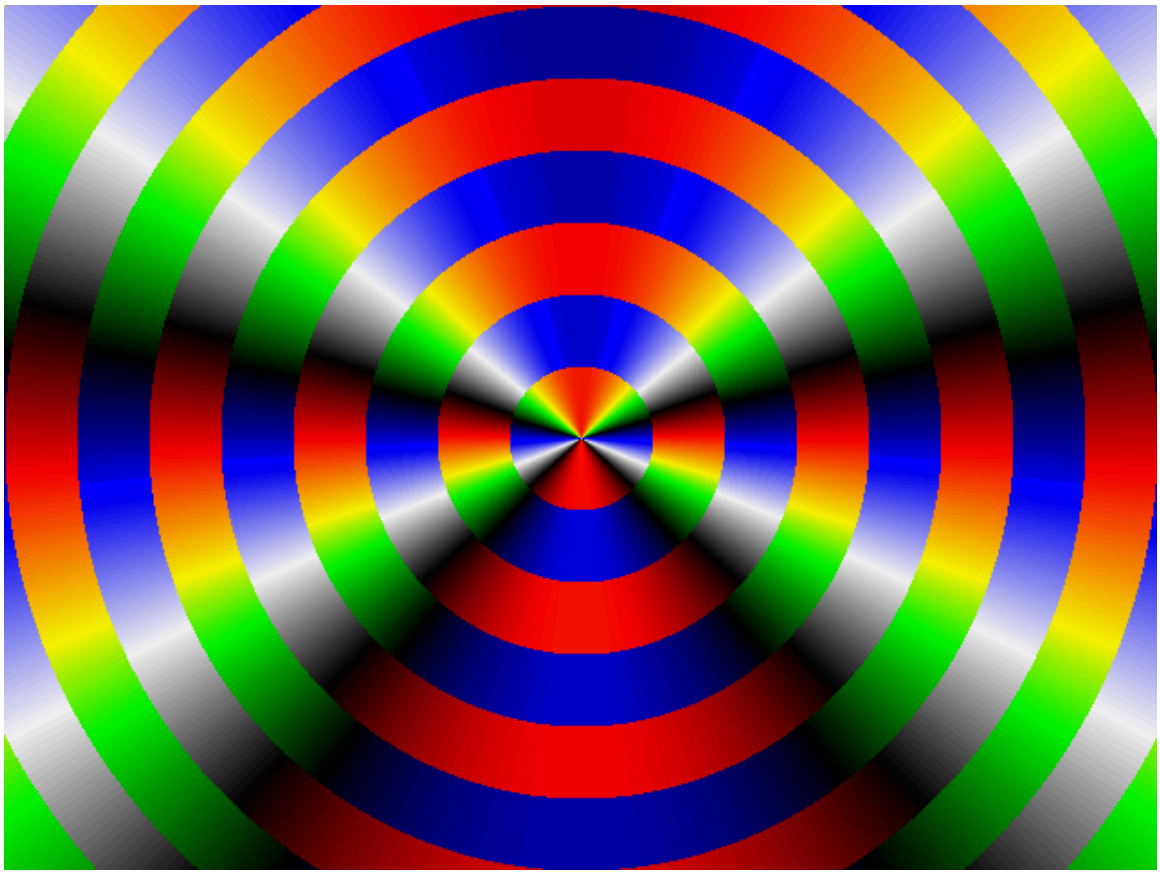
A Quick Intro to Java



Preface

This introduction works best if you already know a programming language. It designed to teach the basics of Java quickly. It is far from comprehensive. I've mostly chosen things that I use a lot in my own programs.

If you used an older version of these notes, please note that the GUI chapter was completely rewritten in May 2015 with a different approach.

There are a few hundred exercises. They are all grouped together in Chapter 8.

Last updated: October 2, 2015.

Contents

Chapter 1

Basics

1.1 Installing Java

First, you will need to download the latest version of the JDK from here:

> http://www.oracle.com/technetwork/java/javase/downloads/index.html

The JDK is the *Java Development Kit*, which contains the Java language and tools. You will also want an IDE to write your programs with. Three popular ones are NetBeans, Eclipse, and IntelliJ. NetBeans is available with the JDK download, while the others are available at the links below.

> http://www.eclipse.org/downloads/
>
> https://www.jetbrains.com/idea/download/

All three are fine. I use Eclipse, probably because it's what I'm familiar with.

If you are installing NetBeans or IntelliJ, just go with the default options.

If you are installing Eclipse, there is actually no installation. The file you download is probably compressed, so you will need to extract it. Then find the Eclipse program in the extracted folder. When you first start up Eclipse, you'll want to click on the icon that, when moused over, says "Go to the workbench." Also, Eclipse ask you to select a workspace. You can click OK and just use what they give you, or you can change it if you want.

A "Hello World" program

Here is a program in Java that prints out "Hello World":

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

For now, all of the programs we write will have a line like the first one that declares a *class*, and a line like the second one that declares the *main method*. Later on, we will learn what these things are for, but for now, just know that your program needs them. Below are instructions for getting this program up in running in each IDE.

Eclipse

Go to the `File` menu, then `New`, then `Java Project`. Under `Project Name`, give your project a name, and then click `Finish`. Then in the Package Explorer on the left, right click your project and select `New` and then `Class`. Give your class a name and make sure `public static void main(String[] args)` is checked. For now, you can ignore any warnings about using the default package.

Eclipse will then generate some code for you. In the `main` method, type the following line:

```
System.out.println("Hello, world!");
```

To run your program, choose `Run` under the `Run` Menu. You should see the output in the bottom console.

NetBeans

Go to `File` and then `New Project`. In the window that opens, `Java` should be highlighted. Choose `Java Application` and click `Next`. Under the `Project Name` field, give your project a name. You can change the location if you want. Make sure `Create Main Class` is *not* checked and click `Finish`.

In the project viewer window on the left, right-click on your project name and select `New` and `Java Class`. Give your class a name and click `Finish`. For now, you can ignore any warnings about using the default package.

NetBeans will then generate some code for you. To create the `main` method type `psvm` and press tab. NetBeans will expand that into the `main` method. Inside the `main` method, type the following line:

```
System.out.println("Hello, world!");
```

To run your program either click on the icon that looks like a play button, type F6, or select `Run Project` from the `Run` menu. You should see the output in the bottom console.

IntelliJ

Go to `File` and then `New Project`. In the window that opens, `Java` should already be highlighted and there are some options that you can skip by clicking on the `Next` button. At the next page, leave the checkbox for `Create project from template` unchecked and click `Next`. At the next page, give your project a name. You can change its location if you want. Then click the `Finish` button.

In the project viewer window on the left, right click on your project name and select `New` and `Java Class`. Give your class a name and click `Finish`. For now, you can ignore any warnings about using the default package.

IntelliJ will then generate some code for you. To create the `main` method type `psvm` and press tab. IntelliJ will expand that into the `main` method. Inside the `main` method, type the following line:

```
System.out.println("Hello, world!");
```

To run your program, select `Run 'Main'` or just `Run` and click on the highlighted option. You can also use the button near the top right that looks like a play button or use the shortcut key F9. You should see the output in the bottom console

Notes

A few important notes:

- The name of the file must match the name of the class. For instance, the class above is `HelloWorld`, and it must go in a file named `HelloWorld.java`.
- The IDE may generate a bunch of comments in your program. If you don't like them, there are settings that can be used to prevent the comments from being generated.
- Whatever IDE you are using, it is not necessary to create a new project for each new program you write. Just right-click on your project name and create a new class using the instructions above. In NetBeans, to run your new class, under the `Run` menu, select `Run File`.

1.2 Basics of Java syntax

- Most lines end in semicolons. For example,

```
x = 3;
y = y + 3;
```

- Braces are used to indicate blocks of code. Indentation is not required, but is recommended to make programs readable. For example,

```
if (x < 5)
{
    y = y + 1;
    z = z - 1;
}
```

The braces indicate the two statements that will be executed if x is less than 5. The exact placement of the braces is up to you. I like to put the starting brace on the line following the if statement, while others like to put it on the same line as the if statement.

- Comments: `//` for a single line comment, `/* */` for multiline comments:

```
x = 3;   // set x to 3

/*  This is a comment
that is spread
across several lines. */
```

1.3 Output

Here is how to print something:

```
System.out.println("This is a test.");
```

To print variables, we can do something like below:

```
int x=3, y=7;
System.out.println("x is " + x + " and y is " + y);
```

The `println` method automatically advances to the next line. If you don't want that, use the `print` method. For example,

```
System.out.print("All of ");
System.out.print("this will print on ");
System.out.print("the same line.");
```

1.4 Variables

Whenever you use a variable, you have to specify what type of variable it is before you use it. Here are the most common types:

- `int` — for integers between about -2 billion and +2 billion
- `long` — for integers between about -9×10^{18} to $+9 \times 10^{18}$
- `double` — for floating-point numbers; you get about 15 digits of precision
- `char` — for a single character
- `boolean` — true or false
- `String` — for strings of characters

The reason for the limitations on the ranges and precision has to do with how Java stores the variables internally. An `int` is 4 bytes, while longs and doubles are 8 bytes. Storing things this way allows for fast computations in hardware, but the cost is some limitation in the values that can be stored.

A Java `char` is denoted with single quotes, while strings are denoted with double quotes. The reason that there is both a character and a string class is partly historical and partly practical. A char uses less memory than a one-character string.

Here are a few typical variable declarations:

```
int p, q;
double x=2.1, y=3.25, z=5.0;
char c = 'A';
boolean foundPrime = false;
String name = "Java";
```

From the above we see that a declaration starts with a data type and is followed by one or more variables. You can initialize the variables to a value or not.

1.5 Operators

The operators `+`, `-`, `*`, and `/` work more or less the way you would expect on numbers.

