Methods

A *method* is a named sequence of Java statements that can be invoked by other Java code. When a method is invoked, it is passed zero or more values known as *arguments*. The method performs some computations and, optionally, returns a value. As described earlier in "Expressions and Operators" on page 30, a method invocation is an expression that is evaluated by the Java interpreter. Because method invocations can have side effects, however, they can also be used as expression statements. This section does not discuss method invocation, but instead describes how to define methods.

Defining Methods

You already know how to define the body of a method; it is simply an arbitrary sequence of statements enclosed within curly braces. What is more interesting about a method is its *signature*.[3] The signature specifies the following:

- The name of the method
- The number, order, type, and name of the parameters used by the method
- The type of the value returned by the method
- The checked exceptions that the method can throw (the signature may also list unchecked exceptions, but these are not required)
- Various method modifiers that provide additional information about the method

A method signature defines everything you need to know about a method before calling it. It is the method *specification* and defines the API for the method. In order to use the Java platform's online API reference, you need to know how to read a method signature. And, in order to write Java programs, you need to know how to define your own methods, each of which begins with a method signature.

A method signature looks like this:

```
modifiers type name ( paramlist ) [ throws exceptions ]
```

The signature (the method specification) is followed by the method body (the method implementation), which is simply a sequence of Java statements enclosed in curly braces. If the method is *abstract* (see Chapter 3), the implementation is omitted, and the method body is replaced with a single semicolon.

The signature of a method may also include type variable declarations—such methods are known as *generic methods*. Generic methods and type variables are discussed in Chapter 4.

3 In the Java Language Specification, the term "signature" has a technical meaning that is slightly different than that used here. This book uses a less formal definition of method signature.

Here are some example method definitions, which begin with the signature and are followed by the method body:

```
// This method is passed an array of strings and has no return value.
// All Java programs have an entry point with this name and signature.
public static void main(String[] args) {
     if (args.length > 0) System.out.println("Hello " + args[0]);
     else System.out.println("Hello world");
}

// This method is passed two double arguments and returns a double.
static double distanceFromOrigin(double x, double y) {
     return Math.sqrt(x*x + y*y);
}

// This method is abstract which means it has no body.
// Note that it may throw exceptions when invoked.
protected abstract String readText(File f, String encoding)
    throws FileNotFoundException, UnsupportedEncodingException;
```

`modifiers` is zero or more special modifier keywords, separated from each other by spaces. A method might be declared with the `public` and `static` modifiers, for example. The allowed modifiers and their meanings are described in the next section.

The *`type`* in a method signature specifies the return type of the method. If the method does not return a value, *`type`* must be `void`. If a method is declared with a non-`void` return type, it must include a `return` statement that returns a value of (or convertible to) the declared type.

A *constructor* is a block of code, similar to a method, that is used to initialize newly created objects. As we'll see in Chapter 3, constructors are defined in a very similar way to methods, except that their signatures do not include this *`type`* specification.

The *`name`* of a method follows the specification of its modifiers and type. Method names, like variable names, are Java identifiers and, like all Java identifiers, may contain letters in any language represented by the Unicode character set. It is legal, and often quite useful, to define more than one method with the same name, as long as each version of the method has a different parameter list. Defining multiple methods with the same name is called *method overloading*.

Unlike some other languages, Java does not have anonymous methods. Instead, Java 8 introduces lambda expressions, which are similar to anonymous methods, but which the Java runtime automatically converts to a suitable named method—see "Lambda Expressions" on page 76 for more details.

For example, the `System.out.println()` method we've seen already is an overloaded method. One method by this name prints a string and other methods by the same name print the values of the various primitive types. The Java compiler

decides which method to call based on the type of the argument passed to the method.

When you are defining a method, the name of the method is always followed by the method's parameter list, which must be enclosed in parentheses. The parameter list defines zero or more arguments that are passed to the method. The parameter specifications, if there are any, each consist of a type and a name and are separated from each other by commas (if there are multiple parameters). When a method is invoked, the argument values it is passed must match the number, type, and order of the parameters specified in this method signature line. The values passed need not have exactly the same type as specified in the signature, but they must be convertible to those types without casting.

When a Java method expects no arguments, its parameter list is simply `()`, not `(void)`. Java does not regard `void` as a type—C and C++ programmers in particular should pay heed.

Java allows the programmer to define and invoke methods that accept a variable number of arguments, using a syntax known colloquially as *varargs*. Varargs are covered in detail later in this chapter.

