1

Getting Started

COPYRIGHTED MATERIAL

This journey into the world of algorithms begins with some preparation and background information. You'll need to know a few things before learning the many algorithms and data structures in the rest of the book. Although you're keen to get going, reading this chapter will make the rest of the book more useful for you, as it includes concepts that are prerequisites for understanding the explanations of all the code and the analyses of the algorithms themselves.

This chapter discusses:

- What an algorithm is
- The role of algorithms in software and everyday life
- What is meant by the complexity of an algorithm
- Several broad classes of algorithm complexity that enable you to distinguish quickly between different solutions to the same problem
- "Big-O" notation
- What unit testing is and why it is important
- How to write unit tests with JUnit

Defining Algorithms

Perhaps you already know that algorithms are an important part of computing, but what exactly are they? What are they good for? And should you even care?

Well, as it turns out, algorithms aren't just limited to computing; you use algorithms every day of your life. In simple terms, an *algorithm* is a set of well-defined steps required to accomplish some task. If you've ever baked a cake, or followed a recipe of any kind, then you've used an algorithm.

Algorithms also usually involve taking a system from one state to another, possibly transitioning through a series of intermediate states along the way. Another example of this from everyday life

is simple integer multiplication. Although most of us memorized multiplication tables back in grade school, the actual process of multiplication can also be thought of as a series of additions. The expression 5×2, for example, is really a shorthand way of saying $2 + 2 + 2 + 2 + 2$ (or $5 + 5$ for that matter). Therefore, given any two integers *A* and *B*, we can say that multiplying *A times B* involves adding *B* to itself, *A* times. This can be expressed as a sequence of steps:

1. Initialize a third integer, *C*, to zero.
2. If *A* is zero, we're done and *C* contains the result. Otherwise, proceed to step 3.
3. Add the value of *B* to *C*.
4. Decrement *A*.
5. Go to step 2.

Notice that unlike a recipe for baking a cake, the multiplication-using-addition algorithm loops back on itself at step 5. Most algorithms involve some kind of looping to repeatedly apply a calculation or other computation. *Iteration* and *recursion* — the two main types of looping — are covered in detail in the next chapter.

Quite often, algorithms are described in pseudo-code, a kind of *made-up* programming language that is easy to understand, even for nonprogrammers. The following code shows a function, `Multiply`, that takes two integers — *A* and *B* — and returns $A \times B$ using only addition. This is pseudo-code representing the act of multiplying two integers using addition:

```
Function Multiply(Integer A, Integer B)
    Integer C = 0

    While A is greater than 0
        C = C + B
        A = A - 1
    End

    Return C
End
```

Of course, multiplication is a very simple example of an algorithm. Most applications you are likely to encounter will involve algorithms that are far more complex than this. The problem with complex algorithms is that they are inherently more difficult to understand and therefore are more likely to contain bugs. (In fact, a large part of computer science involves proving that certain algorithms work correctly.)

Coming up with algorithms isn't always easy. In addition, more than one algorithm may solve a given problem. Some solutions will be simple, others will be complex, and some will be more efficient than others. The simplest solution isn't always the most obvious either. While rigorous, scientific analysis is always a good starting point, you can often find yourself stuck in *analysis paralysis*. Sometimes a bit of good old-fashioned creativity is needed. Try different approaches, and investigate hunches. Determine why your current attempts at a solution work for some cases and not for others. There is a reason why one of the seminal works on so-called computer *science* and software *engineering* is called *The* Art *of Computer Programming* (authored by Donald E. Knuth). Most of the algorithms in this book are *deterministic* — the result of any algorithm can be determined exactly based on the inputs. Sometimes, however, a problem is so difficult that finding a precise solution can be too costly in terms of time or resources. In this case, a *heuristic* may be a more practical approach. Rather than try to find a perfect solution, a

heuristic uses certain well-known characteristics of a problem to produce an approximate solution. Heuristics are often used to sift through data, removing or ignoring values that are irrelevant so that the more computationally expensive parts of an algorithm can operate on a smaller set of data.

