Type in this code and try out a few simple cases like creating a BigInteger and printing it. Notice that `println` knows how to print BigIntegers! Don't forget to add `import java.math.BigInteger` to the beginning of your program.

e. Unfortunately, because BigIntegers are not primitive types, we cannot use the usual math operators on them. Instead we have to use object methods like `add`. In order to add two BigIntegers, you have to invoke `add` on one of the objects and pass the other as an argument. For example:

```
BigInteger small = BigInteger.valueOf (17);
BigInteger big = BigInteger.valueOf (1700000000);
BigInteger total = small.add (big);
```

Try out some of the other methods, like `multiply` and `pow`.

f. Convert `factorial` so that it performs its calculation using BigIntegers, and then returns the BigInteger as a result. You can leave the parameter alone—it will still be an integer.

g. Try printing the table again with your modified factorial function. Is it correct up to 30? How high can you make it go? I calculated the factorial of all the numbers from 0 to 999, but my machine is pretty slow, so it took a while. The last number, 999!, has 2565 digits.

Exercise 8.5 Many encryption algorithms depends on the ability to raise large integers to an integer power. Here is a method that implements a (reasonably) fast algorithm for integer exponentiation:

```
public static int pow (int x, int n) {
    if (n==0) return 1;

    // find x to the n/2 recursively
    int t = pow (x, n/2);

    // if n is even, the result is t squared
    // if n is odd, the result is t squared times x

    if (n%2 == 0) {
      return t*t;
    } else {
      return t*t*x;
    }
}
```

The problem with this method is that it will only work if the result is smaller than 2 billion. Rewrite it so that the result is a `BigInteger`. The parameters should still be integers, though.

You can use the BigInteger methods `add` and `multiply`, but don't use the built-in `pow` method, which would spoil the fun.

Chapter 9

Create your own objects

9.1 Class definitions and object types

Every time you write a class definition, you create a new object type, with the same name as the class. Way back in Section 1.5, when we defined the class named `Hello`, we also created an object type named `Hello`. We didn't create any variables with type `Hello`, and we didn't use the `new` command to create any `Hello` objects, but we could have!

That example doesn't make much sense, since there is no reason to create a `Hello` object, and it is not clear what it would be good for if we did. In this chapter, we will look at some examples of class definitions that create *useful* new object types.

Here are the most important ideas in this chapter:

- Defining a new class also creates a new object type with the same name.
- A class definition is like a template for objects: it determines what instance variables the objects have and what methods can operate on them.
- Every object belongs to some object type; hence, it is an instance of some class.
- When you invoke the `new` command to create an object, Java invokes a special method called a **constructor** to initialize the instance variables. You provide one or more constructors as part of the class definition.
- Typically all the methods that operate on a type go in the class definition for that type.

Here are some syntax issues about class definitions:

- Class names (and hence object types) always begin with a capital letter, which helps distinguish them from primitive types and variable names.

- You usually put one class definition in each file, and the name of the file must be the same as the name of the class, with the suffix `.java`. For example, the `Time` class is defined in the file named `Time.java`.

- In any program, one class is designated as the **startup class**. The startup class must contain a method named `main`, which is where the execution of the program begins. Other classes *may* have a method named `main`, but it will not be executed.

With those issues out of the way, let's look at an example of a user-defined type, `Time`.

9.2 Time

A common motivation for creating a new Object type is to take several related pieces of data and encapsulate them into an object that can be manipulated (passed as an argument, operated on) as a single unit. We have already seen two built-in types like this, `Point` and `Rectangle`.

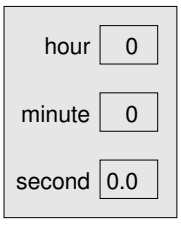
Another example, which we will implement ourselves, is `Time`, which is used to record the time of day. The various pieces of information that form a time are the hour, minute and second. Because every `Time` object will contain these data, we need to create instance variables to hold them.

The first step is to decide what type each variable should be. It seems clear that `hour` and `minute` should be integers. Just to keep things interesting, let's make `second` a `double`, so we can record fractions of a second.

Instance variables are declared at the beginning of the class definition, outside of any method definition, like this:

```
class Time {
  int hour, minute;
  double second;
}
```

All by itself, this code fragment is a legal class definition. The state diagram for a `Time` object would look like this:



After declaring the instance variables, the next step is usually to define a constructor for the new class.

9.3 Constructors

The usual role of a constructor is to initialize the instance variables. The syntax for constructors is similar to that of other methods, with three exceptions:

- The name of the constructor is the same as the name of the class.
- Constructors have no return type and no return value.
- The keyword `static` is omitted.

Here is an example for the `Time` class:

```
public Time () {
  this.hour = 0;
  this.minute = 0;
  this.second = 0.0;
}
```

Notice that where you would expect to see a return type, between `public` and `Time`, there is nothing. That's how we (and the compiler) can tell that this is a constructor.