The final part of a method signature is the `throws` clause, which is used to list the *checked exceptions* that a method can throw. Checked exceptions are a category of exception classes that must be listed in the `throws` clauses of methods that can throw them. If a method uses the `throw` statement to throw a checked exception, or if it calls some other method that throws a checked exception and does not catch or handle that exception, the method must declare that it can throw that exception. If a method can throw one or more checked exceptions, it specifies this by placing the `throws` keyword after the argument list and following it by the name of the exception class or classes it can throw. If a method does not throw any exceptions, it does not use the `throws` keyword. If a method throws more than one type of exception, separate the names of the exception classes from each other with commas. More on this in a bit.

Method Modifiers

The modifiers of a method consist of zero or more modifier keywords such as `public`, `static`, or `abstract`. Here is a list of allowed modifiers and their meanings:

`abstract`
: An `abstract` method is a specification without an implementation. The curly braces and Java statements that would normally comprise the body of the method are replaced with a single semicolon. A class that includes an `abstract` method must itself be declared `abstract`. Such a class is incomplete and cannot be instantiated (see Chapter 3).

`final`
: A `final` method may not be overridden or hidden by a subclass, which makes it amenable to compiler optimizations that are not possible for regular methods. All `private` methods are implicitly `final`, as are all methods of any class that is declared `final`.

`native`
: The `native` modifier specifies that the method implementation is written in some "native" language such as C and is provided externally to the Java program. Like `abstract` methods, `native` methods have no body: the curly braces are replaced with a semicolon.

Implementing native Methods

When Java was first released, `native` methods were sometimes used for efficiency reasons. That is almost never necessary today. Instead, `native` methods are used to interface Java code to existing libraries written in C or C++. `native` methods are implicitly platform-dependent, and the procedure for linking the implementation with the Java class that declares the method is dependent on the implementation of the Java virtual machine. `native` methods are not covered in this book.

`public, protected, private`
: These access modifiers specify whether and where a method can be used outside of the class that defines it. These very important modifiers are explained in Chapter 3.

`static`
: A method declared `static` is a *class method* associated with the class itself rather than with an instance of the class (we cover this in more detail in Chapter 3).

`strictfp`
: The `fp` in this awkwardly named, rarely used modifier stands for "floating point." Java normally takes advantage of any extended precision available to the runtime platform's floating-point hardware. The use of this keyword forces Java to strictly obey the standard while running the `strictfp` method and only perform floating-point arithmetic using 32- or 64-bit floating-point formats, even if this makes the results less accurate.

`synchronized`
: The `synchronized` modifier makes a method threadsafe. Before a thread can invoke a `synchronized` method, it must obtain a lock on the method's class (for `static` methods) or on the relevant instance of the class (for non-`static` methods). This prevents two threads from executing the method at the same time.

The `synchronized` modifier is an implementation detail (because methods can make themselves threadsafe in other ways) and is not formally part of the method specification or API. Good documentation specifies explicitly whether a method is threadsafe; you should not rely on the presence or absence of the `synchronized` keyword when working with multithreaded programs.

Annotations are an interesting special case (see Chapter 4 for more on annotations)—they can be thought of as a halfway house between a method modifier and additional supplementary type information.

Checked and Unchecked Exceptions

The Java exception-handling scheme distinguishes between two types of exceptions, known as *checked* and *unchecked* exceptions.

The distinction between checked and unchecked exceptions has to do with the circumstances under which the exceptions could be thrown. Checked exceptions arise in specific, well-defined circumstances, and very often are conditions from which the application may be able to partially or fully recover.

For example, consider some code that might find its configuration file in one of several possible directories. If we attempt to open the file from a directory it isn't present in, then a `FileNotFoundException` will be thrown. In our example, we want to catch this exception and move on to try the next possible location for the file. In other words, although the file not being present is an exceptional condition, it is one from which we can recover, and it is an understood and anticipated failure.

On the other hand, in the Java environment there are a set of failures that cannot easily be predicted or anticipated, due to such things as runtime conditions or abuse of library code. There is no good way to predict an `OutOfMemoryError`, for example, and any method that uses objects or arrays can throw a `NullPointerException` if it is passed an invalid `null` argument.

These are the unchecked exceptions—and practically any method can throw an unchecked exception at essentially any time. They are the Java environment's version of Murphy's law: "Anything that can go wrong, will go wrong." Recovery from an unchecked exception is usually very difficult, if not impossible—simply due to their sheer unpredictability.

To figure out whether an exception is checked or unchecked, remember that exceptions are `Throwable` objects and that exceptions fall into two main categories, specified by the `Error` and `Exception` subclasses. Any exception object that is an `Error` is unchecked. There is also a subclass of `Exception` called `RuntimeException`—and any subclass of `RuntimeException` is also an unchecked exception. All other exceptions are checked exceptions.