One thing to be careful of is dividing two integers. For example, `x=3/2;` will actually set `x` to 1, not 1.5. The reason is that 3 and 2 are both integers, and when both operands are integers, the result will also be an integer. The decimal part of 1.5 is removed, leaving just 1. If you want 1.5, one way to fix this is to do 3.0/2.0 instead of 3/2. The same problem can also happen if `x` and `y` are both integer variables and you try `x/y`. Sometimes an integer value is what you want, but if not, then do `x/(double)y`. This is called a *cast*. See Section 2.1 for more on casts.

The modulo operator is `%`. See Section 3.4 for some uses of it.

There is no operator for raising things to powers. In Java, the operator `^` is the so-called bitwise XOR, not exponentiation. For powers, use `Math.pow`. For instance, `Math.pow(2, 5)` computes 2^5 and `Math.pow(2, .5)` computes $\sqrt{2}$.

The `+` operator can be used to concatenate strings. Variables are automatically converted to strings when added to a string with the `+` operator.

Here is an example of several of the operators in use:

```
int timeInSeconds = 780;
int minutes = timeInSeconds / 60;
int seconds = timeInSeconds % 60;
System.out.println(minutes + ":" + seconds);
```

Shortcut operators

Certain operations, like `x=x+1`, occur often enough that Java has a special shortcut syntax for them. Here are some common operations and their shortcuts:

Normal	Shortcut
`x = x + 1`	`x++`
`x = x - 1`	`x--`
`x = x + 3`	`x += 3`
`x = x * 3`	`x *= 3`

There are also `-=`, `*=`, `/=`, and `%=`, among others.

1.6 For loops

Here is a for loop that will run from `i=0` to `i=9`:

```
for (int i=0; i<10; i++)
{
    System.out.println("This is a");
    System.out.println("test.");
}
```

Braces mark off the statements to be repeated. Braces are optional if there is just one statement to be repeated, like below:

```
for (int i=0; i<10; i++)
    System.out.println("This is a test.");
```

In terms of the for statement itself, there are three parts to it: The first part, `int i=0`, declares the loop counter variable and sets it equal to its starting value, in this case 0. The second part is the condition that has to be true for the loop to keep running. In this case, the loop runs as long as the counter variable `i` is less than 10. The third part is run each time through the loop after all the statements in the loop have been done. In this case, `i++` increments the value of `i` by 1 each time through the loop. That condition can be changed to something like `i--` to run backwards or `i+=2` to increment by twos. For example, the following loop counts down from 10 to 1:

```
for (int i=10; i>=1; i--)
    System.out.println(i);
```

1.7 If statements

Here is a typical if statement:

```
if (grade >= 90)
    System.out.println("You got an A.");
```

An if statement can have `else if` and `else` blocks, like below:

```
if (grade >= 90)
    System.out.println("You got an A.");
else if (grade >= 80)
    System.out.println("You got a B.");
else if (grade >= 70)
    System.out.println("You got a C.");
else
    System.out.println("Try harder.");
```

Just like with loops, use braces if you have multiple statements to run:

```
if (grade > 100)
{
    grade = 100;
    System.out.println("You got an A.");
}
```

Logical and comparison operators

The logical and comparison operators are below:

Operator	Description
&&	and
||	or
!	not
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

1.8 Importing things

To import libraries, use `import` statements at the top of your program. For example, to use random numbers, we import `java.util.Random`, as in the program below:

```
import java.util.Random;

public class RandomPrinter
{
    public static void main(String[] args)
    {
        Random random = new Random();
        System.out.println(random.nextInt(10));
    }
}
```

Good Java IDEs will import things for you almost automatically. For instance, if you were to just type in the line `Random random = new Random()`, an IDE will flag that line as being an error. If you mouse over it, the IDE will give you the option to import `java.util.Random`.

1.9 Input

To allow the user to enter things from the keyboard, import `java.util.Scanner` and follow the examples below:

```
Scanner scanner = new Scanner(System.in);

System.out.println("Enter some text: ");
String s = scanner.nextLine();

System.out.println("Enter an integer: ");
int x = scanner.nextInt();

System.out.println("Enter a double: ");
double y = scanner.nextDouble();
```

Note that if you need to read a number and then read some text afterwards, add the following line to refresh the scanner before reading the text:

```
scanner = new Scanner(System.in);
```

1.10 Math functions

Math functions are contained in `java.lang.Math`. You don't need to import it. Instead, to use a math function, say to find sin(0), just call it like this: `Math.sin(0)`.

Java provides the following useful functions:

Function	Description
`abs(x)`	absolute value
`max(x,y)`	returns the larger of x and y
`min(x,y)`	returns the smaller of x and y
`round(x)`	rounds x to the nearest integer
`floor(x)`	returns the greatest integer $\leq x$
`ceil(x)`	returns the least integer $\geq x$
`pow(x,y)`	returns x^y

Java also provides trigonometric, logarithmic, and other functions.

1.11 Random numbers

To use random numbers, import `java.util.Random`. Here is an example that sets `x` to a random integer from 0 to 9:

```
Random random = new Random();
int x = random.nextInt(10);
```

Calling `nextInt(b)` returns a random integer from 0 to $b-1$. If you want a random number from a to b, use the following:

```
int x = random.nextInt(b-a+1) + a
```

For example, to get a random number from 10 to 15, use `random.nextInt(6)+10`. The call to `nextInt` will return a random number from 0 to 5, and adding 10 to it gives a random number from 10 to 15. Another way to think of this is as giving 6 random numbers starting at 10.

The `Random` class also contains some other occasionally useful methods. See the Java API documentation.

1.12 While loops

For loops are useful when you know how many times you need to repeat something. While loops are useful when you need to repeat something for an unknown number of times, usually until something happens. Here is a simple example. The user enters numbers to be summed up and indicates they are done by entering a 0.

```
Scanner scanner = new Scanner(System.in);
int sum = 0;

while (x != 0)
{
    System.out.println("Enter a number (0 to stop): ");
    x = scanner.nextInt();
    sum += x;
}
```

1.13 Strings

Java contains a number of tools for working with strings. As a small demonstration, suppose `s` is a string variable that contains a name followed by an email address in parentheses and we want to extract the email address. Here is how we could do that:

```
String s = "John Q. Smith (JQSmith@gmail.com)";
String email = s.substring(s.indexOf("(")+1, s.indexOf(")"));
```

The code above makes use of the `substring` method that returns a part of a string, and the code makes use of the `indexOf` method that returns the location of the first occurrence of something.

List of common string methods

Method	Description
`length()`	returns the string's length
`charAt(i)`	returns the character at index `i`
`indexOf(s)`	returns the location of the first occurrence of the character/substring `s`
`equals(s)`	returns whether the string equals `s`
`equalsIgnoreCase(s)`	like above, but ignores upper/lowercase

`substring(i)`	returns the substring from index `i` to the end
`substring(i,j)`	returns the substring from index `i` to `j-1`
`toLowerCase()`	returns the string with all uppercase changed to lowercase
`toUpperCase()`	returns the string with all lowercase changed to uppercase
`replace(s,t)`	returns the string where each substring `s` is replaced with `t`
`contains(s)`	returns whether the string contains the substring `s`
`startsWith(s)`	returns whether the string starts with the substring `s`
`endsWith(s)`	returns whether the string ends with the substring `s`
`trim()`	returns the string with any whitespace at the start and end removed
`compareTo(s)`	compares strings alphabetically

Note that none of these methods affect the original string. For instance, just calling `s.toLowerCase()` will return a lowercase version of `s` but will not change `s`. In order to change `s`, do the following:

```
s = s.toLowerCase();
```

Useful things to do with strings

Comparing strings

The following code to check if two strings are equal *will not work*:

```
if (s == "Java")
```

Instead, use the `equals` method, like below:

```
if (s.equals("Java"))
```

This is just the way Java is. The `==` operator on strings actually compares the memory locations of the strings instead of their values. Similarly, for comparing strings alphabetically, the `<` and `>` operators won't work. Instead, use the `compareTo` method.

Substrings

The `substring` method is used to return a part of a string. Here are a few examples:

```
s.substring(0,3)          // first three characters of s
s.substring(2,5)          // characters 2, 3, 4 (the last index is not included)
s.substring(2)            // characters from index 2 to the end of the string
s.substring(s.length()-3) // last three characters
s.substring(i,i+1)        // character at index i as a string
```

Remember that Java makes a distinction between single characters and strings. Calling `s.substring(i,i+1)` will return a string containing a single character. On the other hand, calling `s.charAt(i)` will return that same character, but as an object of type `char`. Both ways have their uses.

Removing things from a string

Suppose we want to remove all the commas from a string `s`. We can use the replace method as follows:

```
s = s.replace(",", "");
```

This works by replacing the commas with an empty string. Notice that we have to set `s` equal to the new result. `s.replace` by itself will not change the original string.

