3.1 Class Declarations

A class declaration introduces a new reference type. For the purpose of this book, we will use the following simplified syntax of a class declaration:

```
class_modifiers class class_name
                      extends_clause
                      implements_clause // Class header
{ // Class body
    field_declarations
    method_declarations
    constructor_declarations
}
```

In the class header, the name of the class is preceded by the keyword `class`. In addition, the class header can specify the following information:

- An *accessibility modifier* (§4.5, p. 118)
- Additional *class modifiers* (§4.6, p. 120)
- Any class it *extends* (§7.1, p. 264)
- Any interfaces it *implements* (§7.6, p. 290)

The class body, enclosed in braces (`{}`), can contain *member declarations*. In this book, we discuss the following two kinds of member declarations:

- *Field declarations* (§2.3, p. 40)
- *Method declarations* (§3.2, p. 49)

Members declared `static` belong to the class and are called *static members*. Non-static members belong to the objects of the class and are called *instance members*. In addition, the following declarations can be included in a class body:

- *Constructor declarations* (§3.3, p. 53)

The declarations can appear in any order in the class body. The only mandatory parts of the class declaration syntax are the keyword `class`, the class name, and the class body braces (`{}`), as exemplified by the following class declaration:

```
class X { }
```

To understand which code can be legally declared in a class, we distinguish between *static context* and *non-static context*. A static context is defined by static methods, static field initializers, and static initializer blocks. A non-static context is defined by instance methods, non-static field initializers, instance initializer blocks, and constructors. By *static code*, we mean expressions and statements in a static context; by *non-static code*, we mean expressions and statements in a non-static context. One crucial difference between the two contexts is that static code can refer only to other static members.

3.2 Method Declarations

For the purpose of this book, we will use the following simplified syntax of a method declaration:

method_modifiers return_type method_name
(*formal_parameter_list*) *throws_clause* `// Method header`

```
{ // Method body
    local_variable_declarations
    statements
}
```

In addition to the name of the method, the method header can specify the following information:

- Scope or *accessibility modifier* (§4.7, p. 123)
- Additional *method modifiers* (§4.8, p. 131)
- The *type* of the *return value*, or `void` if the method does not return any value (§6.4, p. 224)
- A *formal parameter list*
- Any *exceptions* thrown by the method, which are specified in a `throws` clause (§6.9, p. 251)

The *formal parameter list* is a comma-separated list of parameters for passing information to the method when the method is invoked by a *method call* (§3.5, p. 72). An empty parameter list must be specified by `()`. Each parameter is a simple variable declaration consisting of its type and name:

optional_parameter_modifier type parameter_name

The parameter names are local to the method (§4.4, p. 117). The *optional parameter modifier* `final` is discussed in §3.5, p. 80. It is recommended to use the `@param` tag in a Javadoc comment to document the formal parameters of a method.

The *signature* of a method comprises the name of the method and the types of the formal parameters only.

The method body is a *block* containing the *local variable declarations* (§2.3, p. 40) and the *statements* of the method.

The mandatory parts of a method declaration are the return type, the method name, and the method body braces (`{}`), as exemplified by the following method declaration:

```
void noAction() {}
```

Like member variables, member methods can be characterized as one of two types:

- *Instance methods*, which are discussed later in this section
- *Static methods*, which are discussed in §4.8, p. 132

Statements

Statements in Java can be grouped into various categories. Variable declarations with explicit initialization of the variables are called *declaration statements* (§2.3, p. 40, and §3.4, p. 60). Other basic forms of statements are *control flow statements* (§6.1, p. 200) and *expression statements*.

An *expression statement* is an expression terminated by a semicolon. Any value returned by the expression is discarded. Only certain types of expressions have meaning as statements:

- Assignments (§5.6, p. 158)
- Increment and decrement operators (§5.9, p. 176)
- Method calls (§3.5, p. 72)
- Object creation expressions with the `new` operator (§5.17, p. 195)

A solitary semicolon denotes the *empty statement*, which does nothing.

A block, `{}`, is a *compound* statement that can be used to group zero or more local declarations and statements (§4.4, p. 117). Blocks can be nested, since a block is a statement that can contain other statements. A block can be used in any context where a simple statement is permitted. The compound statement that is embodied in a block begins at the left brace, `{`, and ends with a matching right brace, `}`. Such a block must not be confused with an array initializer in declaration statements (§3.4, p. 60).

Labeled statements are discussed in §6.4 on page 220.

