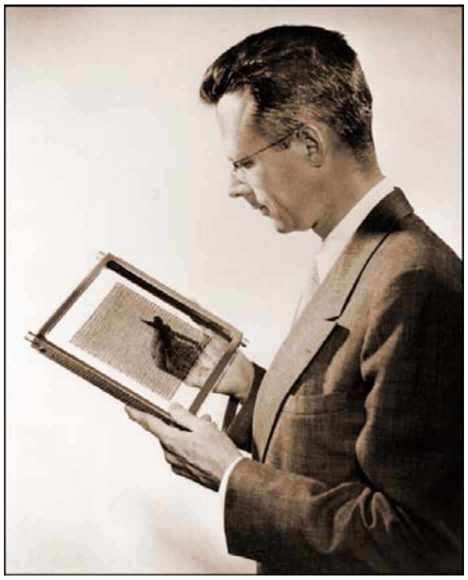
Chapter 7
The Object Memory Model

A teacher who can arouse a feeling for one single good action, for one single good poem, accomplishes more than he who fills our memory with rows on rows of natural objects, classified with name and form.

— Goethe, *Elective Affinities,* 1808



Jay Forrester

After growing up on a Midwestern cattle ranch without electricity, Jay Forrester studied electrical engineering at the University of Nebraska and MIT, where he became director of the Navy's Project Whirlwind in 1944. Along with the ENIAC system at the Moore School in Philadelphia and the MARK I system at Harvard, Whirlwind played a central role in the early history of computers as they evolved from earlier analog designs to the digital systems that are standard in the industry today. Forrester's most significant contribution to computer hardware design was the development of core memory, in which small ferrite disks could be magnetized in one direction or the other to represent a binary 0 or 1. Magnetic core memory revolutionized hardware designs and was used in essentially all computers until it was replaced by integrated-circuit memory in the late 1970s. In 1956, Forrester joined the faculty of the Sloan School of Management, where he founded the new discipline of system dynamics, which attempts to focus holistically on large-scale systems and their interactions rather than looking only at their individual parts.

One of the enormous advantages of high-level languages like Java is that they free you from having to worry about the details of low-level representation inside the hardware of the machine. Managing the details of the internal representation is typically time-consuming, tedious, and susceptible to error; having the language keep track of these details can therefore increase programmer productivity. The problem, however, is that it is much more difficult to understand the programming process without having some idea of how that internal representation works. The structures that programmers have historically used—and that continue to form the foundation of languages like Java—reflect the capabilities of the hardware on which programs run. Knowing something about those low-level machine capabilities provides a conceptual framework for understanding programming at a higher, more abstract level.

Developing an intuitive feel for how data representation works inside the machine is particularly important when you begin to work with objects instead of primitive data. In Java, passing an object as an argument to a method seems very different from the corresponding process of using primitive types as arguments. If you understand how objects are represented inside the computer, however, the reasons for that apparent difference become much clearer and in fact turn out to be far more consistent than it might initially appear.

7.1 The structure of memory

In Chapter 1, you had the opportunity to learn a little bit about the internal structure of a computer, but only at a very abstract level. Figure 1-1 identified the major hardware components of a typical computer system including the CPU, memory, secondary storage, and I/O devices. To develop a mental model of how objects are stored inside a computer, it is important to look at the structure of the memory system in more detail.

Bits, bytes, and words

At the most primitive level, all data values inside the computer are stored in the form of fundamental units of information called *bits*. A **bit** records the simplest possible value, which can be in one of two possible states. If you think of the circuitry inside the machine as if it were a tiny light switch, you might label those states as *on* and *off*. Because the word *bit* comes originally from a contraction of *binary digit*, it is more common to label those states using the values 0 and 1, which are the two digits used in the binary number system on which computer arithmetic is based.

Since a single bit holds so little information, the individual bits do not provide a convenient mechanism for storing data. To make it easier to store such traditional types of information as numbers or characters, bits are collected together into larger units that are then treated as integral units of storage. The smallest combined unit is called a **byte** and is composed of eight individual bits. On most machines, bytes are assembled into larger structures called **words**—defined to be four bytes in Java—that are large enough to hold an integer value.

The amount of memory available to a particular computer varies over a wide range. Early machines supported memories whose size was measured in kilobytes (KB); today's machines have memory sizes measured in megabytes (MB) or even gigabytes (G172es *kilo, mega,* and *giga* stand for one thousand, one million, and one billion, respectively. In the world of computers, however, those base-10 values do not fit well into the internal structure of the machine. By tradition, therefore, these prefixes are taken to represent the power of two closest to their traditional interpretations. Thus, in programming, the prefixes *kilo, mega,* and *giga* have the following meanings:

kilo (K)	=	2^{10}	=	1,024
mega (M)	=	2^{20}	=	1,048,576
giga (G)	=	2^{30}	=	1,037,741,824

A 64KB computer from the early 1970s would have had 64 x 1024 or 65,536 bytes of memory. Similarly, a 512MB machine would have 512 x 1,048,576 or 536,870,912 bytes of memory.

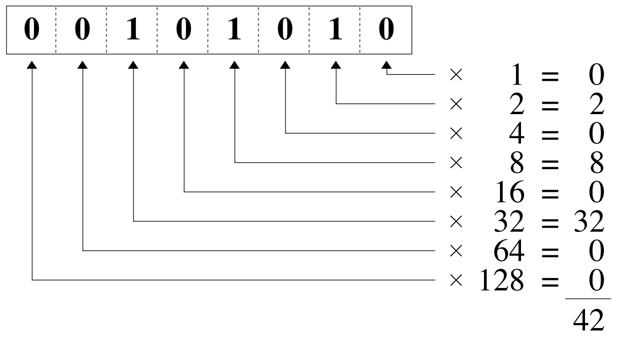
Binary and hexadecimal representations

Each of the bytes inside a machine holds data whose meaning depends on how the system interprets the individual bits. Depending on the hardware instructions that are used to manipulate it, a particular sequence of bits can represent an integer, a character, or a floating-point value, each of which requires some kind of encoding scheme. The easiest encoding scheme to describe is that for integers, in which the bits are used to represent an integer represented in **binary notation**, in which the only legal values are 0 and 1, just as is true for the underlying bits. Binary notation is similar in structure to our more familiar decimal notation, but uses 2 rather than 10 as its base. The contribution that a binary digit makes to the entire number depends on its position within the number as a whole. The rightmost digit represents the units field, and each of the other positions counts for twice as much as the digit to its right.

Consider, for example, the eight-bit byte containing the following binary digits:

0	**0**	**1**	**0**	**1**	**0**	**1**	**0**

That sequence of bits represents the number forty-two, which you can verify by calculating the contribution for each of the individual bits, as follows:

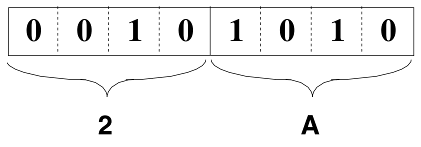


This diagram illustrates how to map an integer into bits using binary notation, but is not a particularly convenient representation. Binary numbers are cumbersome, mostly because they tend to be so long. Decimal representations are intuitive and familiar, but make it difficult to understand how the number translates into bits. For those applications in which it is useful to understand how a number translates into its binary representation without having to work with binary numbers that stretch all the way across the page, computer scientists tend to use **hexadecimal** (base 16) representation instead.