Removing the last character

To remove the last character from a string, the following can be used:

```
s = s.substring(0, s.length()-1);
```

Using `indexOf`

The `indexOf` method gives us the location (index) of the first occurrence of something in a string. For instance, suppose we have an email address of the form `smith@gmail.com` stored in a variable called `email` and we want the username, the part before the `@` symbol. We could use the `substring` method, but we don't know how long the username will be in general. The `indexOf` method saves us from having to worry about that. To get the username, we find the substring starting at 0 and ending at the location of the `@` symbol, like below:

```
String username = email.substring(0, email.indexOf("@"));
```

Converting strings to numbers

If you need to convert a string `s` to an integer or double, do the following:

```
int x = Integer.parseInt(s);
double y = Double.parseDouble(s);
```

These are useful if you need to extract a numerical value from a string to do some math to. For example, suppose `s` is a string containing a filename like `pic1023.jpg` and we want to increase the number in it by 1. Assuming the numerical part of the filename runs from the `c` to the period, we could do the following:

```
String s = "pic1023.jpg";
int n = Integer.parseInt(s.substring(s.indexOf("c")+1, s.indexOf(".")));
s = "pic" + (n+1) + ".jpg";
```

Converting numbers to strings

To convert numbers to strings, there are a few options. First, the string concatenation operator, `+`, will interpret numbers as strings. Here are some examples of it in use:

```
int x = 2;
System.out.println("The value of x is " + x);
String s = "(" + x + "," + (x*x) + ")";
String s = "" + x;
```

Some people find `""+ x` to be poor style and recommend the following way to convert something to a string:

```
String s = String.valueOf(x);
```

Looping through strings

Here is how to loop over the characters of a string:

```
for (int i=0; i<s.length(); i++)
    // do something with i, s.charAt(i), and/or s.substring(i,i+1)
```

For example, the code below counts how many spaces are in the string `s`:

```
int count = 0;
for (int i=0; i<s.length(); i++)
    if (s.charAt(i) == ' ')
        count++;
```

Building up a string

It is often useful to build up a new string one character at a time. Here is an example that takes a string called s that we will assume consists of several lowercase words separated by spaces and capitalizes the first letter of each word.

```
String capitalized = ""
for (int i=0; i<s.length; i++)
{
    if (s.charAt(i-1)==' ' or i==0)
        capitalized += s.substring(i,i+1).toUpperCase();
    else
        capitalized += s.substring(i,i+1));
}
```

The way it works is if the previous character is a space (or if we are at the start of the string), then we capitalize the current character. We don't actually change the original string. Rather, we create a new one that starts initially empty and we add to it one character at a time, adding at each step either a capitalized version of the current character from s or just the character itself.

Here is another example that shows how to use this adding process to reverse a string.

```
String reversed = "";
for (int i=s.length()-1; i>=0; i--)
    reversed += s.charAt(i);
```

This works fine for building up small strings, but it is slow for large strings. In that case use `StringBuilder` (Section 2.17).

1.14 ASCII codes and escape characters

Each character on the keyboard is associated with a numerical code called its ASCII code. For instance, the code for A is 65. A list of common codes is shown below.

32	space	44	,	65–90	A–Z
33	!	45	-	91	[
34	"	46	.	92	\
35	#	47	/	93	]
36	$	48–57	0–9	94	^
37	%	58	:	95	_
38	&	59	;	96	`
39	'	60	<	97–122	a–z
40	(	61	=	123	{
41	)	62	>	124	|
42	*	63	?	125	}
43	+	64	@	126	~

A character can be specified either by its symbol or its code, as below:

```
char c = 'A'
char c = 65;
```

Working with ASCII codes can allow you to do some interesting things. For instance, here is some code that generates a random 100-character string of capital letters.

```
Random random = new Random();
String s = "";
for (int i=0; i<100; i++)
    s += (char)(random.randint(26) + 65;
```

The (char) in parentheses is a type cast that tells Java to interpret the integer ASCII code generated by randint as a character. Here is another example that rotates all the letters in a string s of capitals forward by 1 letter.

```
String t = "";
```

```
for (int i=0; i<s.length(); i++)
    t += (char)(((s.charAt(i) - 65) + 1) % 26 + 65);
```

There is another standard called Unicode that extends the ASCII standard to include a wide variety of symbols and international characters. Working with Unicode can be a little tricky. If you need to work with Unicode, consult the web or a good reference book.

Escape characters

Here are a few special characters that are good to know:

`\n`	newline
`\t`	tab
`\"`	double quote (useful if you need a quote in a string)
`\'`	single quote
`\\`	backslash

For example, the following prints 5 blank lines:

```
System.out.print("\n\n\n\n\n");
```

1.15 Arrays

An array is a fixed-size, contiguous block of memory all of one data type, useful for storing collections of things.

Creating arrays

Here are a few sample array declarations:

```
int[] a = new int[100];       // array of 100 ints, all initialized to 0
int[] a = {1,2,3,4,5};        // array of consisting of the integers 1 through 5
String[] a = new String[20];  // array of 20 strings, all initialized to ""
```

Basic array operations

The length of an array `a` is `a.length`, with no parentheses.

The first element of `a` is `a[0]`, the second is `a[1]`, etc. The last is `a[a.length-1]`.

To change an element of an array, do something like the following (which sets the element at index 7 to 11):

```
a[7] = 11;
```

Looping

To loop through all the elements in an array, there are two ways. The long way is to do the following:

```
for (int i=0; i<a.length; i++)
    // do something with a[i] and i
```

Here is the shorthand (for an array of strings called `a`):

```
for (String x : a)
    // do something with x
```

The shorthand works with other data types. Just replace `String` with the data type of the array.

Printing an array

To print out all the elements in an array a of doubles, one per line, do the following:

```
for (double x:a)
    System.out.println(x);
```

This will work with other data types; just replace `double` with the appropriate data type. Another way to print the contents of an array is to import `java.util.Arrays` and use `Arrays.toString`:

```
System.out.println(Arrays.toString(a))
```

A few examples

Here is some code that creates an array containing the numbers 1 through 100:

```
int[] a = new int[100];
for (int i=0; i<100; i++)
    a[i] = i + 1;
```

Here is some code that adds 1 to every element of an array `a`:

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

1.16 Lists

One problem with arrays is their size is fixed. It often occurs that you don't know ahead of time how much room you will need. A *list* is a data structure like an array that can grow when needed. Most of the time in Java you will want to work with lists instead of arrays, as lists are more flexible and come with a number of useful methods. Here is a how to declare a list in Java:

```
List<Integer> list = new ArrayList<Integer>();
```

The `Integer` in the slanted braces indicates the data type that the list will hold. This is an example of Java generics (see Section 4.8 for more on those). Notice that it is `Integer` and not `int`. We can put any class name here. Here are some examples of other types of lists we could have:

`List<String>` — list of strings
`List<Double>` — list of doubles
`List<List<Integer>>` — list of integer lists

To use lists, we need to import `java.util.List` and `java.util.ArrayList`.

Printing a list

Printing a list is easy. To print a list called `list`, just do the following:

```
System.out.println(list);
```

Modifying a list

One drawback of lists is the array notation `a[i]` does not work with lists. Instead, we have to use the `get` and `set` methods. Here are some examples showing array and list operations side by side.

```
a[2]          list.get(2)
a[2] = 4      list.set(2, 4)
a[2]++        list.set(2, list.get(2)+1)
```

Adding things to a list

The `add` method is useful for filling up a list. By default, it adds things to the end of a list. Here is a simple example that fills up a list with numbers from 1 to 100:

```
List<Integer> list = new ArrayList<Integer>();
for (int i=1; i<=100; i++)
    list.add(i);
```

Unlike with arrays, there is no easy way to initialize a list when it is declared. However, `Collections.addAll` can be used for this purpose. Here is an example:

```
List<Integer> primes = new ArrayList<Integer>();
Collections.addAll(primes, 2, 3, 5, 7, 11, 13, 17, 19);
```

Inserting and deleting things

The `add` method can used to an item into the middle of a list. Here is an example that inserts 99 at index 1.

```
list.add(1, 99);
```

If the list had been `[7, 8, 9]`, the new list would be `[7, 99, 8, 9]`.

To remove the element at a specified index, use the `remove` method. For instance, to remove the element at index 1, do the following:

```
list.remove(1);
```

Making a copy of a list

A simple way to make a copy of a list called `list` is shown below:

```
List<Integer> copy = new ArrayList<Integer>(list);
```

`List` methods

Besides the methods already mentioned, here are some other useful methods for working with lists.

Method	Description
`contains(x)`	returns whether `x` is in the list
`indexOf(x)`	returns the location of the first occurrence of `x` in the list
`isEmpty()`	returns whether the list is empty
`size()`	returns the number of elements in the list
`sort()`	sorts the elements (Java 8 and later)
`subList(i,j)`	returns a slice of a list from index `i` to `j-1`, sort of like `substring`

Methods of `java.util.Collections`

Below are some useful methods found in `java.util.Collections` that operate on lists. See the Java API documentation for more methods.

Method	Description
`addAll(list,x1,...,xn)`	adds `x1`, `x2`, and `xn` to `list`

`binarySearch(list,x)`	returns whether `x` is in `list` (`list` must be sorted)
`max(list)`	returns the largest element in `list`
`min(list)`	returns the smallest element in `list`
`reverse(list)`	reverses the elements of `list`
`shuffle(list)`	puts the contents of `list` in a random order
`sort(list)`	sorts `list`

Note that the `binarySearch` method for large lists is much faster than the list's `contains` method. However, it will only work if the list is sorted.

The `shuffle` method

Of all the Collections methods, the `shuffle` method is particularly useful. For instance, a quick way to come up with a random ordering of players for a game is shown below:

```
List<String> players = new ArrayList<String>();
Collections.addAll(players, "Andrew", "Bob", "Carol", "Deb", "Erin");
Collections.shuffle(players);
System.out.println(players);
```

As another example, if we need five random elements from a list, we can shuffle it and then use the first five elements of the shuffled list.

Iterating

Just like with strings and arrays, there is a long way and a short way to iterate over lists. Here is the long way:

```
for (int i=0; i<list.size(); i++)
    // do something with list.get(i) and i
```

Here is the shortcut (for a list of doubles, other types are similar):

```
for (double x:list)
    // do something with x
```

1.17 Two-dimensional arrays

The following creates a 3×5 array of integers, all initially set to zero:

```
int[][] a = new int[3][5];
```

The following creates a 3×5 array of integers, specifying the initial values:

```
int[][] a = { {1, 4, 5, 8, 3},
              {2, 4, 4, 0, 4},
              {9, 8, 9, 1, 8} };
```

Getting entries

To get the entry in row i, column j, use the following (remember that indices start at 0):

```
a[i][j]
```

Number of rows and columns

You can use `a.length` to get the number of rows and `a[0].length` to get the number of columns.