A rather lighthearted example of a heuristic involves crossing the street in different countries of the world. In North America and most of Europe, vehicles drive on the right-hand side of the road. If you've lived in the United States your whole life, then you're no doubt used to looking left and then right before crossing the street. If you were to travel to Australia, however, and looked left, saw that the road was clear, and moved onto the street, you would be in for quite a shock because in Australia, as in the United Kingdom, Japan, and many other countries, vehicles drive on the left-hand side of the road.

One simple way to tell which way the cars are traveling irrespective of which country you're in is to look at the direction of the parked cars. If they are all lined up pointing left-to-right, then chances are good you will need to look left and then right before crossing the road. Conversely, if the parked cars are lined up pointing right-to-left, then you will need to look right and then left before crossing the street. This simple heuristic works *most* of the time. Unfortunately, there are a few cases in which the heuristic falls down: when there are no parked cars on the road, when the cars are parked facing in different directions (as seems to happen quite a lot in London), or when cars drive on either side of the street, as is the case in Bangalore.

Therefore, the major drawback with using a heuristic is that it is usually not possible to determine how it will perform all of the time — as just demonstrated. This leads to a level of uncertainty in the overall algorithm that may or may not be acceptable depending on the application.

In the end, though, whatever problem you are trying to solve, you will undoubtedly use an algorithm of some kind; and the simpler, more precise, and more understandable you can make your algorithm, the easier it will be to determine not only whether it works correctly but also how well it will perform.

Understanding Complexity in Relation to Algorithms

Having come up with your new, groundbreaking algorithm, how do you determine its efficiency? Obviously, you want your code to be as efficient as possible, so you'll need some way to prove that it will actually work as well as you had hoped. But what do we mean by efficient? Do we mean CPU time, memory usage, disk I/O, and so on? And how do we measure the efficiency of an algorithm?

One of the most common mistakes made when analyzing the efficiency of an algorithm is to confuse *performance* (the amount of CPU/memory/disk usage) with *complexity* (how well the algorithm scales). Saying that an algorithm takes 30 milliseconds to process 1,000 records isn't a particularly good indication of efficiency. While it is true that, ultimately, resource consumption is important, other aspects such as CPU time can be affected heavily by the efficiency and performance of the underlying hardware on which the code will run, the compiler used to generate the machine code, in addition to the code itself. It's more important, therefore, to ascertain how a given algorithm behaves as the size of the problem increases. If the number of records to process was doubled, for example, what effect would that have on processing time? Returning to our original example, if one algorithm takes 30 milliseconds to process 1,000 records while another takes 40 milliseconds, you might consider the first to be "better." However, if the first algorithm takes 300 milliseconds to process 10,000 records (ten times as many) and the second algorithm only takes 80 milliseconds, you might reconsider your choice.

Generally speaking, complexity is a measure of the amount of a particular resource required to perform a given function. While it is therefore possible — and often useful — to measure complexity in terms of disk I/O, memory usage, and so on, we will largely focus on complexity as it affects CPU time. As such, we will further redefine complexity to be a measure of the number of computations, or operations, required to perform a particular function.

Interestingly, it's usually not necessary to measure the precise number of operations. Rather, what is of greater interest is how the number of operations performed varies with the size of the problem. As in the previous example, if the problem size were to increase by an order of magnitude, how does that affect the number of operations required to perform a simple function? Does the number remain the same? Does it double? Does it increase linearly with the size of the problem? Does it increase exponentially? This is what we mean when we refer to algorithm complexity. By measuring the complexity of an algorithm, we hope to predict how it will perform: Complexity affects performance, but not vice versa.

Throughout this book, the algorithms and data structures are presented along with an analysis of their complexity. Furthermore, you won't require a Ph.D. in mathematics to understand them either. In all cases, a very simple theoretical analysis of complexity is backed by easy-to-follow empirical results in the form of test cases, which you can try for yourself, playing around with and changing the inputs in order to get a good feel for the efficiency of the algorithms covered. In most cases, the average complexity is given — the expected average-case performance of the code. In many cases, a worst-case and best-case time is also given. Which measure — best, worst, or average — is most appropriate will depend on the algorithm to some extent, but more often than not it is a function of the type of data upon which the algorithm will operate. In all cases, it is important to remember that complexity doesn't provide a precise measure of expected performance, but rather places certain bounds or limits on the achievable performance.