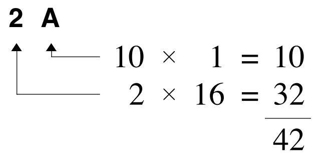
In hexadecimal notation, there are sixteen digits, representing values from 0 to 15. The decimal digits 0 through 9 are perfectly adequate for the first ten digits, but classical arithmetic does not define the extra symbols you need to represent the remaining six digits. Computer science traditionally uses the letters **A** through **F** for this purpose, as follows:

Hex digit	Value
A	10
B	11
C	12
D	13
E	14
F	15

What makes hexadecimal notation so attractive is that you can quickly move back and forth between hexadecimal values and the underlying binary representation. All you need to do is combine the bits into groups of four. For example, the number forty-two can be converted from binary to hexadecimal like this:



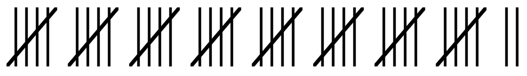
The first four bits represent the number 2, and the next four represent the number 10. Converting each of these to the corresponding hexadecimal digit gives **2A** as the base-16 form. You can then verify that this number still has the value 42 by adding up the independent digit values, as follows:



For the most part, the numeric representations that appear in this book use decimal notation for readability. If the base is not clear from the context, the text follows the usual strategy of using a subscript to denote the base. Thus, the three most common representations for the number forty-two—decimal, binary, and hexadecimal—look like this:

$$42_{10} = 101010_2 = 2A_{16}$$

The key point is that the number itself is always the same; the numeric base affects only the representation. Forty-two has a real-world interpretation that is independent of the base. That real-world interpretation is perhaps easiest to see in the representation an elementary school student might use, which is after all just another way of writing it down:

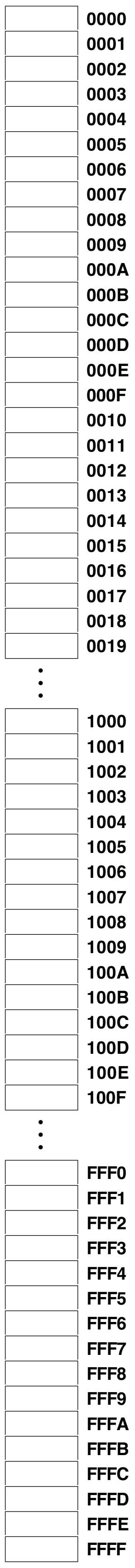


The number of line segments in this representation is forty-two. The fact that a number is written in binary, decimal, or any other base is a property of the representation and not of the number itself.

Memory addresses

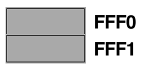
Within the memory system of a typical computer, every byte is identified by a numeric **address.** The first byte in the computer is numbered 0, the second is numbered 1, and so on, up to the number of bytes in the machine. For example, you can diagram the memory bytes of a tiny 64KB computer as shown along the right margin of this page. The numbering scheme, however, may seem unfamiliar at first, but only because the addresses are written using hexadecimal rather than decimal values. Hexadecimal notation was introduced in the preceding section, along with techniques for converting a hexadecimal value to its binary or decimal equivalent. The important thing to remember is that the addresses are simply numbers and that the base is relevant only to how those numbers are written down. The final address in the diagram is the address **FFFF**, which corresponds to the decimal value 65535. It would have been easy to write the addresses in decimal notation, which would have made them a little easier to read as numbers. This text, however, uses hexadecimal for the following reasons:

1. Address numbers are conventionally written in hexadecimal, and Java debuggers and runtime environments will generally display addresses in this form.
2. Writing address numbers in their hexadecimal form and using a sans-serif font makes it easier to recognize that a particular number represents an address rather than some unidentified integer.
3. Using hexadecimal values makes it easier to see why particular limits are chosen. When you write it as a decimal value, the number 65535 seems like a rather random value. If you express that same number in hexadecimal as **FFFF**, it becomes easier to recognize that this value is the largest value that can be represented in 16 bits.
4. Making the numbers seem less familiar may discourage you from thinking about them arithmetically. Java does a wonderful job of hiding the underlying address calculations from the programmer. What is important to understand is that addresses are represented as numbers. It is completely unimportant—and indeed usually impossible to determine—what those numbers are.

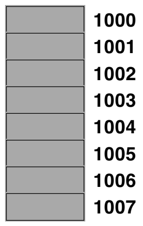


Memory

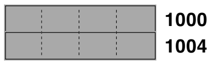
In earlier days, byte values were useful in and of themselves because characters could be represented using a single byte. Today, bytes are too small to represent the large character sets found in a modern computer. The Unicode representation discussed in Chapter 8, for example, requires two bytes for each character. A value of type `char` therefore takes up two of these byte-sized units in memory, as illustrated by the shaded bytes in the following diagram:



Data values requiring multiple bytes are identified by the address of the first byte, so that the character represented by the shaded area is considered to be at address **FFF0**. As a second example, values of type `double` require eight bytes of memory, so that a variable of type `double` stored at address **1000** would take up all the bytes between addresses **1000** and **1007**, inclusive:

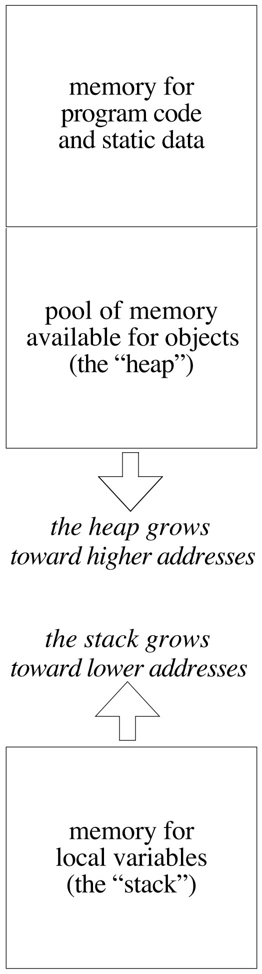


Although addresses in memory are usually specified in terms of bytes, the fact that bytes are too small to be of much use has led hardware designers to group consecutive bytes into larger units. Typical architectures today include instructions for manipulating memory units in the following sizes: one byte (8 bits), two bytes (16 bits), four bytes (32 bits), and eight bytes (64 bits). These units correspond to the built-in Java types **`byte`**, **`short`**, **`int`**, and **`long`**. The four-byte unit used to represent an integer is conventionally called a **word.** Because words tend to be much more useful, it is common to draw memory diagrams in units of words rather than the individual bytes. The **`double`** stored in the bytes numbered between **`1000`** and **`1007`** could therefore be diagrammed more compactly in the following word-based form:



7.2 Allocation of memory to variables

Whenever you declare a variable in a program, the compiler must reserve memory space to hold its value. The process of reserving memory space is called **allocation**. In Java, memory is allocated from one of three different sources depending on how it is declared, each of which is illustrated in the diagram to the right. The three allocation strategies are as follows:


memory for program code and static data
pool of memory available for objects (the "heap")
the heap grows toward higher addresses
the stack grows toward lower addresses
memory for local variables (the "stack")

Memory regions

1. *Static variables and constants*. Any variable whose declaration includes the **`static`** keyword applies to the class as a whole and not to an individual object. These variables are typically allocated at the beginning of the memory space allocated to the program alongside the memory cells used to store the instructions for the program. In the examples included in this book, the only static declarations are those for named constants.
2. *Dynamically allocated objects*. All objects created using the **`new`** keyword are assigned storage from a region of memory called the **heap.** Although there is no explicit rule that enforces this convention, most implementations of the Java Virtual Machine assign heap memory beginning immediately after the fixed region assigned to the static declarations in a class, as illustrated in the diagram. The advantage of doing so is that this allocation strategy ensures that the program can take advantage of as much memory as possible. Because the stack and the heap start at opposite ends of memory and grow toward each other, neither will run out until all available space is exhausted.
3. *Local variables*. All variables that are declared as local variables inside a method are allocated from a region of memory called the **stack,** which is typically implemented at least partially in hardware. In modern architectures, the stack begins at the highest legal address in memory and grows toward lower addresses as new methods are

called. Calling a method increases the size of the stack by the amount needed to hold the local variables that method declares. Collectively, the memory assigned to the local variables for a method are called a **stack frame.** When a method returns, its stack frame is discarded, restoring the frame of its caller.