Printing a 2d array

To print out the contents of a 2d array, use the following:

```
for (int i=0; i<a.length; i++)
{
    for (int j=0; j<a[i].length; j++)
        System.out.print(a[i][j] + " ");
    System.out.println();
}
```

The inner loop prints out one row at a time, and the empty print statement advances to the next line after each row is printed. Nested loops like this are useful for working with two-dimensional arrays.

Working with two-dimensional arrays

Here are some examples of how to work with 2d arrays:

- Adding up all the entries in the 2d array:

```
int total = 0;
for (int i=0; i<a.length; i++)
    for (int j=0; j<a[i].length; j++)
        total += a[i][j];
```

- Adding up the entries in the third row:

```
int total = 0;
for (int i=0; i<a[2].length; i++)
    total += a[2][i];
```

- Adding up the entries in the third column:

```
int total = 0;
for (int i=0; i<a.length; i++)
    total += a[i][2];
```

- Adding up the entries along the diagonal:

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

- Adding up the entries below the diagonal:

```
int total = 0;
for (int i=0; i<a.length; i++)
    for (int j=0; j<i; j++)
        total += a[i][j];
```

1.18 Arrays versus lists

Lists are probably the better choice most of the time. In particular, lists would be appropriate if the number of elements can vary or if having methods like `indexOf` or `contains` would be helpful.

On the other hand, sometimes, the `get` and `set` methods can get a little messy and confusing. For instance, suppose you need to add 1 to the value of an element in a list or array. Shown below are how to that with arrays and lists:

```
a[i]++;
list.set(i, list.get(i)+1);
```

The `get`/`set` methods get particularly annoying if you need two work with a two-dimensional list. For example, here are the ways to add 1 to an element of a two-dimensional array and a two-dimensional list.

```
a[i][j]++;
list.get(i).set(j, list.get(i).get(j)+1);
```

In summary, if you don't need all the power of lists and the `get`/`set` notation gets too cumbersome, then arrays are a good choice. Otherwise, go with lists.

1.19 Reading and writing to text files

Reading from a file

The `Scanner` class is versatile enough that it can handle input not only from the keyboard, but also from a file. Here is an example that reads each line of a file into a list of strings:

```
public static void main(String[] args) throws FileNotFoundException
{
    List<String> lines = new ArrayList<String>();
    Scanner textFile = new Scanner(new File("filename.txt"));
    while (textFile.hasNextLine())
        lines.add(textFile.nextLine());
}
```

We include the `main` method here because Java requires that we consider what happens if the file is not found. In this case we choose to do nothing (which is what the `throws` statement is about). You will need to do something about the `FileNotFoundException` whenever working with files in Java. See Section 4.12 for more about exceptions.

Important note: If you need to reread from the same file, you will have to refresh the scanner. Use a line like the following before rereading the file:

```
textFile = new Scanner(new File("filename.txt"));
```

Reading a file into a string

In Java 7 or later, if you want to read an entire text file into a single string, use the following:

```
String text = new String(Files.readAllBytes(Paths.get("filename.txt")));
```

Reading a file with several things on each line

Here is an example that reads a file where each line consists of a name followed by three integer test scores. The code then prints out the average of the scores on each line:

```
Scanner textFile = new Scanner(new File("filename.txt"));
while (textFile.hasNextLine())
{
    String name = textFile.next();
    int s1 = textFile.nextInt();
    int s2 = textFile.nextInt();
    System.out.println(name + "'s average: " +  (s1+s2)/2.0);
}
```

Below is another approach. It uses Java's `split` method (see Section 2.2), which breaks up a string at a specified character and returns an array of the pieces.

```
Scanner textFile = new Scanner(new File("filename.txt"));
while (textFile.hasNextLine())
{
    String[] a = textFile.nextLine().split(" ");
    String name = a[0];
    int s1 = Integer.parseInt(a[1]);
    int s2 = Integer.parseInt(a[2]);
    System.out.println(name + "'s average: " + (s1+s2)/2.0);
}
```

CSV files

A CSV file is a "comma-separated value" file. A typical line of a CSV contains several values, all separated by commas, maybe something like this:

```
Smith, sophomore, 3.485, 92, computer science
```

CSVs are important because many data files are stored in the CSV format. In particular, spreadsheet programs have an option to save the spreadsheet as a CSV. For reading simple CSV files, the techniques above work fine. We just read the file line-by-line and split each line at commas. Sometimes, though, CSVs can be a little tricky to handle, like if the individual fields themselves have comma in them. In this case, it might be worth searching the internet for a good Java CSV library.

Writing to files

Here is a simple way to write to a file:

```
PrintWriter output = new PrintWriter("outfile.txt");
output.println("Hello, file!");
output.close();
```

If you don't close the file when you are done with it, your changes might not be saved.

The `PrintWriter` class also has `print` and `printf` methods.

Appending to files

The above technique will overwrite whatever is in the file, if the file already exists. If you want to append to the file, that is add stuff onto the end of it, then change how the PrintWriter object is defined, like below:

```
PrintWriter output = new PrintWriter(new FileWriter("outfile.txt", true));
```

If the file doesn't already exist, this will create a new file. If the file does exist, all print statements will add to the end of the file, preserving the original data.

Chapter 2

Miscellaneous Programming Topics

2.1 Type casts

A lot of programming errors arise when programmers mix up data types. For instance, if you try assign a double value to an integer, you would be trying to store something with fractional values into a whole number. Some programming languages will allow you to do this by automatically dropping the fractional part. Java, on the other hand, will flag this as an error. Often this will save you from making a mistake, but if it is something that you really want to do, you can use a *type cast* to essentially tell Java that you know what you are doing. See below:

```
int x = Math.sqrt(4);       // error b/c Math.sqrt returns a double
int x = (int)Math.sqrt(4);  // x will be set to 2
```

A type cast consists of a data type in parentheses and it tells Java to interpret the expression immediately following as that data type. You may need to use parentheses around the expression, like below (assume `x`, `y`, and `z` are doubles):

```
int x = (int)(x+y+z);
```

Here is another example. Suppose we have a list of integers and we want to find the average of the integers. Suppose we store the sum of the entries in an integer variable called `sum`. Consider the two approaches below:

```
double average = sum / list.size();
double average = sum /(double) list.size()
```

The the first approach will not give a very accurate result. The problem is that both `sum` and `list.size()` are integers, and the result of dividing two integers will be an integer. For instance, the result of `5/2` would be 2 instead of 2.5. However, if we use a cast, like the one in the second line, to cast one of the values to a double, then the result of the division will be a double.

Here is one more example. Suppose we want to generate random capital letter. One approach is to use the fact that the ASCII codes for capital letters run between 65 and 90 like below:

```
Random random = new Random();
char c = random.nextInt(26) + 65; // error
```

But Java will flag an error because we are trying to assign a 32-bit integer to an 8-bit character. We can use a cast, like below, to fix that:

```
Random random = new Random()
char c = (char)(random.nextInt(26) + 65);
```

2.2 The `split` method

Java's String class has a very useful method called `split` that is used to break a string into pieces. Here are some short examples:

```
String s = "this is a test.";
String[] a = s.split(" ");
// a = ["this", "is", "a", "test."]

String s = "2/29/2004";
String[] a = s.split("/");
// a = ["2", "29", "2004"]
```

In the second example, you might want to use the following code to convert the strings to integers:

```
int month = Integer.parseInt(a[0]);
int day = Integer.parseInt(a[1]);
int year = Integer.parseInt(a[2]);
```

The `split` method can be used to split by a string of any length. It is also possible to split at patterns, like at any digit or any whitespace, by specifying what are called *regular expressions* as the argument to `split`. There is a lot to regular expressions, but here are a few simple examples to show what is possible:

```
String[] a = s.split(",|;");  // split at commas or semicolons
String[] a = s.split("\\s+"); // split at any whitespace
String[] a = s.split("\\d");  // split at any single digit
```

Here is an example. Suppose we have a string called `text` that consists of integers separated by spaces, maybe varying amounts of spaces, and we want to add up all of those integers. The code below will do that:

```
int sum = 0;
for (String s : text.split("\\s+"))
    sum += Integer.parseInt(s);
```

2.3 Uses for scanners

Three common uses for scanners are to get keyboard input, to read from a text file, and to parse a string.

```
Scanner scanner = new Scanner(System.in);             // get keyboard input
Scanner textFile = new Scanner(new File("file.txt")); // get input from file
Scanner scanner = new Scanner(someString);            // parse a string
```

Here is an example of how to use scanners to parse a string. Suppose we have a string `s` that consists of a user ID, followed by three integers, followed by a double. Suppose everything is separated by spaces. We can use the following code to break up the string:

```
Scanner scanner = new Scanner(s);
String id = scanner.next();
int num1 = scanner.nextInt();
int num2 = scanner.nextInt();
int num3 = scanner.nextInt();
double num4 = scanner.nextDouble();
```

We could have also done this using the `split` method.

Changing the scanner's delimiter

By default, scanners use whitespace to determine where things start and end. The scanner method `useDelimiter` can be use to change that to something else. For instance, if we have a string whose tokens are delimited by commas, we can do something like below:

```
Scanner scanner = new Scanner("name,88,4.33");
scanner.useDelimiter(",");
String s = scanner.next();
int x = scanner.nextInt();
double y = scanner.nextDouble();
```

2.4 Short-circuiting

operators!short-circuiting

Suppose you need to check to see if `a[i]` is positive, where `a` is an array and `i` is some index. It's possible that `i` could be beyond the bounds of the array, and if it is, then trying to access `a[i]` will crash your program. The solution is to use the following if statement:

```
if (i<a.length && a[i]>0)
```

It might seem like we would still run into the same problem when we check `a[i]>0`, but we don't. Recall that for an `&&` operation to be true, both its operands must be true. If it turns out that `i` is not less than `a.length`, then the `&&` expression has no chance of being true. Java will recognize that fact and not even bother evaluating the second half of the expression. This behavior is called *short-circuiting* and it is something you can depend on.