Understanding Big-O Notation

As mentioned earlier, the precise number of operations is not actually that important. The complexity of an algorithm is usually defined in terms of the *order of magnitude* of the number of operations required to perform a function, denoted by a capital `O` for *order of* — hence, big-O — followed by an expression representing some growth relative to the size of the problem denoted by the letter N. The following list shows some common orders, each of which is discussed in more detail a little later:

- `O(1)`: Pronounced "order 1" and denoting a function that runs in constant time
- `O(N)`: Pronounced "order N" and denoting a function that runs in linear time
- $O(N^2)$: Pronounced "order N squared" and denoting a function that runs in quadratic time
- `O(log N)`: Pronounced "order log N" and denoting a function that runs in logarithmic time
- `O(N log N)`: Pronounced "order N log N" and denoting a function that runs in time proportional to the size of the problem and the logarithmic time
- `O(N!)`: Pronounced "order N factorial" and denoting a function that runs in factorial time

Of course, there are many other useful orders besides those just listed, but these are sufficient for describing the complexity of the algorithms presented in this book.

Figure 1-1 shows how the various measures of complexity compare with one another. The horizontal axis represents the size of the problem — for example, the number of records to process in a search algorithm. The vertical axis represents the computational effort required by algorithms of each class. This is not indicative of the running time or the CPU cycles consumed; it merely gives an indication of how the computational resources will increase as the size of the problem to be solved increases.

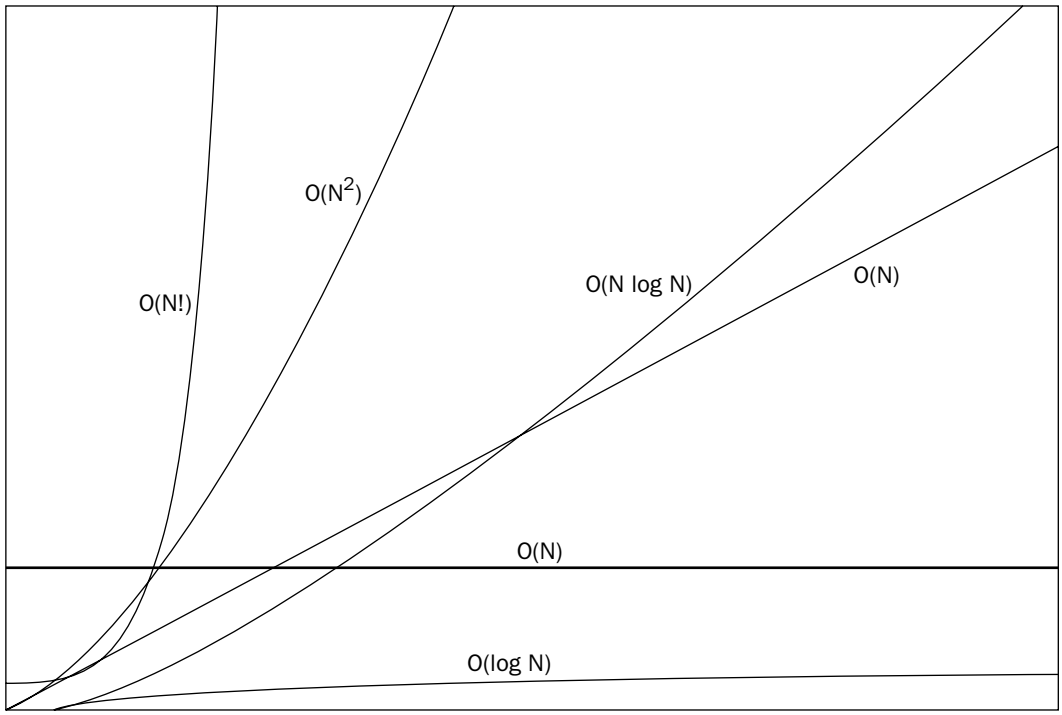

Figure 1-1: Comparison of different orders of complexity.