This constructor does not take any arguments, as indicated by the empty parentheses `()`. Each line of the constructor initializes an instance variable to an arbitrary default value (in this case, midnight). The name `this` is a special keyword that is the name of the object we are creating. You can use `this` the same way you use the name of any other object. For example, you can read and write the instance variables of `this`, and you can pass `this` as an argument to other methods.

But you do not declare `this` and you do not use `new` to create it. In fact, you are not even allowed to make an assignment to it! `this` is created by the system; all you have to do is store values in its instance variables.

A common error when writing constructors is to put a `return` statement at the end. Resist the temptation.

9.4 More constructors

Constructors can be overloaded, just like other methods, which means that you can provide multiple constructors with different parameters. Java knows which constructor to invoke by matching the arguments of the `new` command with the parameters of the constructors.

It is very common to have one constructor that takes no arguments (shown above), and one constructor that takes a parameter list that is identical to the list of instance variables. For example:

```
  public Time (int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }
```

The names and types of the parameters are exactly the same as the names and types of the instance variables. All the constructor does is copy the information from the parameters to the instance variables.

If you go back and look at the documentation for `Point`s and `Rectangle`s, you will see that both classes provide constructors like this. Overloading constructors provides the flexibility to create an object first and then fill in the blanks, or to collect all the information before creating the object.

So far this might not seem very interesting, and in fact it is not. Writing constructors is a boring, mechanical process. Once you have written two, you will find that you can churn them out in your sleep, just by looking at the list of instance variables.

9.5 Creating a new object

Although constructors look like methods, you never invoke them directly. Instead, when you use the `new` command, the system allocates space for the new object and then invokes your constructor to initialize the instance variables.

The following program demonstrates two ways to create and initialize `Time` objects:

```
class Time {
  int hour, minute;
  double second;

  public Time () {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
  }

  public Time (int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }

  public static void main (String[] args) {

    // one way to create and initialize a Time object
    Time t1 = new Time ();
```

```
    t1.hour = 11;
    t1.minute = 8;
    t1.second = 3.14159;
    System.out.println (t1);

    // another way to do the same thing
    Time t2 = new Time (11, 8, 3.14159);
    System.out.println (t2);
  }
}
```

As an exercise, figure out the flow of execution through this program.

In `main`, the first time we invoke the `new` command, we provide no arguments, so Java invokes the first constructor. The next few lines assign values to each of the instance variables.

The second time we invoke the `new` command, we provide arguments that match the parameters of the second constructor. This way of initializing the instance variables is more concise (and slightly more efficient), but it can be harder to read, since it is not as clear which values are assigned to which instance variables.

9.6 Printing an object

The output of this program is:

```
Time@80cc7c0
Time@80cc807
```

When Java prints the value of a user-defined object type, it prints the name of the type and a special hexadecimal (base 16) code that is unique for each object. This code is not meaningful in itself; in fact, it can vary from machine to machine and even from run to run. But it can be useful for debugging, in case you want to keep track of individual objects.

In order to print objects in a way that is more meaningful to users (as opposed to programmers), you usually want to write a method called something like `printTime`:

```
public static void printTime (Time t) {
  System.out.println (t.hour + ":" + t.minute + ":" + t.second);
}
```

Compare this method to the version of `printTime` in Section 3.10.

The output of this method, if we pass either `t1` or `t2` as an argument, is `11:8:3.14159`. Although this is recognizable as a time, it is not quite in the standard format. For example, if the number of minutes or seconds is less than 10, we expect a leading `0` as a place-keeper. Also, we might want to drop the decimal part of the seconds. In other words, we want something like `11:08:03`.

In most languages, there are simple ways to control the output format for numbers. In Java there are no simple ways.

Java provides very powerful tools for printing formatted things like times and dates, and also for interpreting formatted input. Unfortunately, these tools are not very easy to use, so I am going to leave them out of this book. If you want, though, you can take a look at the documentation for the `Date` class in the `java.util` package.

9.7 Operations on objects

Even though we can't print times in an optimal format, we can still write methods that manipulate `Time` objects. In the next few sections, I will demonstrate several of the possible interfaces for methods that operate on objects. For some operations, you will have a choice of several possible interfaces, so you should consider the pros and cons of each of these:

pure function: Takes objects and/or primitives as arguments but does not modify the objects. The return value is either a primitive or a new object created inside the method.

modifier: Takes objects as arguments and modifies some or all of them. Often returns void.

fill-in method: One of the arguments is an "empty" object that gets filled in by the method. Technically, this is a type of modifier.

9.8 Pure functions

A method is considered a pure function if the result depends only on the arguments, and it has no side effects like modifying an argument or printing something. The only result of invoking a pure function is the return value.

One example is `after`, which compares two `Time`s and returns a `boolean` that indicates whether the first operand comes after the second:

```
public static boolean after (Time time1, Time time2) {
  if (time1.hour > time2.hour) return true;
  if (time1.hour < time2.hour) return false;

  if (time1.minute > time2.minute) return true;
  if (time1.minute < time2.minute) return false;

  if (time1.second > time2.second) return true;
  return false;
}
```

What is the result of this method if the two times are equal? Does that seem like the appropriate result for this method? If you were writing the documentation for this method, would you mention that case specifically?