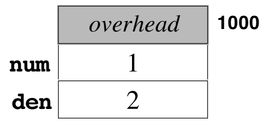
It is important to recognize that declaring an object variable is likely to reserve memory in both the stack and the heap. The local variable used to store the object appears on the stack just like any local variable. That variable, however, holds only the memory address of the object for which the actual data is stored in the heap. As noted in Chapter 6, that address is called a **reference.**

At this point, it is useful to consider a simple example to illustrate the relationship between heap storage and stack storage. Suppose that you have decided to use the **`Rational`** class introduced in section 6.3, which allows for a precise representation of fraction numbers. Internally, each **`Rational`** object declares two instance variables, **`num`** and **`den`**, to store the numerator and denominator of the fraction, respectively. The important question is then what happens inside the computer when you execute a declaration such as the following:

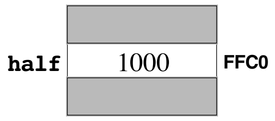
```
Rational half = new Rational(1, 2);
```

The variable **`half`** is a local variable and therefore appears in the stack frame for its caller. That variable, however, contains only enough storage to hold a memory address, which is typically four bytes on a machine with 32-bit addressing, although the growing number of 64-bit architectures means that the memory required will presumably grow to eight bytes over time. In any event, the amount of memory that appears on the stack is small. No matter how large the object itself turns out to be, a *reference* to that object can be stored in a single word.

The complete picture of what happens during the execution of this Java statement requires looking at what happens in both the heap and the stack. The evaluation of the initial-value expression on the right-hand side of the equal sign creates a new object in the heap. The heap storage for that object contains space for the integer values **`num`** and **`den`** along with some additional information common to all objects, which is used by the Java runtime system to manage objects in memory. From your perspective as a programmer, the details of that additional information are of no consequence, but it sometimes helps to know that objects take up more memory than you might expect from looking at the memory requirements of their instance variables. In this book, this extra management information is called the **object overhead** and is represented as a shaded area in memory diagrams. For example, the object created by the call to **`new Rational(1, 2)`** might look like this:



Once the new **`Rational`** object has been allocated in the heap, the next step in the declaration process is to store the address of that object in the variable **`half`**. That variable is allocated on the stack and therefore appears in a stack frame somewhere toward the high end of memory on most machines. Although there is no way to know what other values might exist in the stack frame without seeing the code for the entire method, the word assigned to **`half`** might look like this:



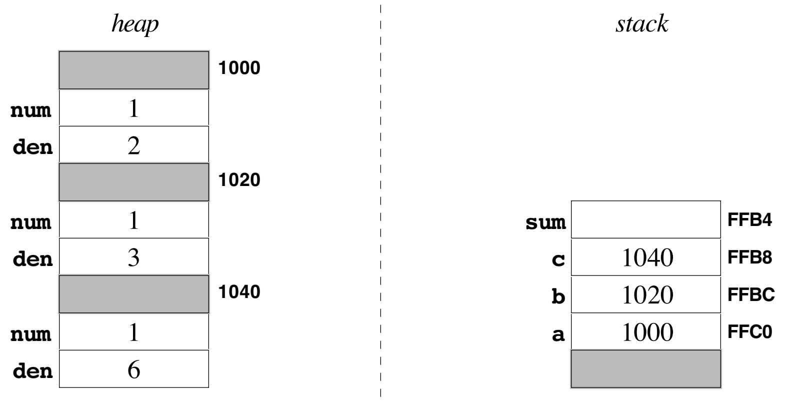
The value 1000 shown in the variable **half** is not intended as an integer but rather to signify the address at which the actual object is located. The choice of 1000 for that address—along with the addresses of every other heap and stack variable used in this book—is entirely arbitrary. In general, there is no way for you as a Java programmer to know the address of an object, nor is there any reason for you to do so. It is, however, important for you to remember that every object has some address and that Java programs can gain access to the data in the object by keeping track of the address at which that object lives.

Even after you become more familiar with objects, you will discover that it often helps to create diagrams that make it easier to follow what's happening inside the memory. To illustrate this process, it is useful to try a somewhat more elaborate example. Consider the following **run** method, which first appeared in Chapter 6:

```
public void run() {
   Rational a = new Rational(1, 2);
   Rational b = new Rational(1, 3);
   Rational c = new Rational(1, 6);
   Rational sum = a.add(b).add(c);
   println(a + " + " + b + " + " + c + " = " + sum);
}
```

The idea of this program, of course, is to add three **Rational** variables and display their sum. Our concern here, however, is simply to understand how these values are represented inside the machine.

The first three declarations are easy enough. As in the simple example of the variable **half**, each of these declarations constructs a new **Rational** object and assigns its address to a variable in the stack frame for the **run** method, giving rise to a memory diagram that looks like this:



As in most languages, Java creates the complete stack frame for a method at the time the method is called, so that all four local variables—**a**, **b**, **c**, and **sum**—have been allocated on the stack, but only the first three have been initialized at this point in the execution. The value used to initialize **sum** is the result of the expression

```
a.add(b).add(c)
```

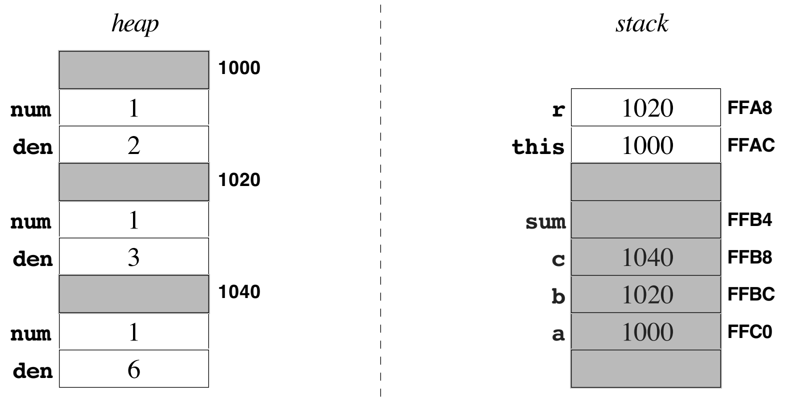
It is worth tracing through this operation step by step.

The first step in the process is to invoke the **add** method on the **Rational** object **a**, passing it the value of **b** as a parameter. Because this operation is a method call, a new stack frame must be created for it, binding the formal parameters of the method to the values of the actual arguments included in the call. As you can see from Figure 6-9, the code for the **add** method appears as follows:

```
public Rational add(Rational r) {
   return new Rational(this.num * r.den + r.num * this.den,
                       this.den * r.den);
}
```

The formal parameter is named **r** inside the **add** method, which means that **r** must be assigned the value of the actual argument, which is named **b** in the calling frame. The value currently sitting in the variable **b** is the number 1020, which is the address of the corresponding object in the heap. Java initializes the variable **r** in the new frame simply by copying the address, which means that the variable **r** in the new frame will also contain 1020.

