85

A class defines a new reference type, such as the `Point` type defined in Chapter 2. An *object* is an *instance* of a class. The `Point` class defines a type that is the set of all possible two-dimensional points. A `Point` object is a value of that type: it represents a single two-dimensional point.

Objects are usually created by *instantiating* a class with the `new` keyword and a constructor invocation, as shown here:

```
Point p = new Point(1.0, 2.0);
```

A class definition consists of a *signature* and a *body*. The class signature defines the name of the class and may also specify other important information. The body of a class is a set of *members* enclosed in curly braces. The members of a class may include fields and methods, constructors and initializers, and nested types.

Members can be *static* or nonstatic. A static member belongs to the class itself while a nonstatic member is associated with the instances of a class (see "Fields and Methods" later in this chapter).

The signature of a class may declare that the class *extends* another class. The extended class is known as the *superclass* and the extension is known as the *subclass*. A subclass *inherits* the members of its superclass and may declare new members or *override* inherited methods with new implementations.

The signature of a class may also declare that the class *implements* one or more interfaces. An *interface* is a reference type that defines method signatures but does not include method bodies to implement the methods. A class that implements an interface is required to provide bodies for the interface's methods. Instances of such a class are also instances of the interface type that it implements.

The members of a class may have *access modifiers* `public`, `protected`, or `private`, which specify their visibility and accessibility to clients and to subclasses. This allows classes to hide members that are not part of their public API. When applied to fields, this ability to hide members enables an object-oriented design technique known as *data encapsulation*.

Excerpts from Flanagan, D. (2005). *Java in a Nutshell*. Cambridge, MA. O'Reilly.

86

At its simplest level, a class definition consists of the keyword `class` followed by the name of the class and a set of class members within curly braces. The `class` keyword may be preceded by modifier keywords and annotations (see Chapter 4). If the class extends another class, the class name is followed by the `extends` keyword and the name of the class being extended. If the class implements one or more interfaces then the class name or the `extends` clause is followed by the `implements` keyword and a comma-separated list of interface names. For example:

```
public class Integer extends Number implements Serializable, Comparable {
    // class members go here
}
```

Generic class declarations include additional syntax that is covered in Chapter 4.

Class declarations may include zero or more of the following modifiers:

`public`
: A `public` class is visible to classes defined outside of its package. See "Data Hiding and Encapsulation" later in this chapter.

`abstract`
: An `abstract` class is one whose implementation is incomplete and cannot be instantiated. Any class with one or more `abstract` methods must be declared `abstract`.

`final`
: The `final` modifier specifies that the class may not be extended. Declaring a class `final` may enable the Java VM to optimize its methods.

`strictfp`
: If a class is declared `strictfp`, all its methods behave as if they were declared `strictfp`. This rarely used modifier is discussed in "Methods" in Chapter 2.

A class cannot be both `abstract` and `final`. By convention, if a class has more than one modifier, they appear in the order shown.

87

A class can be viewed as a collection of data and code to operate on that data. The data is stored in fields, and the code is organized into methods. This section covers fields and methods, the two most important kinds of class members. Fields and methods come in two distinct types: class members (also known as static members) are associated with the class itself, while instance members are associated with individual instances of the class (i.e., with objects). This gives us four kinds of members:

- Class fields
- Class methods
- Instance fields
- Instance methods

Field declaration syntax is much like the syntax for declaring local variables (see Chapter 2) except that field definitions may also include modifiers. The simplest field declaration consists of the field type followed by the field name. The type may be preceded by zero or more modifier keywords or annotations (see Chapter 4), and the name may be followed by an equals sign and initializer expression that provides the initial value of the field. If two or more fields share the same type and modifiers, the type may be followed by a comma-separated list of field names and initializers. Here are some valid field declarations:

```
int x = 1;
private String name;
public static final DAYS_PER_WEEK = 7;
String[] daynames = new String[DAYS_PER_WEEK];
private int a = 17, b = 37, c = 53;
```

Field modifiers are comprised of zero or more of the following keywords:

`public`, `protected`, `private`
: These access modifiers specify whether and where a field can be used outside of the class that defines it. These important modifiers are covered in "Data Hiding and Encapsulation" later in this chapter. No more than one of these access modifiers may appear in any field declaration.

`static`
: If present, this modifier specifies that the field is associated with the defining class itself rather than with each instance of the class.

`final`
: This modifier specifies that once the field has been initialized, its value may never be changed. Fields that are both `static` and `final` are compile-time constants that the compiler can inline. `final` fields can also be used to create classes whose instances are immutable.

88

Any field declared without the `static` modifier is an *instance field*:

```
public double r;    // The radius of the circle
```

Instance fields are associated with instances of the class, rather than with the class itself. Thus, every `Circle` object we create has its own copy of the `double` field `r`. In our example, `r` represents the radius of a circle. Thus, each `Circle` object can have a radius independent of all other `Circle` objects.

Inside a class definition, instance fields are referred to by name alone. You can see an example of this if you look at the method body of the `circumference()` instance method. In code outside the class, the name of an instance method must be prefixed with a reference to the object that contains it. For example, if the variable `c` holds a reference to a `Circle` object, we use the expression `c.r` to refer to the radius of that circle:

```
Circle c = new Circle(); // Create a Circle object; store a reference in c
c.r = 2.0;               // Assign a value to its instance field r
Circle d = new Circle(); // Create a different Circle object
d.r = c.r * 2;           // Make this one twice as big
```

Instance fields are key to object-oriented programming. Instance fields hold the state of an object; the values of those fields make one object distinct from another.

A *class field* is associated with the class in which it is defined rather than with an instance of the class. The following line declares a class field:

```
public static final double PI = 3.14159;
```

This line declares a field of type `double` named `PI` and assigns it a value of 3.14159. As you can see, a field declaration looks quite a bit like a local variable declaration. The difference, of course, is that variables are defined within methods while fields are members of classes.

The key point to understand about a static field is that there is only a single copy of it. This field is associated with the class itself, not with instances of the class. If you look at the various methods of the `Circle` class, you'll see that they use this field. From inside the `Circle` class, the field can be referred to simply as `PI`. Outside the class, however, both class and field names are required to uniquely specify the field. Methods that are not part of `Circle` access this field as `Circle.PI`.

Excerpts from Flanagan, D. (2005). *Java in a Nutshell*. Cambridge, MA. O'Reilly.