Instance Methods and the Object Reference `this`

Instance methods belong to every object of the class and can be invoked only on objects. All members defined in the class, both static and non-static, are accessible in the context of an instance method. The reason is that all instance methods are passed an implicit reference to the *current object*—that is, the object on which the method is being invoked. The current object can be referenced in the body of the instance method by the keyword `this`. In the body of the method, the `this` reference can be used like any other object reference to access members of the object. In fact, the keyword `this` can be used in any non-static context. The `this` reference can be used as a normal reference to reference the current object, but the reference cannot be modified—it is a `final` reference (§4.8, p. 133).

The `this` reference to the current object is useful in situations where a local variable hides, or *shadows*, a field with the same name. In Example 3.1, the two parameters `noOfWatts` and `indicator` in the constructor of the `Light` class have the same names as the fields in the class. The example also declares a local variable `location`, which has the same name as one of the fields. The reference `this` can be used to distinguish the fields from the local variables. At (1), the `this` reference is used to identify the field `noOfWatts`, which is assigned the value of the parameter `noOfWatts`. Without the `this` reference at (2), the value of the parameter `indicator` is assigned back to

this parameter, and not to the field by the same name, resulting in a logical error. Similarly at (3), without the `this` reference, it is the local variable `location` that is assigned the value of the parameter `site`, and not the field with the same name.

Example 3.1 *Using the* `this` *Reference*

```
public class Light {
  // Fields:
  int     noOfWatts;      // Wattage
  boolean indicator;      // On or off
  String  location;       // Placement

  // Constructor
  public Light(int noOfWatts, boolean indicator, String site) {
    String location;

    this.noOfWatts = noOfWatts;   // (1) Assignment to field
    indicator = indicator;        // (2) Assignment to parameter
    location = site;              // (3) Assignment to local variable
    this.superfluous();           // (4)
    superfluous();                // equivalent to call at (4)
  }

  public void superfluous() {
    System.out.printf("Current object: %s%n", this); // (5)
  }

  public static void main(String[] args) {
    Light light = new Light(100, true, "loft");
    System.out.println("No. of watts: " + light.noOfWatts);
    System.out.println("Indicator:    " + light.indicator);
    System.out.println("Location:     " + light.location);
  }
}
```

Probable output from the program:

```
Current object: Light@1bc4459
Current object: Light@1bc4459
No. of watts: 100
Indicator:    false
Location:     null
```

If a member is not shadowed by a local declaration, the simple name `member` is considered a short-hand notation for `this.member`. In particular, the `this` reference can be used explicitly to invoke other methods in the class. This usage is illustrated at (4) in Example 3.1, where the method `superfluous()` is called.

If, for some reason, a method needs to pass the current object to another method, it can do so using the `this` reference. This approach is illustrated at (5) in Example 3.1, where the current object is passed to the `printf()` method. The `printf()` method

prints the string representation of the current object (which comprises the name of the class of the current object and the hexadecimal representation of the current object's hash code). (The *hash code* of an object is an `int` value that can be used to store and retrieve the object from special data structures called *hash tables*.)

Note that the `this` reference cannot be used in a static context, as static code is not executed in the context of any object.

Method Overloading

Each method has a *signature*, which comprises the name of the method plus the types and order of the parameters in the formal parameter list. Several method implementations may have the same name, as long as the method signatures differ. This practice is called *method overloading*. Because overloaded methods have the same name, their parameter lists must be different.

Rather than inventing new method names, method overloading can be used when the same logical operation requires multiple implementations. The Java SE platform API makes heavy use of method overloading. For example, the class `java.lang.Math` contains an overloaded method `min()`, which returns the minimum of two numeric values.

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

In the following examples, five implementations of the method `methodA` are shown:

```
void methodA(int a, double b) { /* ... */ }      // (1)
int  methodA(int a)           { return a; }      // (2)
int  methodA()                { return 1; }      // (3)
long methodA(double a, int b) { return b; }      // (4)
long methodA(int x, double y) { return x; }      // (5) Not OK.
```

The corresponding signatures of the five methods are as follows:

```
methodA(int, double)          1'
methodA(int)                  2': Number of parameters
methodA()                     3': Number of parameters
methodA(double, int)          4': Order of parameters
methodA(int, double)          5': Same as 1'
```

The first four implementations of the method named `methodA` are overloaded correctly, each time with a different parameter list and, therefore, different signatures. The declaration at (5) has the same signature `methodA(int, double)` as the declaration at (1) and, therefore, is not a valid overloading of this method.

```
void bake(Cake k)  { /* ... */ }                 // (1)
void bake(Pizza p) { /* ... */ }                 // (2)

int     halfIt(int a) { return a/2; }            // (3)
double  halfIt(int a) { return a/2.0; }          // (4) Not OK. Same signature.
```

The method named `bake` is correctly overloaded at (1) and (2), with two different parameter lists. In the implementation, changing just the return type (as shown at (3) and (4) in the preceding example), is not enough to overload a method, and will be flagged as a compile-time error. The parameter list in the declarations must be different.