The `||` operator works similarly. As long as one or the other of its operands is true, the expression will be true, so if the first half is true, it will not bother checking the second half.

2.5 Scope of variable declarations

A variable can be declared within a loop or conditional statement, like in the example below:

```
for (int i=0; i<10; i++)
{
    int square = i * i;
    System.out.println(s);
}
```

The variable `square` only exists within the loop and cannot be accessed outside of it. Similarly, the loop variable `i` in the for loop does not exist outside the for loop.

2.6 Infinite loops

If you need some code to run indefinitely, you can enclose it in the following loop:

```
while (true)
{
    // do something
}
```

2.7 Switch-case

Java has a switch-case statement that can replace a long sequence of if statements. Here is an example:

```
switch (grade)
{
case 'A':
    System.out.println("Excellent!");
    break;
case 'B':
    System.out.println("Good.");
    break;
case 'C':
    System.out.println("Okay.");
    break;
case 'D':
    System.out.println("Not so good.");
    break;
case 'F':
    System.out.println("No good.");
    break;
```

```
default :
    System.out.println("Not a valid grade.");
}
```

The `default` statement is optional. Notice that there are no braces and that cases end with `break` statements.

2.8 Printf

The `printf` method is useful for printing out things formatted nicely. For instance, to print a variable called `cost` to two decimal places, we could do the following:

```
System.out.printf("%.2f", cost);
```

The `%.2f` part is a *formatting code*. The `%` symbol starts the code. The `f` stands for *floating point* and the `.2` specifies that the cost should be rounded and displayed to two decimal places.

Here is another example:

```
System.out.printf("The cost is %.2f and the price is %.2f\n", cost, price);
```

We see that what happens is that the formatting codes act as placeholders for the arguments specified later in the function call. The first `%.2f` is replaced by the value stored in `cost` and the second is replaced by the value stored in `price`. Notice also the use of the newline character. Unlike `println`, `printf` does not automatically advance to the next line.

Here is a list of the most useful formatting codes. There are some other, less useful codes that a web search will turn up if you're interested.

Code	Description
%f	for floating point numbers
%d	for integers
%s	for strings
%g	chooses between regular or scientific notation for floats

Formatting codes can be used to line things up. Here is an example:

```
int x=4, y=17, z=2203;
System.out.printf("%4d\n", x);
System.out.printf("%4d\n", y);
System.out.printf("%4d\n", z);
```

```
   4
  17
2203
```

The code `%4d` says to allot 4 characters to display the integer. If the integer is less than four digits long, then it is padded by spaces on the left. The result is that the integers appear to be right-justified.

To left-justify, use a minus sign in the formatting code, like `%-4d`. In this case, any extra spaces will be added on the right. For instance, `%-12s` will allot 12 characters for a string, adding spaces on the right of the string as needed to bring it up to 12 characters.

As another example, the code `%6.2f` allots six characters for a floating point variable. Two of those will be to the right of the decimal point, one is for the decimal point, and the other three are for stuff to the left of the decimal point.

Here is how we might print a product name and price, nicely lined up.

```
System.out.printf("%15s %6.2f", product, price);
```

There are a few other things you can do with formatting codes. Here are some examples:

```
int x=2, y=29239;
System.out.printf("%04d\n", x); // pad with zeroes
```

```
System.out.printf("%,d\n", y);  // add commas
System.out.printf("%+d\n", y);  // force + sign if positive
System.out.printf("% d\n", y);  // leave a space if positive
```

```
0002
29,239
+29239
 29239
```

The latter two are useful for lining up values that can be positive or negative.

Java's String class has a method called `format` that can be used to store the result of printf-style formatting to a string. Here is a simple example:

```
String s = String.format("%.2f", 3.14159);
```

Anything you can do with `printf`, you can do with `String.format`, and the result will be saved to a string instead of printed out.

The Java Library has classes like `DecimalFormat`, `NumberFormat`, and `DateFormat` that can be used for more sophisticated formatting.

2.9 The break and continue statements

break

The break statement is used to break out of a loop early. Here is an example: Suppose we want the user to enter numbers and sum up those numbers, stopping when a negative is entered. Here is a way to do that with a break statement:

```
int sum = 0;
Scanner scanner = new Scanner(System.in);
while (true)
{
    int x = scanner.nextInt();
    if (x < 0)
        break;
    sum += x;
}
```

This could be written without a break statement, but it would be a little more clumsy.

continue

Suppose we have a list of positive and negative integers, and we want to find the sum of the positives, as well as find the largest element. The following code will do that:

```
int total=0, max=list.get(0);
for (int x : list)
{
    if (x < 0)
        continue;
    total += x;
    if (x > max)
        max = x;
}
```

It uses Java's `continue` statement to essentially skip over the negatives. When the program reaches a `continue` statement, it jumps back to the start of the loop, skipping everything after the `continue` statement in the loop.

Note that we could do this without the `continue` statement, just using an if statement, but the `continue` code is arguably a bit easier to follow (at least once you are familiar with how `continue` works).

In general, `break` and `continue` statements can lead to clearer code in certain situations.

2.10 The ternary operator

Java has an operator that can take the place of an if/else statement. First, here is an if/else statement:

```
if (x < 0)
    y = -1;
else
    y = 1;
```

We can rewrite this using Java's ternary operator as follows:

```
y = x<0 ? -1 : 1;
```

The general syntax of the operator is

```
<condition> ? <statement to do if condition is true> : <statement to do otherwise>
```

Here is another example. Consider the following if/else statement:

```
if (x == 1)
    System.out.println("You bought 1 item.");
else
    System.out.println("You bought " + x + "items.");
```

We could replace it with the line below using the ternary operator:

```
System.out.println("You bought " + x + "item" + (x==1 ? "" : "s"));
```

Just like with `break` and `continue` statements, the ternary operator is not strictly necessary, but (once you get used to it) it can make code shorter and easier to read, when used in appropriate situations.

2.11 Do...while Loop

In some programs, it is more convenient to check a condition at the end of a loop instead of at the start. That is what a do...while loop is useful for. Here is an example that repeatedly asks the user to enter numbers to be summed:

```
Scanner scanner = new Scanner(System.in);
int sum = 0;
do
{
    System.out.print("Enter a number: ");
    sum += scanner.nextInt();
    System.out.print("Keep going? (Y/N)");
} while (scanner.next().equalsIgnoreCase("Y"));
System.out.println(sum);
```

We could rewrite this using an ordinary while loop, but the code would be a little more cumbersome.

2.12 Boolean expressions

An expression like `x < 10` evaluates to `true` or `false` and can be assigned to a boolean variable like below.

```
boolean isSmall = x < 10;
```

Expressions like this are sometimes used in the return statement of a function (functions are covered in Section 4.2). For instance, a function that determines if a string has an even number of characters could be written as below:

```
public static boolean isEvenString(String s)
{
    return s.length() % 2 == 0;
}
```

2.13 Useful character methods

Here a few ways to check things about a character:

```
if (Character.isDigit(c))
if (Character.isLetter(c))
if (Character.isLetterOrDigit(c))
if (Character.isUpperCase(c))
if (Character.isWhitespace(c))
```

Also, `Character.getNumericValue` can be used to turn a character like `'2'` into an integer. And `Character.toLowerCase` and `Character.toUpperCase` can be used to convert characters to lowercase and uppercase. See the Java API documentation for more occasionally useful `Character` methods.

2.14 Lists of lists

Sometimes, you need a list of lists, i.e., a list whose items are themselves lists. Here is a declaration of a list whose items are lists of strings:

```
List<List<String>> list = new ArrayList<List<String>>();
```

We then have to declare the individual string lists. Say we will have 100 of them. Here is what to do:

```
for (int i=0; i<100; i++)
    list.add(new ArrayList<String>())
```

This fills the master list with 100 empty string lists. If we don't do this, we will end up with an error when we try to access the individual string lists. Here is how to add something to the first list followed by how to get the first value of the first list:

```
list.get(0).add("hello")
list.get(0).get(0)
```

2.15 Sets

A set is an unordered collection with no repeated elements. We declare a set in a similar way to how we declare a list:

```
Set<Integer> set = new LinkedHashSet<Integer>();
```

There are three kinds of sets we can use: `TreeSet`, `HashSet`, and `LinkedHashSet`. Use `TreeSet` if you want to store things in the set in numerical/alphabetical order. Use `LinkedHashSet` otherwise. It is faster than `TreeSet` and stores elements in the order in which they were added. `HashSet` is very slightly faster than `LinkedHashSet`, but scrambles the order of elements.

Here are some useful methods of the `Set` interface:

Method	Description
`add(x)`	adds `x` to the set
`contains(x)`	returns whether `x` is in the set
`difference(t)`	returns a set of the elements of `s` not in the set `t`
`intersection(t)`	returns a set of the elements in both `s` and `t`
`remove(x)`	removes `x` from the set
`union(t)`	returns a set of all the elements that are in `s` or `t`

We can loop through the items of a set like below (assume it's an integer set called `set`):

```
for (int x : set)
    // do something with x
```

Note that, unlike lists, sets do not have a `get` method. Sets are inherently unordered, which means it doesn't make sense to ask what is at a specific index.

A useful example

Consider the following code that reads all the words from a file containing dictionary words and stores them in a list.

```
List<String> words = new ArrayList<String>();
Scanner textFile = new Scanner(new File("wordlist.txt"));
while (textFile.hasNext())
    words.add(textfile.next());
```

If we want to later know if a given word `w` is in the list, we can use the following:

```
if (words.contains(w))
```

We can realize a several orders of magnitude speedup if we use a set in place of the list. None of the other code changes at all. Just change the first line to the following:

```
Set<String> words = new LinkedHashSet<String>();
```

For example, suppose we have 40,000 words, and we want to find all the words that are also words backwards, like *bad/dab* or *diaper/repaid*. We can use the code above to fill the list/set. The code below will then find all the desired words:

```
List<String> list = new ArrayList<String>();
for (String w : words)
{
    String b = "";
    for (int i=w.length()-1; i>=0; i--)
        b += w.charAt(i);
    if (words.contains(b))
        list.add(w);
}
```

When I tested this, lists took about 20 seconds, while sets took 0.1 seconds.