Referring back at the list, you may have noticed that none of the orders contain constants. That is, if an algorithm's expected runtime performance is proportional to `N`, `2×N`, `3×N`, or even `100×N`, in all cases the complexity is defined as being `O(N)`. This may seem a little strange at first — surely `2×N` is better than `100×N` — but as mentioned earlier, the aim is not to determine the exact number of operations but rather to provide a means of comparing different algorithms for relative efficiency. In other words, an algorithm that runs in `O(N)` time will generally outperform another algorithm that runs in $O(N^2)$. Moreover, when dealing with large values of N, constants make less of a difference: As a ration of the overall size, the difference between 1,000,000,000 and 20,000,000,000 is almost insignificant even though one is actually 20 times bigger.

Of course, at some point you will want to compare the actual performance of different algorithms, especially if one takes 20 minutes to run and the other 3 hours, even if both are `O(N)`. The thing to remember, however, is that it's usually much easier to halve the time of an algorithm that is `O(N)` than it is to change an algorithm that's inherently $O(N^2)$ to one that is `O(N)`.

Constant Time: O(1)

You would be forgiven for assuming that a complexity of `O(1)` implies that an algorithm only ever takes one operation to perform its function. While this is certainly possible, `O(1)` actually means that an algorithm takes constant time to run; in other words, performance isn't affected by the size of the problem. If you think this sounds a little too good to be true, then you'd be right.

Granted, many simple functions will run in `O(1)` time. Possibly the simplest example of constant time performance is addressing main memory in a computer, and by extension, array lookup. Locating an element in an array generally takes the same amount of time regardless of size.

For more complex problems, however, finding an algorithm that runs in constant time is very difficult: Chapter 3, "Lists," and Chapter 11, "Hashing," introduce data structures and algorithms that have a time complexity of `O(1)`.

The other thing to note about constant time complexity is that it still doesn't guarantee that the algorithm will be very fast, only that the time taken will always be the same: An algorithm that always takes a month to run is still `O(1)` even though the actual running time may be completely unacceptable.

Linear Time: O(N)

An algorithm runs in `O(N)` if the number of operations required to perform a function is directly proportional to the number of items being processed. Looking at Figure 1-1, you can see that although the line for `O(N)` continues upward, the slope of the line remains the same.

One example of this might be waiting in a line at a supermarket. On average, it probably takes about the same amount of time to move each customer through the checkout: If it takes two minutes to process one customer's items, it will probably take $2 \times 10 = 20$ minutes to process ten customers, and $2 \times 40 = 80$ minutes to process 40 customers. The important point is that no matter how many customers are waiting in line, the time taken to process each one remains about the same. Therefore, we can say that the processing time is directly proportional to the number of customers, and thus `O(N)`.

Interestingly, even if you doubled or even tripled the number of registers in operation at any one time, the processing time is still officially `O(N)`. Remember that big-O notation always disregards any constants.

Algorithms that run in `O(N)` time are usually quite acceptable. They're certainly considered to be as efficient as those that run in `O(1)`, but as we've already mentioned, finding constant time algorithms is rather difficult. If you manage to find an algorithm that runs in linear time, you can often make it more efficient with a little bit of analysis — and the occasional stroke of genius — as Chapter 16, "String Searching," demonstrates.

Quadratic Time: $O(N^2)$

Imagine a group of people are meeting each other for the first time and in keeping with protocol, each person in the group must greet and shake hands with every other person once. If there were six people in the group, there would be a total of $5 + 4 + 3 + 2 + 1 = 15$ handshakes, as shown in Figure 1-2.