Only methods declared in the same class and those that are inherited by the class can be overloaded. Overloaded methods should be considered to be individual methods that just happen to have the same name. Methods with the same name are allowed, since methods are identified by their signature. At compile time, the right implementation of an overloaded method is chosen, based on the signature of the method call. Details of method overloading resolution can be found in §7.10 on page 316. Method overloading should not be confused with *method overriding* (§7.2, p. 268).

3.3 Constructors

The main purpose of constructors is to set the initial state of an object, when the object is created by using the `new` operator.

For the purpose of this book, we will use the following simplified syntax of a constructor:

```
accessibility_modifier class_name (formal_parameter_list)
                                   throws_clause // Constructor header
{ // Constructor body
    local_variable_declarations
    statements
}
```

Constructor declarations are very much like method declarations. However, the following restrictions on constructors should be noted:

- Modifiers other than an accessibility modifier are not permitted in the constructor header. For accessibility modifiers for constructors, see §4.7, p. 123.
- Constructors cannot return a value and, therefore, do not specify a return type, not even `void`, in the constructor header. But their declaration can use the `return` statement that does not return a value in the constructor body (§6.4, p. 224).
- The constructor name must be the same as the class name.

Class names and method names exist in different *namespaces*. Thus, there are no name conflicts in Example 3.2, where a method declared at (2) has the same name as the constructor declared at (1). A method must always specify a return type, whereas a constructor does not. However, using such naming schemes is strongly discouraged.

A constructor that has no parameters, like the one at (1) in Example 3.2, is called a *no-argument constructor*.

Example 3.2 *Namespaces*

```
public class Name {

  Name() {                      // (1) No-argument constructor
    System.out.println("Constructor");
  }

  void Name() {                 // (2) Instance method
    System.out.println("Method");
  }

  public static void main(String[] args) {
    new Name().Name();          // (3) Constructor call followed by method call
  }
}
```

Output from the program:

```
Constructor
Method
```

The Default Constructor

If a class does not specify *any* constructors, then a *default constructor* is generated for the class by the compiler. The default constructor is equivalent to the following implementation:

```
class_name() { super(); }   // No parameters. Calls superclass constructor.
```

A default constructor is a no-argument constructor. The only action taken by the default constructor is to call the superclass constructor. This ensures that the inherited state of the object is initialized properly (§7.5, p. 282). In addition, all instance variables in the object are set to the default value of their type, barring those that are initialized by an initialization expression in their declaration.

In the following code, the class `Light` does not specify any constructors:

```
class Light {
  // Fields:
  int     noOfWatts;       // Wattage
  boolean indicator;       // On or off
  String  location;        // Placement

  // No constructors
  //...
}

class Greenhouse {
  // ...
  Light oneLight = new Light();     // (1) Call to default constructor
}
```

In this code, the following default constructor is called when a `Light` object is created by the object creation expression at (1):

```
Light() { super(); }
```

Creating an object using the `new` operator with the default constructor, as at (1), will initialize the fields of the object to their default values (that is, the fields `noOfWatts`, `indicator`, and `location` in a `Light` object will be initialized to `0`, `false`, and `null`, respectively).

A class can choose to provide its own constructors, rather than relying on the default constructor. In the following example, the class `Light` provides a no-argument constructor at (1).

```
class Light {
  // ...
  Light() {                          // (1) No-argument constructor
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }
  //...
}

class Greenhouse {
  // ...
  Light extraLight = new Light();   // (2) Call of explicit default constructor
}
```

The no-argument constructor ensures that any object created with the object creation expression `new Light()`, as at (2), will have its fields `noOfWatts`, `indicator`, and `location` initialized to `50`, `true`, and `"X"`, respectively.

If a class defines *any* constructor, it can no longer rely on the default constructor to set the state of its objects. If such a class requires a no-argument constructor, it must provide its own implementation, as in the preceding example. In the next example the class `Light` does not provide a no-argument constructor, but rather includes a non-zero argument constructor at (1). It is called at (2) when an object of the class `Light` is created with the `new` operator. Any attempt to call the default constructor will be flagged as a compile-time error, as shown at (3).

```
class Light {
  // ...
  // Only non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {          // (1)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}

class Greenhouse {
  // ...
  Light moreLight  = new Light(100, true, "Greenhouse");// (2) OK
  Light firstLight = new Light();                       // (3) Compile-time error
}
```