Removing duplicates from a list

Suppose we have an integer list called `list`. A fast way to remove all duplicates from the list is to convert it to a set and then back to a list. This is shown below:

```
list = new ArrayList<Integer>(new LinkedHashSet<Integer>(list));
```

Checking if a list contains repeats

Suppose we want to know if an integer list called `list` contains repeats. The following uses a set for this purpose:

```
if (new HashSet<Integer>(list).size() == list.size())
    System.out.println("No repeats!");
```

It creates a set from the list, which has the effect of removing repeats, and checks to see if the set has the same size as the original list. If so, then there must have been no repeats.

2.16 Maps

A map is a generalization of a list, where instead of integers, the indices can be more general objects, especially strings. Here is an example:

```
Map<String, Integer> months = new LinkedHashMap<String, Integer>();
```

```
months.put("January", 31);
months.put("February", 28);

System.out.println(months.get("January"));
```

This prints out 31. The month names in the example above are called *keys* and the day numbers are called *values*. A map ties together the keys and values. Just like with sets, there are three types of maps: `TreeMap`, `HashMap`, and `LinkedHashMap`. Here are some useful map methods:

Method	Description
`containsKey(k)`	returns whether k is a key in the map
`containsValue(v)`	returns whether v is a value in the map
`get(k)`	returns the value associated with the key k
`put(k,v)`	adds the key-value pair `(k,v)` into the map

To loop over a map, use code like below (assume the keys are strings):

```
for (String key : map.keySet())
    // do something with key or map.get(key)
```

Here are some uses for maps:

- Maps are useful if you have two lists that you need to sync together. For instance, a useful map for a quiz game might have keys that are the quiz questions and values that are the answers to the questions.
- For a Scrabble game where each letter has an associated point value, we could use a map whose keys are the letters and whose values are the points for those letters.
- In implementing a substitution cipher, where we encrypt a message by replacing each letter by a different letter, we could use a map whose keys are the letters of the alphabet and whose values are the letters we encrypt with. For example, here is how we might create such a map:

  ```
  Map<Character, Character> map = new LinkedHashMap<Character, Character>();
  String alpha = "abcdefghijklmnopqrstuvwxyz ";
  String key   = "sapqdemtlrikywvgxnufjhcobz ";

  for (int i=0; i<alpha.length(); i++)
      map.put(alpha.charAt(i), key.charAt(i));
  ```

  For instance, a gets mapped to s and b gets mapped to a. Notice the space at the end of each so that spaces get mapped to spaces. To encrypt a message, we could do the following:

  ```
  String message = "this is a secret";
  String encrypted = "";
  for (int i=0; i<message.length(); i++)
      encrypted += map.get(message.charAt(i));
  ```

- Here is an example that reads all the words from a file and counts how many times each occurs. It uses a map whose keys are the words and whose values are the number of occurrences of those words. Assume for simplicity that all punctuation has already been removed from the file.

  ```
  Map<String, Integer> m = new TreeMap<String, Integer>();
  Scanner textFile = new Scanner(new File("someFile.txt"));
  while (textFile.hasNext())
  {
      String line = textFile.nextLine().toLowerCase();
      for (String word : line.split(" "))
      {
          if (m.containsKey(word))
              m.put(word, m.get(word) + 1);
          else
              m.put(word,  1);
      }
  }
  ```

  The code reads the file one line at a time, breaking up that line into its individual words. Looking at each word, we check if it already has an entry in the map. If so, we add one to its count and otherwise, we add it to the map with a count of 1.

- Here is one more example. Suppose we have a text file called `baseball.txt` that contains stats for all the players in the 2014 MLB season. A typical line of the file look like this:

```
Abreu, B    NYM 12  33  1   14  .248
```

The entries in each line are separated by tabs. Suppose we want to know which team hit the most total home runs. To do this we will create a map whose keys are the team names and whose values are the total home runs that team has hit. To find the total number of home runs, we loop through the file, continually adding to the appropriate map entry, as shown below:

```
Scanner textFile = new Scanner(new File("baseball.txt"));
Map<String, Integer> map = new LinkedHashMap<String, Integer>();
while (textFile.hasNext())
{
    String[] fields = textFile.nextLine().split("\t");
    String team = fields[1];
    int hr = Integer.parseInt(fields[4]);

    if (map.containsKey(team))
        map.put(team, map.get(team) + hr);
    else
        map.put(team, hr);
}
```

Then we can loop through the map we created to find the maximum (it turns out to be Baltimore, with 224 home runs):

```
int best = 0;
String bestTeam = "";
for (String team : map.keySet())
{
    if (map.get(team) > best)
    {
        best = map.get(team);
        bestTeam = team;
    }
}
```

2.17 StringBuilder

If you need to build up a very large string character-by-character, StringBuilders run much faster than ordinary strings. Consider the following code that creates a string of 1,000,000 random capital letters:

```
Random random = new Random();
String s = "";
for (int i=0; i<1000000; i++)
    s += (char)(random.nextInt(26) + 65);
```

It takes literally hours to finish because with each operation inside the loop Java has to create a brand new string and copy over all the characters from the previous version. Below is the equivalent code using Java's `StringBuilder` class. This code takes a fraction of a second to run.

```
Random random = new Random();
StringBuilder sb = new StringBuilder();
for (int i=0; i<1000000; i++)
    sb.append((char)(random.nextInt(26) + 65));
```

The `StringBuilder` class has a `toString` method that converts a StringBuilder object into an ordinary string. The class also has many of the same methods as strings, like `charAt` and `substring`. It also has a useful method, `reverse`, that reverses the string's characters as well as a method, `insert`, for inserting characters into the middle of the string.

2.18 Timing and sleeping

A quick way to see how long it takes for something to run is shown below:

```
double startTime = System.currentTimeMillis();

// code to be timed goes here

System.out.println(System.currentTimeMillis() - startTime);
```

This saves the time, does some stuff, and then computes the difference between the current and saved times. If you print out the value of `System.currentTimeMillis()`, you get something that might seem a little bizarre. For instance, I just ran it and got 1420087321702. That value is how many milliseconds have elapsed since midnight UTC on January 1, 1970.

Another method, `System.nanoTime`, works in a similar way and may be more accurate on some systems.

Sleeping

If you want to pause your program for a short period of time, use `Thread.sleep`. Here is an example:

```
public static void main(String[] args) throws InterruptedException
{
    System.out.println("Hi");
    Thread.sleep(1000);
    System.out.println("Hi again");
}
```

The argument to `Thread.sleep` is number of milliseconds to sleep for. This method requires us to do something if the program is interrupted while sleeping. In this case, we use a `throws` statement to ignore any problems.

2.19 Dates and times

To get a representation of the current date and time use the following:

```
LocalDateTime dt = LocalDateTime.now();
```

If you just want the date without the time or vice-versa, use the following:

```
LocalDate d = LocalDate.now();
LocalTime t = LocalTime.now();
```

All of these will only work on Java version 8 and later. For earlier versions of Java, look into the `Calendar` class.

Specific dates

It is possible to create variables for specific points in time, like below:

```
LocalDate d = LocalDate.of(2014, 12, 25);
LocalTime t = LocalTime.of(9, 12, 30);  // 9:12 am, with 30 seconds
LocalDateTime dt = LocalDateTime.of(2014, 12, 25, 13, 45); // 1:45 pm on Dec. 25, 2014
```

Getting information from the objects

To get specific parts of these objects, there are various get methods, such as those below:

```
System.out.println(dt.getMonth());
System.out.println(dt.getSecond());
System.out.println(dt.getDayOfYear());
```

Formatting things

To nicely format the date and/or time for one of these objects, code like below can be used:

```
System.out.println(dt.format(DateTimeFormatter.ofPattern("EEEE MMMM d, yyyy h:mm a")));
System.out.println(dt.format(DateTimeFormatter.ofPattern("MM/dd/yyyy HH:mm")));
```

The output for these looks like below:

```
Thursday December 25, 2014 1:45 PM
12/25/2014 13:45
```

See the Java API documentation for more about how to use these formatting codes.

Arithmetic with dates

It is possible to do some math with dates. For instance, the following adds 30 days to `d`:

```
d = d.plusDays(30));
```

The code below computes the elapsed time between 12/25/2014 and now:

```
Period period = Period.between(LocalDate.of(2014, 12, 25), LocalDate.now());
```

For computing the elapsed time between `LocalTime` (or `LocalDateTime`) objects, a slightly different approach is needed:

```
Duration duration = Duration.between(LocalTime.of(14, 25), LocalTime.now());
```

A final note on dates

There are a lot of subtleties when working with dates, especially regarding things like time zones, leap years, leap seconds, etc. Read up on these things and be very careful if you are using dates for anything important.

2.20 Threads

Sometimes you need a program to do two things at once. One way to do that is to use *threads*. They are especially useful if you have a long-running task that you want to do in the background without freezing the rest of your program. Here is an example:

```
public static class ClassA extends Thread
{
    public void run()
    {
        for (int i=0; i<1000000000; i++) {}
        System.out.println("A is done");
    }
}

public static class ClassB extends Thread
{
    public void run()
    {
        for (int i=0; i<10000; i++) {}
        System.out.println("B is done");
    }
}

public static void main(String[] args)
{
    ClassA a = new ClassA();
    ClassB b = new ClassB();
```

```
        a.start();
        b.start();
    }
```

When we run the program above, the two threads are running simultaneously and Thread B finishes before Thread A because it has far less work to do. Note that the class syntax might not make a lot of sense until after you've seen some object-oriented programming.