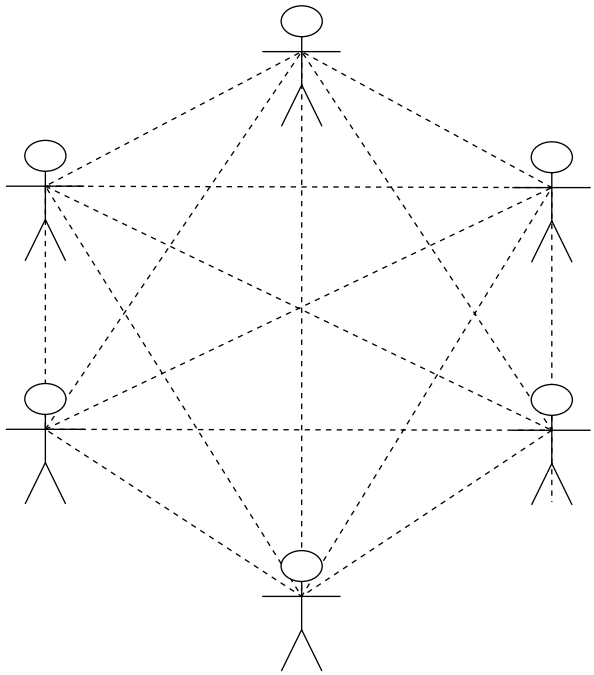

Figure 1-2: Each member of the group greets every other member.

What would happen if there were seven people in the group? There would be `6 + 5 + 4 + 3 + 2 + 1 = 21` handshakes. If there were eight people? That would work out to be `7 + 6 + . . . + 2 + 1 = 28` handshakes. If there were nine people? You get the idea: Each time the size of the group grows by one, that extra person must shake hands with every other person.

The number of handshakes required for a group of size N turns out to be $(N^2 - N) / 2$. Because big-O notation disregards any constants — in this case, the 2 — we're left with $N^2 - N$. As Table 1-1 shows, as N becomes larger, subtracting N from N^2 will have less and less of an overall effect, so we can safely ignore the subtraction, leaving us with a complexity of $O(N^2)$.

Table 1-1: Effect of Subtracting N from N^2 as N Increases

N	N^2	$N^2 - N$	Difference
1	1	0	100.00%
10	100	90	10.00%
100	10,000	9,900	1.00%
1,000	1,000,000	999,000	0.10%
10,000	100,000,000	99,990,000	0.01%

Algorithms that run in quadratic time may be a computer programmer's worst nightmare; any algorithm with a complexity of $O(N^2)$ is pretty much guaranteed to be useless for solving all but the smallest of problems. Chapters 6 and 7, on sorting, provide some rather interesting examples.

Logarithmic Time: O(log N) and O(N log N)

Looking at Figure 1-1, you can see that although $O(\log N)$ is better than $O(N)$ it's still not as good as O(1).

The running time of a logarithmic algorithm increases with the log — in most cases, the log base 2 — of the problem size. This means that even when the size of the input data set increases by a factor of a million, the run time will only increase by some factor of $\log(1{,}000{,}000) = 20$. An easy way to calculate the log base 2 of an integer is to work out the number of binary digits required to store the number. For example, the log base 2 of 300 is 9, as it takes 9 binary digits to represent the decimal number 300 (the binary representation is 100101100).

Achieving logarithmic running times usually requires your algorithm to somehow discard large portions of the input data set. As a result, most algorithms that exhibit this behavior involve searching of some kind. Chapter 9, "Binary Searching," and Chapter 10, "Binary Search Trees," both cover algorithms that run in $O(\log N)$.

Looking again at Figure 1-1, you can see that $O(N \log N)$ is still better than $O(N^2)$ but not quite as good as $O(N)$. Chapters 6 and 7 cover algorithms that run in $O(N \log N)$.

Factorial Time: O(N!)

You may not have thought so, but some algorithms can perform even more poorly than $O(N^2)$ — compare the $O(N^2)$ and $O(N!)$ lines in Figure 1-1. (Actually, there are many other orders that are far worse than even these but we don't cover any of them in this book.)

It's fairly unusual to encounter functions with this kind of behavior, especially when trying to think of examples that don't involve code, so in case you've forgotten what factorial is — or for those who never knew in the first place — here's a quick refresher:

> *The factorial of an integer is the product of itself and all the integers below it.*

For example, 6! (pronounced "six factorial") $= 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$ and $10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3{,}628{,}800$.

Table 1-2 provides a comparison between N^2 and N! for the integers between 1 and 10.

Table 1-2: Comparison between N^2 and N! for Small Integers

N	N^2	N!
1	1	1
2	4	2
3	9	6
4	16	24

N	N^2	N!
5	25	120
6	36	720
7	49	5,040
8	64	40,320
9	81	362,880
10	100	3,628,800

As you can see, for values of `N` up to and including `N=2`, the factorial is less than the quadratic, after which point the factorial takes off and leaves everything else in its wake. As a consequence, even more so than with $O(N^2)$, you'd better hope that your code isn't `O(N!)`.

