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 33, 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.

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 ]
```

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);
}
```


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 is 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*.

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.

Excerpts from Flanagan, D. (2019). *Java in a Nutshell*. Sebastopol, CA. O'Reilly.

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.

`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).

`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.

Now that we have introduced operators, expressions, statements, and methods, we can finally talk about classes. A *class* is a named collection of fields that hold data values and methods that operate on those values. Classes are just one of five reference types supported by Java, but they are the most important type. Classes are thoroughly documented in a chapter of their own (Chapter 3). We introduce them here, however, because they are the next higher level of syntax after methods, and because the rest of this chapter requires a basic familiarity with the concept of a class and the basic syntax for defining a class, instantiating it, and using the resulting *object*.

The most important thing about classes is that they define new data types. For example, you might define a class named `Point` to represent a data point in the two-dimensional Cartesian coordinate system. This class would define fields (each of type `double`) to hold the *x* and *y* coordinates of a point and methods to manipulate and operate on the point. The `Point` class is a new data type.

When discussing data types, it is important to distinguish between the data type itself and the values the data type represents. `char` is a data type: it represents Unicode characters. But a `char` value represents a single specific character. A class is a data type; a class value is called an *object*. We use the name class because each class defines a type (or kind, or species, or class) of objects. The `Point` class is a data type that represents *x,y* points, while a `Point` object represents a single specific *x,y* point. As you might imagine, classes and their objects are closely linked. In the sections that follow, we will discuss both.

Class	Object
• Category	• Single item
• Type	• Value
• Defines fields	• Has data in fields

Excerpts from Flanagan, D. (2019). *Java in a Nutshell*. Sebastopol, CA. O'Reilly.