A second example is `addTime`, which calculates the sum of two times. For example, if it is `9:14:30`, and your breadmaker takes 3 hours and 35 minutes, you could use `addTime` to figure out when the bread will be done.

Here is a rough draft of this method that is not quite right:

```
public static Time addTime (Time t1, Time t2) {
  Time sum = new Time ();
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;
  return sum;
}
```

Although this method returns a `Time` object, it is not a constructor. You should go back and compare the syntax of a method like this with the syntax of a constructor, because it is easy to get confused.

Here is an example of how to use this method. If `currentTime` contains the current time and `breadTime` contains the amount of time it takes for your breadmaker to make bread, then you could use `addTime` to figure out when the bread will be done.

```
Time currentTime = new Time (9, 14, 30.0);
Time breadTime = new Time (3, 35, 0.0);
Time doneTime = addTime (currentTime, breadTime);
printTime (doneTime);
```

The output of this program is `12:49:30.0`, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

The problem is that this method does not deal with cases where the number of seconds or minutes adds up to more than 60. In that case, we have to "carry" the extra seconds into the minutes column, or extra minutes into the hours column.

Here's a second, corrected version of this method.

```
public static Time addTime (Time t1, Time t2) {
  Time sum = new Time ();
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;

  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
  }
  return sum;
```

```
}
```

Although it's correct, it's starting to get big. Later, I will suggest an alternative approach to this problem that will be much shorter.

This code demonstrates two operators we have not seen before, `+=` and `-=`. These operators provide a concise way to increment and decrement variables. They are similar to `++` and `--`, except (1) they work on `double`s as well as `int`s, and (2) the amount of the increment does not have to be 1. The statement `sum.second -= 60.0;` is equivalent to `sum.second = sum.second - 60;`

9.9 Modifiers

As an example of a modifier, consider `increment`, which adds a given number of seconds to a `Time` object. Again, a rough draft of this method looks like:

```
public static void increment (Time time, double secs) {
  time.second += secs;

  if (time.second >= 60.0) {
    time.second -= 60.0;
    time.minute += 1;
  }
  if (time.minute >= 60) {
    time.minute -= 60;
    time.hour += 1;
  }
}
```

The first line performs the basic operation; the remainder deals with the same cases we saw before.

Is this method correct? What happens if the argument `secs` is much greater than 60? In that case, it is not enough to subtract 60 once; we have to keep doing it until `second` is below 60. We can do that by simply replacing the `if` statements with `while` statements:

```
public static void increment (Time time, double secs) {
  time.second += secs;

  while (time.second >= 60.0) {
    time.second -= 60.0;
    time.minute += 1;
  }
  while (time.minute >= 60) {
    time.minute -= 60;
    time.hour += 1;
  }
}
```

This solution is correct, but not very efficient. Can you think of a solution that does not require iteration?

9.10 Fill-in methods

Occasionally you will see methods like `addTime` written with a different interface (different arguments and return values). Instead of creating a new object every time `addTime` is invoked, we could require the caller to provide an "empty" object where `addTime` should store the result. Compare the following with the previous version:

```
public static void addTimeFill (Time t1, Time t2, Time sum) {
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;

  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
  }
}
```

One advantage of this approach is that the caller has the option of reusing the same object repeatedly to perform a series of additions. This can be slightly more efficient, although it can be confusing enough to cause subtle errors. For the vast majority of programming, it is worth spending a little run time to avoid a lot of debugging time.

9.11 Which is best?

Anything that can be done with modifiers and fill-in methods can also be done with pure functions. In fact, there are programming languages, called **functional** programming languages, that only allow pure functions. Some programmers believe that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, there are times when modifiers are convenient, and some cases where functional programs are less efficient.

In general, I recommend that you write pure functions whenever it is reasonable to do so, and resort to modifiers only if there is a compelling advantage. This approach might be called a functional programming style.

9.12 Incremental development vs. planning

In this chapter I have demonstrated an approach to program development I refer to as **rapid prototyping with iterative improvement**. In each case,

I wrote a rough draft (or prototype) that performed the basic calculation, and then tested it on a few cases, correcting flaws as I found them.

Although this approach can be effective, it can lead to code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to convince yourself that you have found *all* the errors.

An alternative is high-level planning, in which a little insight into the problem can make the programming much easier. In this case the insight is that a `Time` is really a three-digit number in base 60! The `second` is the "ones column," the `minute` is the "60's column", and the `hour` is the "3600's column."

When we wrote `addTime` and `increment`, we were effectively doing addition in base 60, which is why we had to "carry" from one column to the next.