Unit Testing

Before continuing our journey into the realm of algorithms, we need to digress to discuss a topic that's very dear to our hearts: *unit testing*. Over the past several years, unit testing has become very popular among developers who place a high value on the quality of the systems they build. Many of these developers are not comfortable creating software without also creating an accompanying suite of automated tests that prove the software they've created does what they intend. As you may have guessed, we both hold this point of view. That's why for every algorithm we show you, we'll also show you how it works and what it does by providing unit tests for it. We strongly recommend that you adopt this habit in your own development efforts. It will greatly increase your odds of leaving work on time!

The next few sections provide a quick overview of unit testing in general and introduce the JUnit framework for unit testing Java programs. We use JUnit throughout the book, so you'll need to be familiar with it in order to make sense of the examples provided. Feel free to skip this section of the book if you're already a hardcore test-infected developer. Good for you!

What Is Unit Testing?

A *unit test* is simply a program that tests another program. In Java terms, it's usually a Java class whose purpose is to test another class. That's really it. Like most things, though, it's easy to learn but hard to master. Unit testing is an art as well as a science, and you can find many books just about testing, so we won't go into too much depth here. Check Appendix A for some good books with more detail on this topic.

The basic operation of a unit test is as follows:

1. Set up any objects that you need to support the test, such as sample data. This stuff is called *fixtures*.
2. Run your test by exercising the objects and ensuring that what you expected to happen did indeed happen. This is called making *assertions*.
3. Finally, clean up anything no longer needed. This is called *tearing down*.

The common convention when naming unit tests in this book is to create the test class using the class name, followed by `Test`. For example, if you are going to test a class called `Widget`, you create a new class called `WidgetTest` to act as the unit test for it. You will see many examples of this in the book. You should also notice a common convention for organizing source files. This involves placing unit tests in a parallel source tree with the same package structure as the main source files. For example, if the Widget lives in a package called `com.wrox.algorithms`, the source files are organized something like what you see in Figure 1-3.

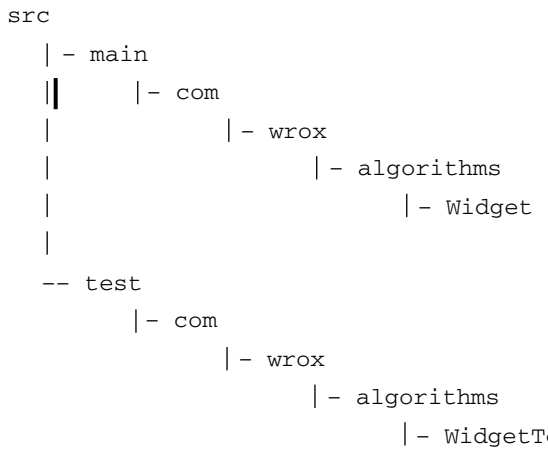

Figure 1-3: Unit test source files live in a parallel package structure.

This means that the Java package statement at the top of each file would be exactly the same, but the files themselves live in different directories on the file system. This model keeps production code separate from test code for easy packaging and distribution of the main line of code, and makes it easy to ensure that production code doesn't rely on test code during the build process, by having slightly different classpath arrangements when compiling the two directories. Some people also like the fact that it can enable tests to access package-scoped methods, so that's something else to consider.

Before we finish describing unit testing, be aware that you may come across several other common types of testing. We'll provide some basic definitions here to give you some context and to avoid some unnecessary confusion. This book only makes use of unit testing, so check the references for additional information about other types of testing. Some of the terms you may encounter include the following:

- **Black-box testing:** Imagine you want to test your DVD player. All you have access to (without voiding your warranty) are the buttons on the front and the plugs at the back. You can't test individual components of the DVD player because you don't have access to them. All you have is the externally visible controls provided on the outside of the black box. In software terms, this is akin to only being able to use the user interface for a fully deployed application. There are many more components, but you may not have access to them.
- **Functional testing:** This is usually used interchangeably with black-box testing.
- **White-box testing:** This refers to testing that can get inside the overarching component organization of a system to a greater or lesser extent and test individual components, usually without the user interface.