For this example, it is important to note that the stack frame for any method invoked on a receiving object also contains a reference to that object, which Java identifies using the keyword **this**. Intuitively, it is best to think of **this** as simply a local variable that refers to the receiving object, which in this case is the value stored in the variable **a**. As with the argument value, the identity of the receiver is also copied into the new frame as a simple address. Thus, the state of the memory once the new frame has been created looks like this:



In this diagram, the stack region below the current frame is shown in gray to emphasize that the shaded region is inaccessible to the **add** method. The only variables accessible to **add** are the references stored in the variables **r** and **this** in the current frame. These reference variables are exactly what the **add** method needs to perform its computation. The **add** method returns a new **Rational** value whose numerator is given by the expression

```
this.num * r.den + r.num * this.den
```

and whose denominator has the value

```
this.den * r.den
```

Each term in these expressions is a selection operation that extracts one of the two fields, **num** or **den**, from the appropriate **Rational** value. For example, the term **r.den** specifies the selection of the **den** field of the value whose reference is stored in **r**. Since **r** contains the address 1020, the term **r.den** indicates the denominator component of the object stored at address 1020, which is the integer 3. Expanding the value for each term in a similar fashion reveals that the result of the **add** method is given by the expression

```
new Rational(1 * 3 + 1 * 2, 2 * 3)
```

which can be further simplified to

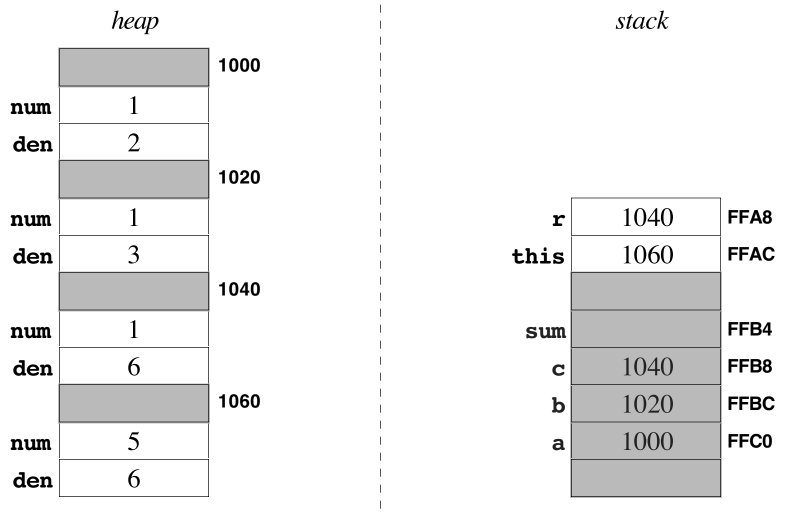
```
new Rational(5, 6)
```

This expression constructs a new **Rational** object with the indicated components in the heap and returns the address of that object as the value of the call **a.add(b)**, clearing away the stack frame for **add** as it returns.

Unlike the result of most of the computations you have seen so far, the new value computed by this method does not get stored in a variable. The calculation **a.add(b)** is only a subexpression of the longer expression

```
a.add(b).add(c)
```

The value of **a.add(b)** therefore becomes the receiver of a new **add** call with the argument **c**. Setting up the frame for this new method call leaves memory in the following state, with the new **Rational** object shown at address 1060 in the heap:



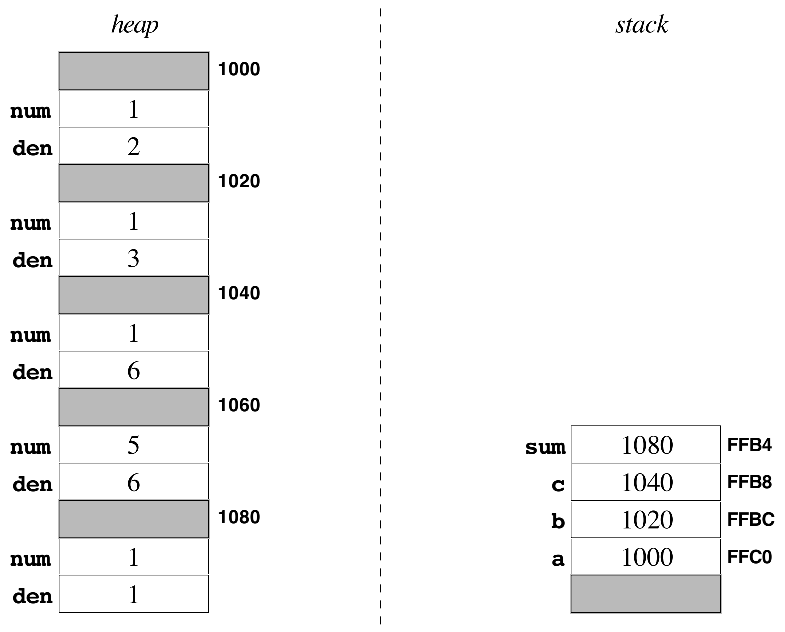
Once again, the evaluation of the **add** method requires determining the values of the fields in the two **Rational** values accessible from the current frame. This time, the straightforward expansion of the result expression gives

```
new Rational(5 * 6 + 1 * 6, 6 * 6)
```

which works out to

```
new Rational(36, 36)
```

Fortunately, the constructor for **Rational** provides additional simplification of the result by using Euclid's algorithm to reduce the fraction to lowest terms. Eliminating the common factor of 36 from both the numerator and the denominator gives rise to a new **Rational** object with 1 as the value of both its **num** and **den** field. When that value is assigned to the variable **sum**, the heap and stack look like this:



Garbage collection

The example from the preceding section illustrates an interesting wrinkle that often arises when objects are stored in memory. If you look carefully at the most recent diagram, you will see that the heap contains five different **`Rational`** objects even though only four of those objects have references on the stack. The fraction 5/6 stored at address 1060 was created during the calculation as the intermediate result from the subexpression

```
a.add(b)
```

While that value is necessary to complete the calculation, it becomes irrelevant once the final addition is performed. In the traditional parlance of computer science, that value is now **garbage.** Calculations in object-oriented languages often generate quite a bit of garbage along the way as intermediate results are computed as part of complex computations. Unfortunately, those values occupy space in the heap. If that garbage is allowed to persist, the heap will eventually fill up, even if there is plenty of room for the active data.

To get around the problem of having the heap fill up with objects that are no longer useful, the Java runtime system adopts a strategy called **garbage collection,** which does pretty much exactly what it sounds like it does. Whenever the memory available in the heap seems to be running short, Java defers what it's doing to collect any garbage in the heap and return that memory to the pool of assignable storage. To do so, Java uses a two-phase process that operates in more or less the following way, although the strategy presented here has been simplified to ensure that the basic idea remains clear:

1. Go through every variable reference on the stack and every variable reference in static storage and mark the corresponding object as being "in use" by setting a flag that lives in the object's overhead space. In addition, whenever you set the "in use" flag for an object, determine whether that object contains references to other objects and mark those objects as being "in use" as well.
2. Go through every object in the heap and delete any objects for which the "in use" flag has not been set, returning the storage used for that object to the heap for subsequent reallocation. It must be safe to delete these objects, because there is no way for the

program to gain access to them, given that no references to those objects still exist in the program variables.

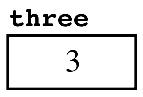
Because the garbage-collection algorithm operates in two phases—one to mark the objects that are in use and one to sweep through memory collecting inaccessible objects—this strategy is traditionally called a **mark-and-sweep** collector.