Thus an alternative approach to the whole problem is to convert `Time`s into `double`s and take advantage of the fact that the computer already knows how to do arithmetic with `double`s. Here is a method that converts a `Time` into a `double`:

```
public static double convertToSeconds (Time t) {
  int minutes = t.hour * 60 + t.minute;
  double seconds = minutes * 60 + t.second;
  return seconds;
}
```

Now all we need is a way to convert from a `double` to a `Time` object. We could write a method to do it, but it might make more sense to write it as a third constructor:

```
public Time (double secs) {
  this.hour = (int) (secs / 3600.0);
  secs -= this.hour * 3600.0;
  this.minute = (int) (secs / 60.0);
  secs -= this.minute * 60;
  this.second = secs;
}
```

This constructor is a little different from the others, since it involves some calculation along with assignments to the instance variables.

You might have to think a bit to convince yourself that the technique I am using to convert from one base to another is correct. Assuming you are convinced, we can use these methods to rewrite `addTime`:

```
public static Time addTime (Time t1, Time t2) {
  double seconds = convertToSeconds (t1) + convertToSeconds (t2);
  return new Time (seconds);
}
```

This is much shorter than the original version, and it is much easier to demonstrate that it is correct (assuming, as usual, that the methods it invokes are correct). As an exercise, rewrite `increment` the same way.

9.13 Generalization

In some ways converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers, and make the investment of writing the conversion methods (`convertToSeconds` and the third constructor), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add more features later. For example, imagine subtracting two `Time`s to find the duration between them. The naive approach would be to implement subtraction complete with "borrowing." Using the conversion methods would be much easier.

Ironically, sometimes making a problem harder (more general) makes it easier (fewer special cases, fewer opportunities for error).

9.14 Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an **algorithm**. I mentioned this word in Chapter 1, but did not define it carefully. It is not easy to define, so I will try a couple of approaches.

First, consider some things that are not algorithms. For example, when you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions, so that knowledge is not really algorithmic.

But if you were "lazy," you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the most difficult to express algorithmically. Understanding natural

language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

Later you will have the opportunity to design simple algorithms for a variety of problems.

9.15 Glossary

class: Previously, I defined a class as a collection of related methods. In this chapter we learned that a class definition is also a template for a new type of object.

instance: A member of a class. Every object is an instance of some class.

constructor: A special method that initializes the instance variables of a newly-constructed object.

project: A collection of one or more class definitions (one per file) that make up a program.

startup class: The class that contains the `main` method where execution of the program begins.

function: A method whose result depends only on its parameters, and that has no side-effects other than returning a value.

functional programming style: A style of program design in which the majority of methods are functions.

modifier: A method that changes one or more of the objects it receives as parameters, and usually returns `void`.

fill-in method: A type of method that takes an "empty" object as a parameter and fills in its instance variables instead of generating a return value. This type of method is usually not the best choice.

algorithm: A set of instructions for solving a class of problems by a mechanical process.

9.16 Exercises

Exercise 9.1 In the board game Scrabble[1], each tile contains a letter, which is used to spell words, and a score, which is used to determine the value of a word.

a. Write a definition for a class named `Tile` that represents Scrabble tiles. The instance variables should be a character named `letter` and an integer named `value`.

[1] Scrabble is a registered trademark owned in the U.S.A and Canada by Hasbro Inc., and in the rest of the world by J.W. Spear & Sons Limited of Maidenhead, Berkshire, England, a subsidiary of Mattel Inc.

b. Write a constructor that takes parameters named `letter` and `value` and initializes the instance variables.

c. Write a method named `printTile` that takes a `Tile` object as a parameter and prints the instance variables in some reader-friendly format.

d. Write a method named `testTile` that creates a Tile object with the letter `Z` and the value 10, and then uses `printTile` to print the state of the object.

The point of this exercise is to practice the mechanical part of creating a new class definition and code that tests it.

Exercise 9.2 Write a class definition for `Date`, an object type that contains three integers, `year`, `month` and `day`. This class should provide two constructors. The first should take no parameters. The second should take parameters named `year`, `month` and `day`, and use them to initialize the instance variables.

Add code to `main` that creates a new `Date` object named `birthday`. The new object should contain your birthdate. You can use either constructor.

Exercise 9.3

A rational number is a number that can be represented as the ratio of two integers. For example, 2/3 is a rational number, and you can think of 7 as a rational number with an implicit 1 in the denominator. For this assignment, you are going to write a class definition for rational numbers.