In Java 8 and later, if the code is short enough then we could avoid creating the separate classes and just do the following in the `main` method:

```
Thread a = new Thread(() -> {for (int i=0; i<10000; i++) {} System.out.println("A done");});
Thread b = new Thread(() -> {for (int i=0; i<10000; i++) {} System.out.println("B done");});

a.start();
b.start();
```

This is just to get you started with threads. There is a lot more to them, especially when two threads can both modify the same object. All sorts of tricky things can happen.

When working with GUI-based programs, there is something called `SwingWorker` for using threads.

Chapter 3

Common Programming Techniques

3.1 Counting

An extremely common task in programming is to count how often something happens. The technique usually involves declaring a count variable, setting it equal to 0, and incrementing it each time something happens. Here is some code that will count how many zeros are in an integer list called `list`.

```
int count = 0;
for (int x: list)
{
    if (x==0)
        count++;
}
```

3.2 Summing

Summing up things is useful for things like keeping score in a game or finding an average. We use a similar technique to counting. Here is an example that sums up the entries in an integer array `a`:

```
int sum = 0;
for (int x: a)
    sum += x;
```

The summing technique is a lot like the counting technique. This same idea is used, as mentioned earlier, to build up a string character by character. For example, here is some code that reads through a string `s` and builds up a string `t` from `s`, replacing every fifth character with a space..

```
String t = "";
for (int i=0; i<s.length(); i++)
{
    if (i % 5 == 0)
        t += " ";
    else
        t += s.charAt(i);
}
```

3.3 Maxes and mins

Here is some code to find the maximum value in an integer array `a`.

```
int max = a[0];
for (int x : a)
{
    if (x > max)
        max = x;
```

```
}
```

The basic idea is we have a variable `max` that keeps track of the largest element found so far as we go through the array. Replacing > with < will find the minimum. Note the first line sets `max` equal to the first thing in the array. We could also replace that with any value that is guaranteed to be less or equal to than everything in the array. One such value is `Integer.MIN_VALUE`, a value built into Java.

Sometimes, we might not only need the max, but also where it occurs in the array or some other piece of information. Here is some code that will find the max and the index at which it occurs.

```
int max = a[0];
int index = 0;
for (int i=0; i<a.length; i++)
{
    if (a[i] > max)
    {
        max = a[i];
        index = i;
    }
}
```

An alternative would be to just store the index and compare `a[i]` to `a[index]`.

3.4 The modulo operator

Here are several common uses for the modulo operator:

Checking divisibility

To see if an integer `n` is divisible by an integer `d`, check if `n` modulo `d` is 0. For example, to check if an integer `x` is even, use the following if statement:

```
if (x % 2 == 0)
```

Scheduling things to occur every second, every third, etc. time through a loop

Building off the previous idea, if we want something to occur every second time through a loop, mods come in handy. For example, the program below alternates printing `Hello` and `Goodbye`:

```
for (int i=0; i<10; i++)
{
    if (i % 2 == 0)
        System.out.println("Hello");
    else
        System.out.println("Goodbye");
}
```

Modding by 3 would allow us to schedule something to happen on every third iteration, modding by 4 would be for every fourth iteration, etc.

Wrapping around

Mods are useful for when you have to wrap around from the end of something to the beginning. Here are a couple of examples to demonstrate this.

Say we have an array `a` of 365 sales amounts, one for each day of the year, and we want to see which 30-day period has the largest increase in sales. It could be any 30-day period, including, say, December 15 to January 14. We can compute the differences of the form `a[i+30]-a[i]`, but that won't work for 30-day periods from December to January (in fact, we will probably get an index out of bounds exception). The trick is to use a mod, like below:

```
int difference = a[(i+30) % 365] - a[i];
```

Here is another example. A simple way to animate a character moving in a graphical program is to alternate between two or more images. For instance, a simple Pacman animation would alternate between an image with Pacman's mouth open and one with it closed. We could do this by keeping a counter, and whenever the count mod 2 is 0, we display one image and otherwise we display the other. Or if we had 3 images of Pacman, then we would show a different image based on whether the count is 0, 1, or 2 mod 3.

Conversions

Mods are useful for certain types of conversions. For instance, if we have a time in seconds, and we want to convert it to minutes and seconds (as in 130 seconds going to 2 minutes, 10 second), we can use the following:

```
int minutes = time / 60;
int seconds = time % 60;
```

Similarly, to convert a length in inches to inches and feet (for instance, 40 inches to 3 feet, 4 inches), we can do the following:

```
int feet = length / 12;
int inches = length % 12;
```

In a similar way, we can convert between two ways of representing a grid. We can use a one-dimensional array (or list) or a two-dimensional array (or list of lists). The indices of the two approaches are shown in the table below.

(0,0)	**0**	(0,1)	**1**	(0,2)	**2**	(0,3)	**3**	(0,4)	**4**	(0,5)	**5**	(0,6)	**6**
(1,0)	**7**	(1,1)	**8**	(1,2)	**9**	(1,3)	**10**	(1,4)	**11**	(1,5)	**12**	(1,6)	**13**
(2,0)	**14**	(2,1)	**15**	(2,2)	**16**	(2,3)	**17**	(2,4)	**18**	(2,5)	**19**	(2,6)	**20**
(3,0)	**21**	(3,1)	**22**	(3,2)	**23**	(3,3)	**24**	(3,4)	**25**	(3,5)	**26**	(3,6)	**27**

If our one-dimensional array is `a` and our two-dimensional array is `b`, to go from `b[i][j]` to `a[k]` we use `k = 6*i + j` and to go from `a[k]` to `b[i][j]` use `i = k / 7` and `j = k % 7`. These formulas are occasionally useful.

3.5 Nested loops

Consider the following code:

```
for (int i=1; i<=3; i++)
{
    for (int j=1; j<=3; j++)
        System.out.print("(" + i + ", " + j + ") ");
}
```

This code will print out all pairs `(i,j)` where `i` and `j` are between 1 and 3. The idea is that the outside loop runs over all possible values of `i` from 1 to 3. For each of those values, we loop through all the possible `j` values. The output is shown below.

```
(1,1) (1,2) (1,3) (2,1) (2,2) (2,3) (3,1) (3,2) (3,3)
```

These types of loops are useful for processing two-dimensional arrays and pixels on a screen.

Here is another example: Say we need to find all the solutions (x, y, z) of the equation $x^2 + y^2 = z^2$, where x, y and z are all between 1 and 99. One approach to this problem is to generate all possible triples (x, y, z) and check to see if they satisfy the equation. To generate the triples, we use three nested loops, like below:

```
for (int x=1; x<100; x++)
{
    for (int y=1; y<100; y++)
    {
        for (int z=1; z<100; z++)
```

```
        {
            if (x*x + y*y == z*z)
                System.out.println(x + " " + y + " " + z);
        }
    }
}
```

Here is another important example:

```
for (int i=1; i<=4; i++)
{
    for (int j=1; j<=i; j++)
        System.out.print("(" + i + "," + j + ") ");
    System.out.println();
}
```

Notice how the inner loop refers to the outer loop's variable. This allows the inner loop to vary in how many times it runs. The output is below:

```
(1,1)
(2,1) (2,2)
(3,1) (3,2) (3,3)
(4,1) (4,2) (4,3) (4,4)
```

Loops like this are useful in a variety of contexts.

3.6 Flag variables

Flag variables are variables that are used to let one part of your program know that something has happened in another part of your program. They are usually booleans. Here is an example of a program that uses a flag variable to check if a number n is prime.

```
boolean isPrime = true;
for (int d=2; d<n; d++)
{
    if (n % d == 0)
        isPrime = false;
}

if (isPrime)
    System.out.println("Prime.");
else
    System.out.println("Not prime.");
```

A number is prime if its only divisors are itself and 1. If it has any other divisors, then it is not prime. The program above sets the flag variable `isPrime` to `false` if any such divisor is found. (Note that the above program is not a particularly efficient way to find primes. A better prime checker would have a loop that runs to $\sqrt{n}$ and would stop checking once a divisor was found.)

Note also that the if statement used after the loop uses a bit of a shorthand. The conditions on the left below are equivalent to the ones on the right.

```
if (isPrime)        if (isPrime == true)
if (!isPrime)       if (isPrime == false)
```

Flags are especially useful in longer programs. For instance, a game might have a flag variable called `gameOver` that is initially set to `false` and gets set to `true` when something happens to end the game. That variable would then be used to prevent some code from being executed that should only be executed if the game was still going on. Or a program might have a flag called `useColor` that determines whether or not to print something in color or in black and white.

3.7 Working with variables

Swapping

Here is the code to swap the values of integer variables x and y:

```
int hold = x;
x = y;
y = x;
```

Keeping track of previous values

Suppose we are asking the user to enter 10 names and we want to notify them any time they type the same name in twice in a row. Here is code to do that:

```
Scanner scanner = new Scanner(System.in);

System.out.println("Enter a name: ");
String previous = scanner.nextLine();

for (int i=0; i<9; i++)
{
    System.out.print("Enter a name: ");
    String current = scanner.nextLine();
    if (current.equals(previous))
        System.out.println("You just entered that name.");
    previous = current;
}
```

The way we accomplish this is by using variables to hold both the current and previous names entered. In the loop we compare them and then set `previous` to `current` so that the current entry becomes the previous entry for the next step.

This technique is useful in a variety of contexts.