7.3 Primitive types vs. objects

From a high-level conceptual perspective, you can often think about Java objects in much the same way that you think about the primitive types like **`int`** and **`double`**. If you declare an integer variable by writing

```
int three = 3;
```

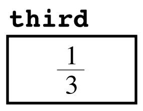
you can diagram that variable by drawing a box named **`three`** and writing a 3 in the box, as follows:



To a certain extent, you can treat objects in the same way. For example, if you create a new **`Rational`** variable using the declaration

```
Rational third = new Rational(1, 3);
```

you can think about this variable conceptually as if it held the actual value as shown in the following diagram:



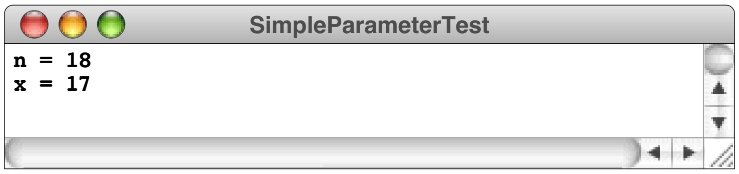
It is, however, important to understand at what point this symmetry breaks down. As you know from the discussion of internal representation earlier in this chapter, the variable **`third`** does not in fact contain the actual object but instead contains the address of that object. This mode of representing objects has several implications that it is important for you, as a budding programmer, to understand.

Parameter passing

One of the most important consequences of the representation scheme that Java uses for objects arises when you pass an object as a parameter to a method. At first glance, the rules for passing objects as parameters seem different from the corresponding rules for passing primitive values. Suppose, for example, that you have written the following pair of methods:

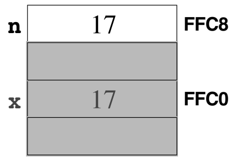
```
public void run() {
   int x = 17;
   increment(x);
   println("x = " + x);
}

private void increment(int n) {
   n++;
   println("n = " + n);
}
```

Running this program produces the following output:



```
n = 18
x = 17
```

As you can see from the output, the `++` operator in the **`increment`** method affects only the value of the local variable **`n`** and not the value of **`x`** in the calling frame. The behavior simply reflects the way these values are stored on the stack. When **`increment`** is called, the value from the stack entry for the local variable **`x`** is copied into a new stack entry for the variable **`n`**, so that the stack looks something like this:

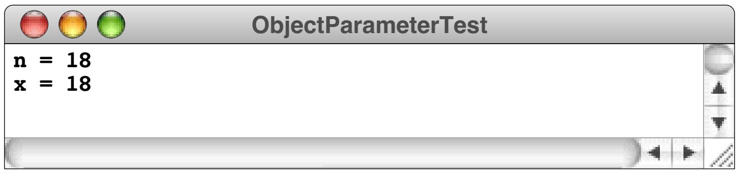


Changing the value of **`n`** inside the new frame has no effect on the original value.

But what would happen if these values were objects instead of simple integers? One way to answer that question is to design a simple class that contains an embedded integer value, as shown in Figure 7-1. The definition of the **`EmbeddedInteger`** class includes a constructor, a **`setValue`** method for setting the internal value, a **`getValue`** method for getting that value back, and a **`toString`** method to make it easier to display the object using **`println`**. Given these methods, you can write a similar test program that looks like this:

```
public void run() {
   EmbeddedInteger x = new EmbeddedInteger(17);
   increment(x);
   println("x = " + x);
}

private void increment(EmbeddedInteger n) {
   n.setValue(n.getValue() + 1);
   println("n = " + n);
}
```

If you run this version of the program, the output is different:

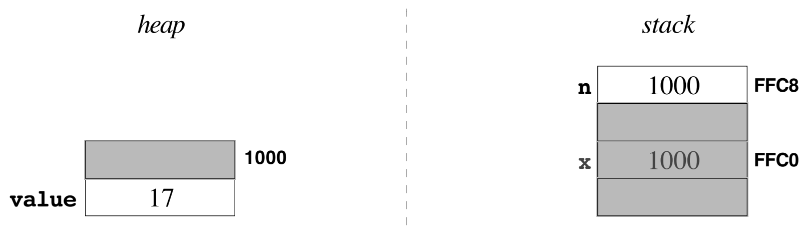


```
n = 18
x = 18
```

In the object-based implementation, changing the value of a field in the underlying object continues to be reflected after the method returns. This change in behavior follows directly from the memory structure. At the beginning of the call to **`increment`** in the new implementation, the heap and stack are in the following state:

FIGURE 7-1 **Implementation of the EmbeddedInteger class**

```
/* Class: EmbeddedInteger */
/**
 * This class allows its clients to treat an integer as an object.
 * The underlying integer value is set using setValue and returned
 * using getValue.
 */
public class EmbeddedInteger {

/* Constructor: EmbeddedInteger(n) */
/**
 * Creates an embedded integer with the value n.
 */
   public EmbeddedInteger(int n) {
      value = n;
   }

/* Method: setValue(n) */
/**
 * Sets the internal value of this EmbeddedInteger to n.
 */
   public void setValue(int n) {
      value = n;
   }

/* Method: getValue() */
/**
 * Returns the internal value of this EmbeddedInteger.
 */
   public int getValue() {
      return value;
   }

/* Method: toString() */
/**
 * Overrides the toString method to make it return the string
 * corresponding to the internal value.
 */
   public String toString() {
      return "" + value;
   }

/* Private instance variable */

   private int value;      /* The internal value */

}
```



In this case, changing the **`value`** field inside the object affects the conceptual value of both **`n`** and **`x`** because those two variables contain the same reference.

The intuitive effect of this difference in representation is that objects—in contrast to primitive values—are shared between the calling method and the method being called. The underlying mechanism, however, is exactly the same. Whenever a new local variable is initialized, the old value is copied into the stack location for the new variable. If that value is an object, what gets copied is the reference and not the underlying value.

Wrapper classes

The **`EmbeddedInteger`** class shown in Figure 7-1 is actually somewhat more than a simple illustration of parameter-passing in the object world. As you will discover in subsequent chapters, the Java library packages contain quite a few classes that are designed to work with any type of object. For example, the **`ArrayList`** class that you will learn about in Chapter 10 makes it possible to maintain an ordered list of objects. Given an **`ArrayList`**, you can add new values, remove existing ones, and perform a number of other useful operations that make sense in the abstract context of a list. The nature of the operations that manipulate the list turn out to be independent of the type of value the list contains. Thus, to make it possible to store any kind of object at all, the **`ArrayList`** class uses the universal class **`Object`** from which all other classes in the Java class hierarchy descend.

Unfortunately, being able to store any kind of object in an **`ArrayList`** does not provide quite as much flexibility as you might like. The Java primitive types are not objects and therefore cannot be used in conjunction with this marvelously convenient class. To get around this problem, Java defines a class, usually called a **wrapper class,** to go along with each of the eight primitive types, as follows:

Primitive type	Wrapper class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Each of the wrapper classes has a constructor that creates a new object from the corresponding primitive type. For example, if the variable **`n`** is an **`int`**, the declaration

```
    Integer nAsInteger = new Integer(n)
```

creates a new **`Integer`** object whose internal value is **`n`** and assigns it to **`nAsInteger`**. Each of the wrapper classes also defines an accessor method to retrieve the underlying value. The name of that method is always the name of the primitive type followed by the suffix **`Value`**. Thus, once you had initialized the variable **`nAsInteger`**, you could get back the value stored inside it by writing

```
nAsInteger.intValue()
```

Because **`nAsInteger`** is a legitimate object, you can store that value in an **`ArrayList`** or any other compound structure.