a. Examine the following program and make sure you understand what it does:

```
public class Complex
{
    double real, imag;

    // simple constructor
    public Complex () {
        this.real = 0.0;  this.imag = 0.0;
    }

    // constructor that takes arguments
    public Complex (double real, double imag) {
        this.real = real;  this.imag = imag;
    }

    public static void printComplex (Complex c) {
        System.out.println (c.real + " + " + c.imag + "i");
    }

    // conjugate is a modifier
    public static void conjugate (Complex c) {
        c.imag = -c.imag;
    }

    // abs is a function that returns a primitive
    public static double abs (Complex c) {
        return Math.sqrt (c.real * c.real + c.imag * c.imag);
```

```
    }

    // add is a function that returns a new Complex object
    public static Complex add (Complex a, Complex b) {
        return new Complex (a.real + b.real, a.imag + b.imag);
    }

    public static void main(String args[]) {

        // use the first constructor
        Complex x = new Complex ();
        x.real = 1.0;
        x.imag = 2.0;

        // use the second constructor
        Complex y = new Complex (3.0, 4.0);

        System.out.println (Complex.abs (y));

        Complex.conjugate (x);
        Complex.printComplex (x);
        Complex.printComplex (y);

        Complex s = Complex.add (x, y);
        Complex.printComplex (s);
    }
}
```

b. Create a new program called `Rational.java` that defines a class named `Rational`. A `Rational` object should have two integer instance variables to store the numerator and denominator of a rational number.

c. Write a constructor that takes no arguments and that sets the two instance variables to zero.

d. Write a method called `printRational` that takes a Rational object as an argument and prints it in some reasonable format.

e. Write a `main` method that creates a new object with type Rational, sets its instance variables to some values, and prints the object.

f. At this stage, you have a minimal testable (debuggable) program. Test it and, if necessary, debug it.

g. Write a second constructor for your class that takes two arguments and that uses them to initalize the instance variables.

h. Write a method called `negate` that reverses the sign of a rational number. This method should be a modifier, so it should return `void`. Add lines to `main` to test the new method.

i. Write a method called `invert` that inverts the number by swapping the numerator and denominator. Remember the swap pattern we have seen before. Add lines to `main` to test the new method.

j. Write a method called `toDouble` that converts the rational number to a double (floating-point number) and returns the result. This method is a pure function; it does not modify the object. As always, test the new method.

k. Write a modifier named `reduce` that reduces a rational number to its lowest terms by finding the GCD of the numerator and denominator and then dividing top and bottom by the GCD. This method should be a pure function; it should not modify the instance variables of the object on which it is invoked.

   You may want to write a method called `gcd` that finds the greatest common divisor of the numerator and the denominator (See Exercise 5.10).

l. Write a method called `add` that takes two Rational numbers as arguments and returns a new Rational object. The return object, not surprisingly, should contain the sum of the arguments.

   There are several ways to add fractions. You can use any one you want, but you should make sure that the result of the operation is reduced so that the numerator and denominator have no common divisor (other than 1).

The purpose of this exercise is to write a class definition that includes a variety of methods, including constructors, modifiers and pure functions.

Chapter 10

Arrays

An **array** is a set of values where each value is identified by an index. You can make an array of `int`s, `double`s, or any other type, but all the values in an array have to have the same type.

Syntactically, array types look like other Java types except they are followed by `[]`. For example, `int[]` is the type "array of integers" and `double[]` is the type "array of doubles."

You can declare variables with these types in the usual ways:

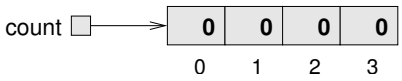
```
int[] count;
double[] values;
```

Until you initialize these variables, they are set to `null`. To create the array itself, use the `new` command.

```
count = new int[4];
values = new double[size];
```

The first assignment makes `count` refer to an array of 4 integers; the second makes `values` refer to an array of `double`s. The number of elements in `values` depends on `size`. You can use any integer expression as an array size.

The following figure shows how arrays are represented in state diagrams:



The large numbers inside the boxes are the **elements** of the array. The small numbers outside the boxes are the indices used to identify each box. When you allocate a new array, the elements are initialized to zero.

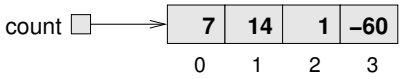
10.1 Accessing elements

To store values in the array, use the `[]` operator. For example `count[0]` refers to the "zeroeth" element of the array, and `count[1]` refers to the "oneth" element.

You can use the `[]` operator anywhere in an expression:

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

All of these are legal assignment statements. Here is the effect of this code fragment:



By now you should have noticed that the four elements of this array are numbered from 0 to 3, which means that there is no element with the index 4. This should sound familiar, since we saw the same thing with `String` indices. Nevertheless, it is a common error to go beyond the bounds of an array, which will cause an `ArrayOutOfBoundsException`. As with all exceptions, you get an error message and the program quits.

You can use any expression as an index, as long as it has type `int`. One of the most common ways to index an array is with a loop variable. For example:

```
int i = 0;
while (i < 4) {
  System.out.println (count[i]);
  i++;
}
```

This is a standard `while` loop that counts from 0 up to 4, and when the loop variable `i` is 4, the condition fails and the loop terminates. Thus, the body of the loop is only executed when `i` is 0, 1, 2 and 3.

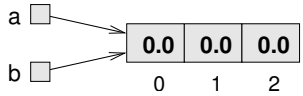
Each time through the loop we use `i` as an index into the array, printing the `i`th element. This type of array traversal is very common. Arrays and loops go together like fava beans and a nice Chianti.

