
      }
    };
    System.out.println(s.hashCode());
  }
}
```

Select the one correct answer.

a. The program will print `42`.

b. The program will print `24`.

c. The program will compile, but it will throw an exception at runtime.

d. The program will fail to compile.