Unlike the **`EmbeddedInteger`** class from the preceding section, however, the wrapper classes provide no method to set the value of the underlying variable in an existing object. The wrapper classes are in fact immutable, as defined in Chapter 6. Immutable classes have the wonderful property that it is always possible to think about them as if they represent pure values in the way that Java's primitive types do. With an immutable class, you don't have to worry whether a value is shared or copied when you pass it from one method to another. Because neither side can change that value, it turns out not to matter.

The wrapper classes also include several static methods that you are likely to find useful, particularly those in the **`Integer`** class that support numeric conversion in arbitrary bases. Figure 7-2 lists a few of the most important static methods for the classes **`Integer`** and **`Double`**; the even more useful static methods in the **`Character`** class are described in Chapter 9. Because these methods are static, you need to include the name of the class. Thus, to convert the integer 50 into a hexadecimal string, you would need to write

```
Integer.toString(50, 16)
```

7.4 Linking objects together

The fact that objects are represented internally as references has one more implication that seems important to consider in the context of this chapter, even though a full discussion of the topic must wait until much later in the book. Because references are small, an object can easily contain references to other objects. If objects were stored only in the complete form in which they appear in the heap, it would obviously be impossible for a small object to contain a larger one. Moreover, given that every object requires some amount of overhead, it would even be impossible for one heap object to contain another object of the same size, because there would no longer be any room in which to store the bookkeeping information that Java needs. Those restrictions, however, disappear

FIGURE 7-2 **Numeric conversion methods in the Integer and Double classes**

Static methods in the **`Integer`** class

`static int parseInt(String str)` *or* **`parseInt(String str, int radix)`** Converts a string of digits into an **`int`** using the specified radix, which defaults to 10.
`static String toString(int i)` *or* **`toString(int i, int radix)`** Converts an **`int`** to its string representation for the specified radix, which defaults to 10.

Static methods in the **`Double`** class

`static double parseDouble(String str)` Converts the string representation of a number into the corresponding **`double`**.
`static String toString(double d)` Converts a **`double`** to its string representation.

completely if objects are stored as references. Although it is true that a small object cannot physically contain a larger object, there is nothing to prevent one object from containing a reference to another object no matter how much heap storage those objects might consume. A reference, after all, is simply the address of an object in memory and can therefore be represented using just a few bytes. Thus, there is no problem in having one object contain a reference to another object of a different class, or even an object of the same class.

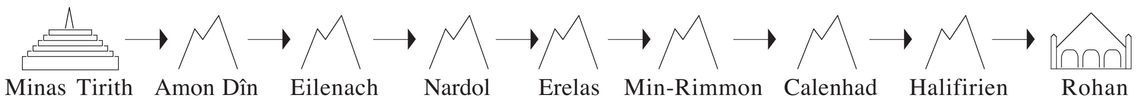
Creating an object that contains references to other objects is an extremely powerful programming technique. Such objects are said to be **linked.** Although an in-depth discussion of linked structures lies beyond the scope of this chapter, it is useful to present a simple example, both because that example will reinforce your understanding of how linked structures are represented in memory and because it offers a powerful illustration of the fundamental concept of **message-passing,** which is the metaphor Java uses to describe communication among the objects in a system. When an object needs to interact with some other object, it does so by invoking a method in the receiving object. In the context of the object-oriented paradigm, however, the act of calling that method is usually described as one of "sending a message" to the receiver. That message may in turn trigger a response in the receiver, such as a change in its state or the generation of additional messages that propagate that message to other objects. Even though the implementation of message-passing depends on method calls, it is important to keep the underlying metaphor in mind.

Message passing in linked structures: The beacons of Gondor

For answer Gandalf cried aloud to his horse. "On, Shadowfax! We must hasten. Time is short. See! The beacons of Gondor are alight, calling for aid. War is kindled. See, there is the fire on Amon Dîn, and flame on Eilenach; and there they go speeding west: Nardol, Erelas, Min-Rimmon, Calenhad, and the Halifirien on the borders of Rohan."

—J. R. R. Tolkien, *The Return of the King,* 1955

In adapting this scene for the concluding episode in his trilogy, Peter Jackson created what may be the most evocative and dramatic example of message-passing ever recorded on film. After the first beacon is lit in the towers of Minas Tirith, we see the signal pass from mountaintop to mountaintop as the keepers of each signal tower, ever vigilant, light their own fires as they see the triggering fire at the preceding station. The message of Gondor's danger thus passes quickly over the miles that separate it from Rohan, as illustrated by the following schematic diagram:



How would you go about simulating the lighting of the beacons of Gondor using the method-passing paradigm? Each signal tower is presumably a separate object that contains such information as the name of the tower. Each tower, however, must also be able to identify the next tower in the chain so that it knows where to send its message. The most straightforward strategy is to have each tower contain a reference to its successor. Thus, the object that represents Minas Tirith contains a reference to the object used to model Amon Dîn, which in turn contains a reference to the object that represents Eilenach, and so on. If you adopt this approach, the private data for each object must include—possibly along with other information required by the application as a whole—the following instance variables:

```
private String towerName;
private SignalTower nextTower;
```

The first variable contains the name of the tower, and the second is a reference to the next tower in sequence.

While this model makes sense for Minas Tirith and the intermediate towers in the chain, using this strategy to represent Rohan raises a minor issue by virtue of the fact that Rohan appears at the end of the chain. Like the other towers, Rohan has a **`nextTower`** field that is supposed to contain a reference to the next tower. Rohan doesn't have one, and the question is simply how to represent this fact. Fortunately, Java defines a special value called **`null`** for precisely this situation. The constant **`null`** represents a reference to a nonexistent value and can be assigned to any variable that holds an object reference. Thus, you can record the fact that Rohan is the last tower in the chain simply by making sure that its **`nextTower`** field is **`null`**.

Figure 7-3 contains a simple definition of a **`SignalTower`** class that implements the idea of passing a message along a chain of towers. The constructor takes the name of the tower and a reference to the next tower in sequence. The only other public method is **`signal`**, which is the action associated with the "Light your signal fire" message that propagates through the towers. When a tower receives the **`signal`** message, it lights its own signal fire and then passes the message along by sending the **`signal`** message to its successor, if any.

In the design of the **`SignalTower`** class shown in Figure 7-3, the process of lighting the current signal tower is the responsibility of the **`lightCurrentTower`** method. In this implementation, **`lightCurrentTower`** consists of the code

```
public void lightCurrentTower() {
   System.out.println("Lighting " + towerName);
}
```

which prints a message including the name of the tower on **`System.out`**, which is a standard output stream that is always available to Java programs. Unless your signal tower application is acting only as a test program, simply displaying the name of the tower is probably not what you want to do at this point. For example, if you were writing a graphical implementation of the "Beacons of Gondor" program—as you will have a chance to do in the exercises for Chapter 8—the implementation of this method would have to take whatever actions were necessary to display the signal fire on the screen. Given that the point here is to illustrate how including references as links inside each tower makes it possible to propagate the signal from one tower to the next, the additional detail required for a graphical implementation would only get in the way.

Even though the **`lightCurrentTower`** implementation shown has an effect that is useful only in test programs, it still makes sense, particularly in the object-oriented paradigm, to include it as a separate method in the class. A simple implementation that you intend to replace later is usually called a **stub.** If you got around to writing a more elaborate application, you could simply replace this stub with the appropriate code. But you could probably do better still to create a new subclass that overrides the definition of **`lightCurrentTower`**. The extended class can substitute its own version of this method but still rely on all the other features provided by the base class for linking the towers together.