10.2 Copying arrays

When you copy an array variable, remember that you are copying a reference to the array. For example:

```
double[] a = new double [3];
double[] b = a;
```

This code creates one array of three `double`s, and sets two different variables to refer to it. This situation is a form of aliasing.



Any changes in either array will be reflected in the other. This is not usually the behavior you want; instead, you should make a copy of the array, by allocating a new array and copying each element from one to the other.

```
double[] b = new double [3];

int i = 0;
while (i < 4) {
  b[i] = a[i];
  i++;
}
```

10.3 for loops

The loops we have written so far have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they do something to that variable, like increment it.

This type of loop is so common that there is an alternative loop statement, called `for`, that expresses it more concisely. The general syntax looks like this:

```
for (INITIALIZER; CONDITION; INCREMENTOR) {
  BODY
}
```

This statement is exactly equivalent to

```
INITIALIZER;
while (CONDITION) {
  BODY
  INCREMENTOR
}
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. For example:

```
for (int i = 0; i < 4; i++) {
  System.out.println (count[i]);
}
```

is equivalent to

```
int i = 0;
while (i < 4) {
  System.out.println (count[i]);
  i++;
}
```

As an exercise, write a `for` loop to copy the elements of an array.

10.4 Arrays and objects

In many ways, arrays behave like objects:

- When you declare an array variable, you get a reference to an array.
- You have to use the `new` command to create the array itself.
- When you pass an array as an argument, you pass a reference, which means that the invoked method can change the contents of the array.

Some of the objects we have looked at, like `Rectangles`, are similar to arrays, in the sense that they are named collection of values. This raises the question, "How is an array of 4 integers different from a Rectangle object?"

If you go back to the definition of "array" at the beginning of the chapter, you will see one difference, which is that the elements of an array are identified by indices, whereas the elements (instance variables) of an object have names (like `x`, `width`, etc.).

Another difference between arrays and objects is that all the elements of an array have to be the same type. Although that is also true of `Rectangle`s, we have seen other objects that have instance variables with different types (like `Time`).

10.5 Array length

Actually, arrays do have one named instance variable: `length`. Not surprisingly, it contains the length of the array (number of elements). It is a good idea to use this value as the upper bound of a loop, rather than a constant value. That way, if the size of the array changes, you won't have to go through the program changing all the loops; they will work correctly for any size array.

```
for (int i = 0; i < a.length; i++) {
  b[i] = a[i];
}
```

The last time the body of the loop gets executed, `i` is `a.length - 1`, which is the index of the last element. When `i` is equal to `a.length`, the condition fails and the body is not executed, which is a good thing, since it would cause an exception. This code assumes that the array `b` contains at least as many elements as `a`.

As an exercise, write a method called `cloneArray` that takes an array of integers as a parameter, creates a new array that is the same size, copies the elements from the first array into the new one, and then returns a reference to the new array.

10.6 Random numbers

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. For some applications, though, we would like the computer to be unpredictable. Games are an obvious example, but there are many more.

Making a program truly **nondeterministic** turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate random numbers and use them to determine the outcome of the program. Java provides a built-in method that generates **pseudorandom** numbers, which are not truly random in the mathematical sense, but for our purposes, they will do.

Check out the documentation of the `random` method in the `Math` class. The return value is a `double` between 0.0 and 1.0. To be precise, it is greater than or equal to 0.0 and strictly less than 1.0. Each time you invoke `random` you get the next number in a pseudorandom sequence. To see a sample, run this loop:

```
for (int i = 0; i < 10; i++) {
  double x = Math.random ();
  System.out.println (x);
}
```

To generate a random `double` between 0.0 and an upper bound like `high`, you can multiply `x` by `high`. How would you generate a random number between `low` and `high`? How would you generate a random integer?

Exercise 10.1 Write a method called `randomDouble` that takes two doubles, `low` and `high`, and that returns a random double x so that $low \leq x < high$.

Exercise 10.2 Write a method called `randomInt` that takes two arguments, `low` and `high`, and that returns a random integer between `low` and `high` (including both).

10.7 Array of random numbers

If your implementation of `randomInt` is correct, then every value in the range from `low` to `high` should have the same probability. If you generate a long series of numbers, every value should appear, at least approximately, the same number of times.

One way to test your method is to generate a large number of random values, store them in an array, and count the number of times each value occurs.

The following method takes a single argument, the size of the array. It allocates a new array of integers, fills it with random values, and returns a reference to the new array.

```
public static int[] randomArray (int n) {
  int[] a = new int[n];
```

```
    for (int i = 0; i<a.length; i++) {
      a[i] = randomInt (0, 100);
    }
    return a;
  }
```

The return type is `int[]`, which means that this method returns an array of integers. To test this method, it is convenient to have a method that prints the contents of an array.