- **Integration testing:** This is often used to describe the testing of an individual component of a large distributed system. These types of tests are aimed at ensuring that systems continue to meet their agreed contracts of behavior as they evolve independently from one another.

Unit testing is the most fine-grained of the testing techniques, as it involves usually a single class being tested independently of any other classes. This means unit tests are very quick to run and are relatively independent of each other.

Why Unit Testing Is Important

To understand why you're reading so much about unit testing in a book about algorithms, consider your Java compiler, which enables you to run your Java programs. Would you ever consider a program you wrote to be working if you hadn't compiled it? Probably not! Think of your compiler as one kind of test of your program — it ensures that your program is expressed in the correct language syntax, and that's about it. It cannot give you any feedback regarding whether your program does anything sensible or useful, and that's where unit tests come in. Given that we are more interested in whether our programs actually do something useful than whether we typed in the Java correctly, unit tests provide an essential barrier against bugs of all kinds.

Another benefit of unit tests is that they provide reliable documentation about the behavior of the class under test. When you've seen a few unit tests in action, you'll find that it's easier to work out what a class is doing by looking at the test than by looking at the code itself! (The code itself is where to look when you want to know *how* it does whatever it does, but that's a different matter.)

A JUnit Primer

The first place to visit is the JUnit website at `www.junit.org/`. Here you will find not only the software to download, but also information on how to use JUnit in your IDE, and pointers to the many extensions and enhancements to JUnit that have been created to address particular needs.

Once you've downloaded the software, all you need to do is add `junit.jar` to your classpath and you're ready to create your first unit test. To create a unit test, simply create a Java class that extends the `junit.framework.TestCase` base class. The following code shows the basic structure of a unit test written using JUnit:

```
package com.wrox.algorithms.queues;

import com.wrox.algorithms.lists.LinkedList;
import com.wrox.algorithms.lists.List;
import junit.framework.TestCase;

public class RandomListQueueTest extends TestCase {
    private static final String VALUE_A = "A";
    private static final String VALUE_B = "B";
    private static final String VALUE_C = "C";

    private Queue _queue;
    ...
}
```

Don't be concerned with what this unit test is actually testing; this particular unit test is one you'll understand during the discussion on queues later in the book. The main point here is that a unit test is just a regular class with a base class supplied by the JUnit framework. What this code does is declare the class, extend the base class, and declare some static members and one instance member to hold the queue you're going to test.

The next thing to do is override the `setUp()` method and add any code needed to get the objects ready for testing. In this case, this simply means calling the overridden `setUp()` method in the superclass and instantiating your queue object for testing:

> *Note the spelling of the* `setUp()` *method. That's a capital "U" in the middle of it! One of Java's weaknesses is that methods are only overridden by coincidence, not by explicit intention. If you mistype the name of the method, it won't work as you expect.*

```
protected void setUp() throws Exception {
    super.setUp();

    _queue = new RandomListQueue();
}
```

Part of what is provided to you by the JUnit framework is the guarantee that each time a test method is run (you'll get to those shortly), the `setUp()` method will be called before each test runs. Similarly, after each test method is run, a companion `tearDown()` method provides you with the opportunity to clean up after yourself, as shown by the following code:

```
protected void tearDown() throws Exception {
    super.tearDown();

    _queue = null;
}
```

You might be wondering why you need to bother with setting the instance member field to `null`. While not strictly necessary, in very large suites of unit tests, neglecting this step can cause the unit tests to consume a lot more memory than they need to, so it's a good habit to acquire.