FIGURE 7-3 **Implementation of the SignalTower class**

```
/* Class: SignalTower */
/**
 * This class defines a signal tower object that passes a message
 * to the next tower in a line.
 */

public class SignalTower {

/* Constructor: SignalTower(name, link) */
/**
 * Constructs a new signal tower with the following parameters:
 *
 * @param name The name of the tower
 * @param link A link to the next tower, or null if none exists
 */
   public SignalTower(String name, SignalTower link) {
      towerName = name;
      nextTower = link;
   }

/* Method: signal() */
/**
 * This method represents sending a signal to this tower. The effect
 * is to light the signal fire here and to send an additional signal
 * message to the next tower in the chain, if any.
 */
   public void signal() {
      lightCurrentTower();
      if (nextTower != null) nextTower.signal();
   }

/* Method: lightCurrentTower() */
/**
 * This method lights the signal fire for this tower. This version
 * supplies a temporary implementation (typically called a "stub")
 * that simply prints the name of the tower to the standard output
 * channel. If you wanted to redesign this class to be part of a
 * graphical application, for example, you could override this
 * method to draw an indication of the signal fire on the display.
 */
   public void lightCurrentTower() {
      System.out.println("Lighting " + towerName);
   }

/* Private instance variables */

   private String towerName;          /* The name of this tower    */
   private SignalTower nextTower;     /* A link to the next tower  */
}
```

The constructor for the `SignalTower` class is simple and does nothing more than copy its arguments into the corresponding instance variables. The code looks like this:

```
public SignalTower(String name, SignalTower link) {
   towerName = name;
   nextTower = link;
}
```

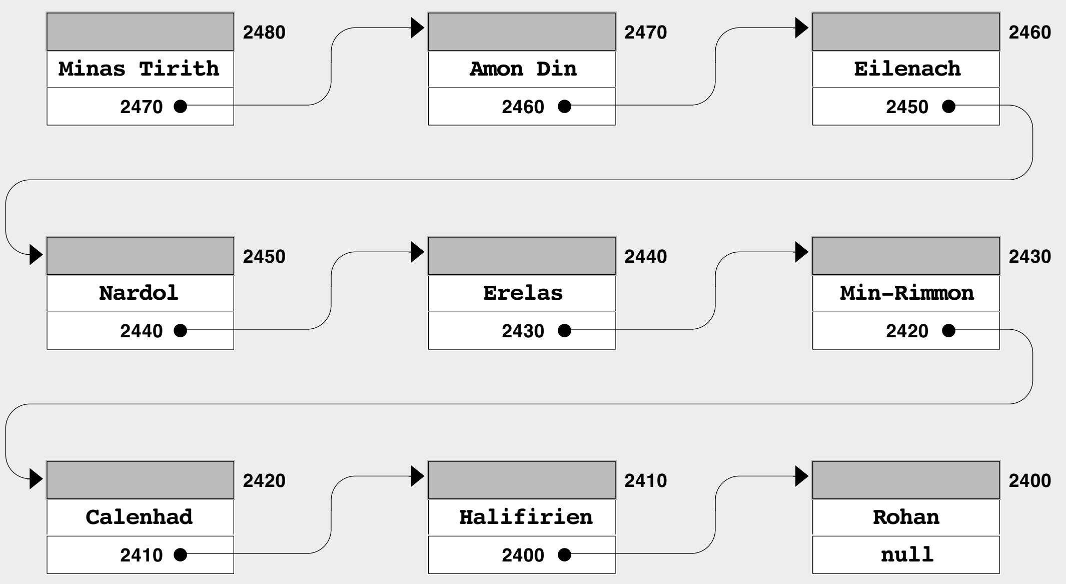
Because each invocation of the constructor for `SignalTower` requires you to specify the link to the next tower in the chain, it is easiest to create the data structure representing the chain of beacons in Gondor if you create it starting at the end of the chain with Rohan and working backwards toward the front of the chain at Minas Tirith. If each of the towers is declared as an instance variable, the following method initializes each of those variables to reflect the structure of the signal tower chain:

```
private void createSignalTowers() {
   rohan = new SignalTower("Rohan", null);
   halifirien = new SignalTower("Halifirien", rohan);
   calenhad = new SignalTower("Calenhad", halifirien);
   minRimmon = new SignalTower("Min-Rimmon", calenhad);
   erelas = new SignalTower("Erelas", minRimmon);
   nardol = new SignalTower("Nardol", erelas);
   eilenach = new SignalTower("Eilenach", nardol);
   amonDin = new SignalTower("Amon Din", eilenach);
   minasTirith = new SignalTower("Minas Tirith", amonDin);
}
```

The internal representation of linked structures

After you call `createSignalTowers` to initialize the towers, the internal representation for the objects in the heap would look something like the diagram in Figure 7-4. Note that the order in which the signal towers appear is indicated only by the chain of references in

FIGURE 7-4 Memory layout for the objects in the SignalTower chain



the final memory cells within each structure and not by the order in which the cells appear in memory. To emphasize the internal connections within the chain, the diagram in Figure 7-4 includes arrows that link each cell to its successor. The only value stored in memory is the numeric address, which is sufficient to find the next object in the chain.

Once you have established the data structure shown in Figure 7-4, all you need to do to get the process started is send a **`signal`** message to the first tower by invoking

```
    minasTirith.signal();
```

The implementation of the **`signal`** method takes whatever action is necessary to simulate lighting the beacon in Minas Tirith and then passes the **`signal`** message to the object representing Amon Dîn. That object in turn passes the **`signal`** message to its successor, and the process continues until it reaches the Rohan tower, at which point the **`null`** in the **`nextTower`** field indicates that the signal has reached the end of the line.

Summary

Even though the object model is designed to promote a more abstract view of data, using objects effectively requires you to have a mental model of how objects are represented in memory. In this chapter, you have had a chance to see how those objects are stored and to get a sense of what goes on "under the hood" as you create and use objects in your programs.

The important points introduced in this chapter include:

- The fundamental unit of information in a modern computer is a *bit,* which can be in one of two possible states. The state of a bit is usually represented in memory diagrams using the binary digits 0 and 1, but it is equally appropriate to think of these values as *off* and *on* or *false* and *true,* depending on the application.
- Sequences of bits are combined inside the hardware to form larger structures, including *bytes,* which are eight bits long, and *words,* which usually contain four bytes or 32 bits.
- The internal memory of a computer is arranged into a sequence of bytes in which each byte is identified by its index position in that sequence, which is called its *address.*
- Computer scientists tend to write address values and the contents of memory locations in *hexadecimal* notation (base 16) because doing so makes it easy to identify the individual bits.
- The primitive types in Java require different amounts of memory. A value of type **`char`** requires two bytes, a value of type **`int`** requires four, and a value of type **`double`** requires eight. The address of a multibyte value is the address of the first byte it contains.
- Data values that you create in a Java program are allocated in three different regions of memory. Static variables and constants that apply to an entire class are allocated in a region of memory devoted to the program code and static data. Local variables are allocated in a region called the *stack,* which is apportioned into structures called *frames* that contain all of the local variables for a method. All objects are allocated in a region called the *heap,* which is simply a pool of available memory. In most systems, the stack and the heap start at opposite ends of memory and grow toward each other to make available as much memory as possible.
- When you declare a local variable of some object class and assign it an initial value by calling its constructor, memory is allocated in both the heap and the stack. The heap

contains the storage for the object data itself. The stack entry for the variable contains only enough memory to hold the address of the object, which is called a *reference*.