```
  public static void printArray (int[] a) {
    for (int i = 0; i<a.length; i++) {
      System.out.println (a[i]);
    }
  }
```

The following code generates an array and prints it:

```
    int numValues = 8;
    int[] array = randomArray (numValues);
    printArray (array);
```

On my machine the output is

```
27
6
54
62
54
2
44
81
```

which is pretty random-looking. Your results may differ.

If these were exam scores, and they would be pretty bad exam scores, the teacher might present the results to the class in the form of a **histogram**, which is a set of counters that keeps track of the number of times each value appear.

For exam scores, we might have ten counters to keep track of how many students scored in the 90s, the 80s, etc. The next few sections develop code to generate a histogram.

10.8 Counting

A good approach to problems like this is to think of simple methods that are easy to write, and that might turn out to be useful. Then you can combine them into a solution. Of course, it is not easy to know ahead of time which methods are likely to be useful, but as you gain experience you will have a better idea.

Also, it is not always obvious what sort of things are easy to write, but a good approach is to look for subproblems that fit a pattern you have seen before.

Back in Section 7.7 we looked at a loop that traversed a string and counted the number of times a given letter appeared. You can think of this program as an example of a pattern called "traverse and count." The elements of this pattern are:

- A set or container that can be traversed, like an array or a string.
- A test that you can apply to each element in the container.
- A counter that keeps track of how many elements pass the test.

In this case, the container is an array of integers. The test is whether or not a given score falls in a given range of values.

Here is a method called `inRange` that counts the number of elements in an array that fall in a given range. The parameters are the array and two integers that specify the lower and upper bounds of the range.

```
public static int inRange (int[] a, int low, int high) {
    int count = 0;
    for (int i=0; i<a.length; i++) {
        if (a[i] >= low && a[i] < high) count++;
    }
    return count;
}
```

In my description of the method, I wasn't very careful about whether something equal to `low` or `high` falls in the range, but you can see from the code that `low` is in and `high` is out. That should keep us from counting any elements twice.

Now we can count the number of scores in the ranges we are interested in:

```
int[] scores = randomArray (30);
int a = inRange (scores, 90, 100);
int b = inRange (scores, 80, 90);
int c = inRange (scores, 70, 80);
int d = inRange (scores, 60, 70);
int f = inRange (scores, 0, 60);
```

10.9 The histogram

The code we have so far is a bit repetitious, but it is acceptable as long as the number of ranges want is small. But now imagine that we want to keep track of the number of times each score appears, all 100 possible values. Would you want to write:

```
int count0 = inRange (scores, 0, 1);
int count1 = inRange (scores, 1, 2);
int count2 = inRange (scores, 2, 3);
...
int count3 = inRange (scores, 99, 100);
```

I don't think so. What we really want is a way to store 100 integers, preferably so we can use an index to access each one. Immediately, you should be thinking "array!"

The counting pattern is the same whether we use a single counter or an array of counters. In this case, we initialize the array outside the loop; then, inside the loop, we invoke `inRange` and store the result:

```
int[] counts = new int [100];

for (int i = 0; i<100; i++) {
    counts[i] = inRange (scores, i, i+1);
}
```

The only tricky thing here is that we are using the loop variable in two roles: as in index into the array, and as the parameter to `inRange`.

10.10 A single-pass solution

Although this code works, it is not as efficient as it could be. Every time it invokes `inRange`, it traverses the entire array. As the number of ranges increases, that gets to be a lot of traversals.

It would be better to make a single pass through the array, and for each value, compute which range it falls in. Then we could increment the appropriate counter. In this example, that computation is trivial, because we can use the value itself as an index into the array of counters.

Here is code that traverses an array of scores, once, and generates a histogram.

```
int[] counts = new int [100];

for (int i = 0; i < scores.length; i++) {
    int index = scores[i];
    counts[index]++;
}
```

Exercise 10.3 Encapsulate this code in a method called `scoreHist` that takes an array of scores and returns a histogram of the values in the array.

Modify the method so that the histogram has only 10 counters, and count the number of scores in each range of 10 values; that is, the 90s, the 80s, etc.

10.11 Glossary

array: A named collection of values, where all the values have the same type, and each value is identified by an index.

collection: Any data structure that contains a set of items or elements.

element: One of the values in an array. The `[]` operator selects elements of an array.

index: An integer variable or value used to indicate an element of an array.

deterministic: A program that does the same thing every time it is invoked.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

histogram: An array of integers where each integer counts the number of values that fall into a certain range.

10.12 Exercises

Exercise 10.4 Write a class method named `areFactors` that takes an integer `n` and an array of integers, and that returns `true` if the numbers in the array are all factors of `n` (which is to say that `n` is divisible by all of them). HINT: See Exercise 5.1.

Exercise 10.5 Write a method that takes an array of integers and an integer named `target` as arguments, and that returns the first index where `target` appears in the array, if it does, and -1 otherwise.

Exercise 10.6 Write a method called `arrayHist` that takes an array of integers and that returns a new histogram array. The histogram should contain 11 elements with the following contents:

```
element 0 -- number of elements in the array that are <= 0
        1 -- number of elements in the array that are == 1
        2 -- number of elements in the array that are == 2
        ...
        9 -- number of elements in the array that are == 9
       10 -- number of elements in the array that are >= 10
```

Exercise 10.7 Some programmers disagree with the general rule that variables and methods should be given meaningful names. Instead, they think variables and methods should be named after fruit.

For each of the following methods, write one sentence that describes abstractly what the method does. For each variable, identify the role it plays.