The following method of actual unit test code is designed to test the behavior of a queue when it is empty and someone tries to take an item off it, which is not allowed by the designer of the object. This is an interesting case because it demonstrates a technique to prove that your classes fail in expected ways under improper use. Here's the code:

```
public void testAccessAnEmptyQueue() {
    assertEquals(0, _queue.size());
    assertTrue(_queue.isEmpty());

    try {
        _queue.dequeue();
        fail();
    } catch (EmptyQueueException e) {
        // expected
    }
}
```

Note the following points about this code:

- The method's name begins with `test`. This is required by the JUnit framework to enable it to differentiate a test method from a supporting method.
- The first line of the method uses the `assertequals()` method to prove that the size of the queue is zero. The syntax of this method is `assertEquals(expected, actual)`. There are overloaded versions of this method for all basic Java types, so you will become very familiar with this method during the course of this book. It is probably the most common assertion in the world of unit tests: making sure that something has the value you expect. If, for some reason, the value turns out to be something other than what you expect, the JUnit framework will abort the execution of the test and report it as a failure. This helps to make the unit test quite concise and readable.
- The second line uses another very common assertion, `asserttrue()`, which is used to ensure that Boolean values are in the expected state during the test run. In this case, we are making sure that the queue reports correctly on its empty state.
- The `try/catch` block surrounds a call to a method on our queue object that is designed to throw an exception when the queue is empty. This construct is a little different than what you'll be used to from normal exception handling in Java, so look at it carefully. In this case, it's considered *good* if the exception is thrown and *bad* if it is not thrown. For this reason, the code does nothing in the `catch` block itself, but in the `try` block, it calls the JUnit framework `fail()` method right after calling the method you're trying to test. The `fail()` method aborts the test and reports it as a failure, so if the method throws the expected exception, then execution will fall through to the end of the method and the test will pass. If no exception is thrown, then the test will immediately fail. If that all sounds a little confusing, read through it again!

Here is another example of a unit test method in the same class:

```
public void testClear() {
    _queue.enqueue(VALUE_A);
    _queue.enqueue(VALUE_B);
    _queue.enqueue(VALUE_C);

    assertEquals(3, _queue.size());
    assertFalse(_queue.isEmpty());

    _queue.clear();

    assertEquals(0, _queue.size());
    assertTrue(_queue.isEmpty());
}
```

The method name again starts with `test` so that JUnit can find it using reflection. This test adds a few items to the queue, asserts that the `size()` and `isEmpty()` methods work as expected, and then clears the queue and again ensures that these two methods behave as expected.

The next thing you'll want to do after writing a unit test is to run it. Note that your unit test does not have a `main()` method, so you can't run it directly. JUnit provides several *test runners* that provide different interfaces — from a simple text-based console interface to a rich graphical interface. Most Java development environments, such as Eclipse or IntelliJ IDEA, have direct support for running JUnit tests,

but if all you have is the command line, you can run the preceding test with the following command (you will need to have `junit.jar` on your classpath, of course):

```
java junit.textui.TestRunner com.wrox.algorithms.queues.RandomListQueueTest
```

Running the graphical version is just as easy:

```
java junit.swingui.TestRunner com.wrox.algorithms.queues.RandomListQueueTest
```

JUnit can also be used from within many tools, such as Ant or Maven, that you use to build your software. Including the running of a good unit test suite with every build of your software will make your development life a lot easier and your software a lot more robust, so check out the JUnit website for all the details.

Test-Driven Development

All the algorithms and data structures we present in the book include unit tests that ensure that the code works as expected. In fact, the unit tests were written *before* the code existed to be tested! This may seem a little strange, but if you're going to be exposed to unit testing, you also need to be aware of an increasingly popular technique being practiced by developers who care about the quality of the code they write: *test-driven development*.

The term *test-driven development* was coined by Kent Beck, the creator of eXtreme Programming. Kent has written several books on the subject of eXtreme Programming in general, and test-driven development in particular. The basic idea is that your development efforts take on a rhythm, switching between writing some test code, writing some production code, and cleaning up the code to make it as well designed as possible (refactoring). This rhythm creates a constant feeling of forward progress as you build your software, while building up a solid suite of unit tests that protect against bugs caused by changes to the code by you or someone else further down the track.

If while reading this book you decide that unit testing is something you want to include in your own code, you can check out several books that specialize in this topic. Check Appendix A for our recommendations.

Summary

In this chapter, you learned the following:

- Algorithms are found in everyday life.
- Algorithms are central to most computer systems.
- What is meant by algorithm complexity.
- Algorithms can be compared in terms of their complexity.
- Big-O notation can be used to broadly classify algorithms based on their complexity.
- What unit testing is and why it is important.
- How to write unit tests using Junit.