- The stack frame for a method call includes an entry identified by the keyword **`this`** that identifies the *receiver,* which is the object on which that method is applied.
- As a computation proceeds, objects on the heap are often used as temporary values that are no longer needed at the end of the computation. Such values, however, continue to occupy memory in the heap and are referred to as *garbage*. The Java runtime system periodically undertakes a process called *garbage collection* to search through the heap and reclaim objects that are no longer needed.
- When an object is passed from one method to another, only the reference to that object is copied into the stack frame for the new method. Because this reference identifies exactly the same object as the identical reference in the caller, object values are effectively shared between the calling and called methods.
- The primitive types in Java are not in fact objects and therefore cannot be used in contexts in which an object is required. To get around this problem, Java defines a set of *wrapper classes* that encapsulate each of the primitive types in a full-fledged object.
- Objects in Java can contain references to other objects. Such objects are said to be *linked*. Linked structures are used quite often in programming and will be covered in more detail in later chapters in this book.

Review questions

1. Define the following terms: *bit, byte,* and *word*.

2. What is the etymology of the word *bit?*

3. How many bytes of memory are there in a 384MB machine?

4. Convert each of the following decimal numbers to its hexadecimal equivalent:

   a) 17
   b) 256
   c) 1729
   d) 2766

5. Convert each of the following hexadecimal numbers to decimal:

   a) **17**
   b) **64**
   c) **CC**
   d) **FAD**

6. What is an *address?*

7. How many bytes does Java assign to a value of type **`int`**? How many bytes are required for a **`double`**?

8. What are the three memory regions in which values can be stored in a Java program?

9. Using the example in section 7.2 as a model, trace the heap and stack operations that occur in the execution of the following method:

```
public void run() {
   Rational x = new Rational(4, 5);
   Rational y = new Rational(5, 2);
   Rational z = x.multiply(y).subtract(z);
   println(x + " x " + y + " - " + y + " = " + z);
}
```

Which objects on the heap are garbage when the **`println`** statement is reached.

10. True or false: When you pass a primitive data value from one method to another, Java always copies that value into the frame of the method being called.

11. True or false: When you pass an object from one method to another, Java copies the data in that object into the new frame.

12. Describe the two phases in a simple mark-and-sweep garbage collector.

13. What is meant by the term *wrapper class?* What purpose do wrapper classes serve?

14. What methods can you use to convert between integers and the string representations of those integers in a particular base?

15. What property identifies a *linked structure?*

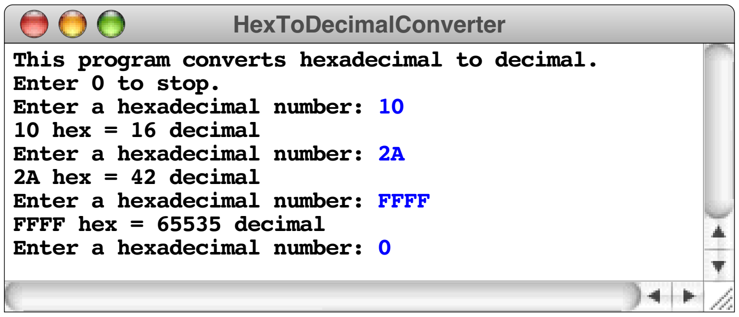
16. Given that objects of a particular class require a certain amount of space in memory, it is clear that an object in the heap could never physically contain another object of that same class and still have room for additional data. What can a Java programmer do to achieve the same effect?

17. What is a *stub?*

18. Why is important for Java to include the special value **`null?`** What does this value represent?

Programming exercises

1. Use the static methods **`Integer.parseInt`** and **`Integer.toString`** to write a program that converts hexadecimal values into their decimal equivalents. Your program should continue to read hexadecimal values until the user enters a 0. A sample run of this program might look like this:



```
This program converts hexadecimal to decimal.
Enter 0 to stop.
Enter a hexadecimal number: 10
10 hex = 16 decimal
Enter a hexadecimal number: 2A
2A hex = 42 decimal
Enter a hexadecimal number: FFFF
FFFF hex = 65535 decimal
Enter a hexadecimal number: 0
```

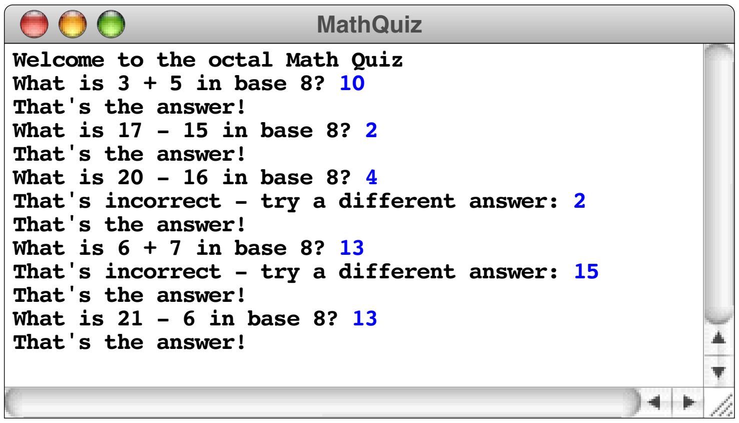
2. The fact that the **`Integer.parseInt`** method makes it possible for a program to read user input as a string and then convert it to an integer makes it possible to write programs that use something other than an integer—such as a blank line—as a

sentinel to signal the end of the input. Rewrite the **`AverageList`** program from exercise 4-6 so that it uses a blank line to mark the end of the input.

3. But don't panic. Base 8 is just like base 10 really—if you're missing two fingers.

   —Tom Lehrer, "The New Math," 1965

   Rewrite the Math Quiz program from exercise 6-5 so that it poses its questions in base 8 instead of base 10, as shown in the following sample run:



```
Welcome to the octal Math Quiz
What is 3 + 5 in base 8? 10
That's the answer!
What is 17 - 15 in base 8? 2
That's the answer!
What is 20 - 16 in base 8? 4
That's incorrect - try a different answer: 2
That's the answer!
What is 6 + 7 in base 8? 13
That's incorrect - try a different answer: 15
That's the answer!
What is 21 - 6 in base 8? 13
That's the answer!
```

4. The **`Runtime`** class in the **`java.lang`** package includes a few simple methods that may help you get a better sense of what Java's garbage collector does. A **`Runtime`** object maintains information about the state of the Java Virtual Machine. If you want to look at that information, you can get the current runtime environment by calling the static method **`getRuntime()`** and storing the result in a variable like this:

```
Runtime myRuntime = Runtime.getRuntime();
```

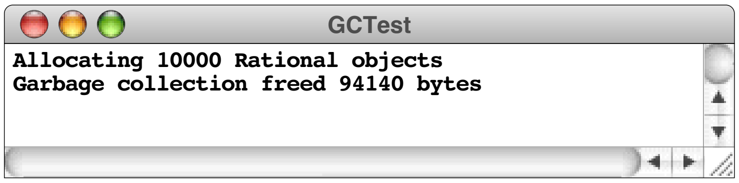
   Once you have this variable, you can find out how much free memory is available by calling

```
myRuntime.freeMemory();
```

   Because memory sizes can be large, the value returned by **`freeMemory`** is a **`long`** rather than an **`int`** and indicates the number of bytes available. You can also explicitly trigger the garbage collector by calling

```
myRuntime.gc();
```

   Write a program that allocates 10000 **`Rational`** objects without saving any of them in variables so that they all become garbage. Once you've done so, measure the amount of free memory before and after garbage collection and use the difference to report how many bytes were freed, as shown in the following sample run:



```
Allocating 10000 Rational objects
Garbage collection freed 94140 bytes
```