```
public static int banana (int[] a) {
    int grape = 0;
    int i = 0;
    while (i < a.length) {
        grape = grape + a[i];
        i++;
    }
    return grape;
}
```

```
public static int apple (int[] a, int p) {
    int i = 0;
    int pear = 0;
    while (i < a.length) {
        if (a[i] == p) pear++;
        i++;
    }
    return pear;
}

public static int grapefruit (int[] a, int p) {
    for (int i = 0; i<a.length; i++) {
        if (a[i] == p) return i;
    }
    return -1;
}
```

The purpose of this exercise is to practice reading code and recognizing the solution patterns we have seen.

Exercise 10.8

a. What is the output of the following program?

b. Draw a stack diagram that shows the state of the program just before `mus` returns.

c. Describe in a few words what `mus` does.

```
public static int[] make (int n) {
    int[] a = new int[n];

    for (int i=0; i<n; i++) {
        a[i] = i+1;
    }
    return a;
}

public static void dub (int[] jub) {
    for (int i=0; i<jub.length; i++) {
        jub[i] *= 2;
    }
}

public static int mus (int[] zoo) {
    int fus = 0;
    for (int i=0; i<zoo.length; i++) {
        fus = fus + zoo[i];
    }
    return fus;
}

public static void main (String[] args) {
    int[] bob = make (5);
```

```
    dub (bob);

    System.out.println (mus (bob));
}
```

Exercise 10.9 Many of the patterns we have seen for traversing arrays can also be written recursively. It is not common to do so, but it is a useful exercise.

a. Write a method called `maxInRange` that takes an array of integers and a range of indices (`lowIndex` and `highIndex`), and that finds the maximum value in the array, considering only the elements between `lowIndex` and `highIndex`, including both ends.

   This method should be recursive. If the length of the range is 1, that is, if `lowIndex == highIndex`, we know immediately that the sole element in the range must be the maximum. So that's the base case.

   If there is more than one element in the range, we can break the array into two pieces, find the maximum in each of the pieces, and then find the maximum of each of the piece-maxima.

b. Methods like `maxInRange` can be awkward to use. To find the largest element in an array, we have to provide a range that includes the entire array.

   ```
   double max = maxInRange (array, 0, a.length-1);
   ```

   Write a method called `max` that takes an array as a parameter and that uses `maxInRange` to find and return the largest value. Methods like `max` are sometimes called **wrapper methods** because they provide a layer of abstraction around an awkward method and provide an interface to the outside world that is easier to use. The method that actually performs the computation is called the **helper method**. We will see this pattern again in Section 14.9.

c. Write a recursive version of `find` using the wrapper-helper pattern. `find` should take an array of integers and a target integer. It should return the index of the first location where the target integer appears in the array, or -1 if it does not appear.

Exercise 10.10 One not-very-efficient way to sort the elements of an array is to find the largest element and swap it with the first element, then find the second-largest element and swap it with the second, and so on.

a. Write a method called `indexOfMaxInRange` that takes an array of integers, finds the largest element in the given range, and returns *its index*. You can modify your recursive version of `maxInRange` or you can write an iterative version from scratch.

b. Write a method called `swapElement` that takes an array of integers and two indices, and that swaps the elements at the given indices.

c. Write a method called `sortArray` that takes an array of integers and that uses `indexOfMaxInRange` and `swapElement` to sort the array from largest to smallest.

Exercise 10.11 Write a method called `letterHist` that takes a String as a parameter and that returns a histogram of the letters in the String. The zeroeth element of

the histogram should contain the number of a's in the String (upper and lower case); the 25th element should contain the number of z's. Your solution should only traverse the String once.

Exercise 10.12 A word is said to be a "doubloon" if every letter that appears in the word appears exactly twice. For example, the following are all the doubloons I found in my dictionary.

> Abba, Anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

Write a method called `isDoubloon` that returns `true` if the given word is a doubloon and `false` otherwise.

Exercise 10.13 In Scrabble each player has a set of tiles with letters on them, and the object of the game is to use those letters to spell words. The scoring system is complicated, but as a rough guide longer words are often worth more than shorter words.

Imagine you are given your set of tiles as a String, like `"qijibo"` and you are given another String to test, like `"jib"`. Write a method called `testWord` that takes these two Strings and returns true if the set of tiles can be used to spell the word. You might have more than one tile with the same letter, but you can only use each tile once.

Exercise 10.14 In real Scrabble, there are some blank tiles that can be used as wild cards; that is, a blank tile can be used to represent any letter.

Think of an algorithm for `testWord` that deals with wild cards. Don't get bogged down in details of implementation like how to represent wild cards. Just describe the algorithm, using English, pseudocode, or Java.