3.8 Using lists to shorten code

Here are a couple of examples of how lists (or arrays) can be used to shorten code. Suppose we have a long sequence of if statements like the one below:

```
if (month == 1)
    monthName = "January";
else if (month == 2)
    monthName = "February";
...
else
    monthName = "December";
```

If we use a list of month names, we can replace all of the above with the following:

```
List<String> names = new ArrayList<String>();
Collections.addAll(names, "January", "February", "March", "April", "May", "June",
                          "July", "August", "September", "October", "November", "December");
monthName = names.get(month - 1);
```

If we needed to go backwards from month names to month numbers, we could also use the above list as below:

```
monthNumber = names.indexOf(monthName) + 1;
```

Here is another example. The code below plays a simple quiz game.

```
int numRight = 0;
String guess;
Scanner scanner = new Scanner(System.in);

// Question 1
```

```
System.out.println("What is the capital of France?");
guess = scanner.nextLine();

if (guess.equalsIgnoreCase("Paris"))
{
    System.out.println("Correct!");
    numRight++;
}
else
    System.out.println("Wrong.");
System.out.println("Score: " + numRight);

// Question 2
System.out.println("Which state has only one neighbor?");
guess = scanner.nextLine();

if (guess.equalsIgnoreCase("Maine"))
{
    System.out.println("Correct!");
    numRight++;
}
else
    System.out.println("Wrong.");
System.out.println("Score: " + numRight);
```

We could add more questions by copying and pasting the code and just changing the questions and answers in each pasted block. The problem with this approach is that the code will become very long and difficult to maintain. If we decide to change one of the messages we print or to add a new feature, then we have to edit each individual block. Moreover, it would be nice to have the questions and answers all in the same place. One solution is to use lists, like below:

```
List<String> questions = new ArrayList<String>();
List<String> answers = new ArrayList<String>();

questions.add("What is the capital of France?");
answers.add("Paris");
questions.add("Which state has only one neighbor?");
answers.add("Maine");

Scanner scanner = new Scanner(System.in);

for (int i=0; i<questions.size(); questions++)
{
    System.out.println(questions.get(i));
    String guess = scanner.nextLine();

    if (guess.equalsIgnoreCase(answers.get(i)))
    {
        System.out.println("Correct!");
        numRight++;
    }
    else
        System.out.println("Wrong.");
    System.out.println("Score: " = numRight);
}
```

If you compare this code with the copy/paste code earlier, you'll see that the places that we had to change in the copy/paste code were replaced with list stuff. Note that an even better approach would be to store the questions and answers in a file instead of hard-coding them.

In general, if you find yourself working with a long string of if statements or other repetitive code, try using lists to shorten things. Not only will it make your code shorter, but it may also make your code easier to change.

Chapter 4

Object-Oriented Programming

4.1 Introduction

When you write a program of 10 or 20 lines, there's usually not too much of a need to worry how your code is organized. But when you write a program of several thousand lines or work on a large project with a team of people, organization becomes vitally important. One of the key ideas is *encapsulation*, which is basically about breaking your project into lots of little pieces, each of which may communicate with the others but doesn't need to know exactly how they do what they do.

For instance, a program to play a card game might break things into classes with one class that just handles shuffling and dealing cards, another class that determines what type of hand something is (straight, full house, etc.), and a third class that displays everything. Each class doesn't need to know the internals of the others; it just needs to know how to communicate with the other classes.

The benefit of this is that you can write your program in small chunks. You can write each one, test it, and move on to the next. If everything is together in one big blob of code, it quickly becomes hard to maintain. If you leave the code and come back several months later, it will be hard to remember what part does what. Also, it will be hard to keep track of how changing a variable in one part of the code will affect another part of the code.

We will start by talking about functions, and then move on to classes.

4.2 Functions

Here is an example of a complete program with a function called `getRandomChar`.

```
public class FunctionExample
{
    public static void main(String[] args)
    {
        System.out.println(getRandomChar("abcdef"));
    }

    public static char getRandomChar(String s)
    {
        Random random = new Random();
        return s.charAt(random.nextInt(s.length()));
    }
}
```

The function called `getRandomChar` takes a string and returns a random letter from that string. Getting a random character from a string is something that is occasionally useful and something we might want to do at multiple points in our program. So we put it into a function.

We can see above the basic mechanics of creating functions. First, we have the `public static` part. We will explain what those words mean once we get to classes. Generally, if your entire program is contained in one

class, you'll usually declare your functions as `public static`.

The next part of the function declaration is `char`. This specifies what type of data the function returns, in our case a `char`. We could have any other data type here or `void` if the function doesn't return anything.

Inside the parentheses is the function's *parameter*. It represents the data given to the function. In this case, the person calling the function gives it a string. We give this parameter the name `s` so that we can refer to it in the body of the function.

In the body of the function goes the code that chooses a random character from the caller's string. The last line of the function returns the random character back to the caller. In general, the `return` statement is used to return a value back to the caller.

So to summarize: In the main part of the program, we *call* the function by doing `getRandomChar("abcdef")`. This passes the string `"abcdef"` to the function. This value is referred to by the parameter `s`. The program then shifts to running the code in the function, choosing a random character from the parameter `s` and returning that to the main part of the program, where it is printed out.

4.3 Examples of functions

1. Here is a function that counts how many times a character appears in a string:

```
public static int count(String s, char c)
{
    int count = 0;
    for (int i=0; i<s.length(); i++)
        if (s.charAt(i) == c)
            count++;
    return count;
}
```

   Here are a couple of ways that we could call this function (from inside of `main` or possibly inside of some other function):

```
System.out.println("The string has " + count(s, ' ') + " spaces.");
int punctuationCount = count(s, '!') + count(s, '.') + count(s, '?');
```

2. Here is a function that prints a specified number of blank lines:

```
public static void printBlankLines(int n)
{
    for (int i=0; i<n; i++)
        System.out.println();
}
```

   Notice that it is a `void` function, so it returns no values. We could call this function like below (from `main` or wherever) to print out 3 blank lines:

```
printBlankLines(3);
```

3. Here is a function that returns a random capital letter:

```
public static char randomCapital()
{
    Random random = new Random();
    return (char) random.nextInt(26) + 65;
}
```

   Here a some ways to call this function:

```
char c = randomCapital();
System.out.println("A random letter: " + randomCapital());
```

4.4 More details about functions

Declarations

The first line of a function, where we specify its name, return type, and parameters is called its *declaration*. Here are some possible function declarations:

```
char myFunction(String s)       // takes a string, returns a char
void myFunction(int x)          // takes an int, returns nothing
List<String> myFunction()       // takes nothing, returns a list of strings
int myFunction(int x, double y) // takes an int and a double, returns an int
```

Functions in Java can have multiple parameters, but they can only return one value, though it is possible to use arrays, lists, or classes to achieve the effect of returning multiple values.

Return statements

When a return statement is reached, the program jumps out of the function and returns back to where it was called from, usually returning with a value.

It is possible for a function to have multiple return statements, like below:

```
public static boolean containsEvens(List<Integer> list)
{
    for (int x : list)
    {
        if (x % 2 == 0)
            return true;
    }
    return false;
}
```

This function checks whether a list contains any even integers. It loops over the list, and as soon as it finds an even, it returns `true`, jumping immediately out of the function without bothering to check the rest of the list. If we get all the way through the list without finding any evens, then the function returns `false`.

Void functions don't require a return statement, though an empty `return` statement in a void function can be used to jump out of the function. Here is an example:

```
public static void printTwoRandomItems(List<String> list)
{
    if (list.size() < 2)
        return;

    List<String> copy = new ArrayList<String>(list);
    Collections.shuffle(copy)
    System.out.println(copy.get(0) + ", " + copy.get(1));
}
```

Local variables

A variable declared in a function is *local* to that function. This means that it essentially only exists within that function and is not accessible from outside of it. This is important as in large programs, you don't want the variables of one function to affect the variables of another function. You also don't have to worry about if you've already used a variable with the same name in another function.

Parameters vs arguments

Just a quick vocabulary note: Suppose we have a function with the declaration `void f(int x, int y)`.

The variables `x` and `y` are the function's parameters. When we call the function with something like `f(3, 6)`, the values 3 and 6 are called *arguments* to the function.

4.5 Classes

A class is a structure that holds both data and functions. The data are called *fields*, *attributes*, or *class variables*. The functions are referred to as *methods*. An example is Java's `String` class. The data it holds are the characters of the string and the methods are things that do something with the characters of the string, like `length()` or `substring()`. Another example is Java's `Scanner` class. The data is the string or file that is being scanned and the methods are things like `nextLine()` or `hasNext()` that do something with that data.

Below are a few examples of classes.

1. The following class could be used as part of a quiz game. The class represents a quiz question. It stores a question and an answer together and has methods that allow us to read what the question and answer are, as well as a method we can use to test if a guess at the answer is correct.

```
public class QuizProblem
{
    private String question;
    private String answer;

    public QuizProblem(String question, String answer)
    {
        this.question = question;
        this.answer = answer;
    }

    public String getQuestion()
    {
        return question;
    }

    public String getAnswer()
    {
        return answer;
    }

    public boolean isCorrect(String guess)
    {
        if (guess.equalsIgnoreCase(answer))
            return true;
        else
            return false;
    }
}
```

   Reading through this class, the first thing we see are the two class variables, `question` and `answer`. They are declared as `private`, which means that only the class itself can read them or change them. We could also have declared them as `public`, which would mean any other Java class could read and change them.

   In Java, people usually declare class variables as `private` and provide special methods, called *getters* and *setters*, that other classes can use to read and change the variables. Our class has getters called `getQuestion()` and `getAnswer()`, but no setters. So we will allow other classes to read our variables but not change them.

   Right before the two getters is a special method called the *constructor*. It is used to create new `QuizProblem` objects. When someone wants to create a new `QuizProblem` object (in another file usually), they specify the question and answer and call the constructor using something like the line below:

```
QuizProblem problem = new QuizProblem("What is the capital of France?", "Paris");
```

   This line creates a new *object* or *instance* of the class. The class acts as a general template and objects are things that follow that template, with specific values for the class's fields. The constructor takes the two values that the caller passes in and stores them in the class variables.

   Notice the keyword `this` used inside the constructor. That keyword is used to refer to the class variables `question` and `answer` to distinguish them from the parameters of the same name. Also, note that the constructor must have the exact same name as the class and has no return type (not even `void`).

   The last part of this class is a method called `isCorrect`. The caller passes in a guess and the code determines if that guess is equal to the answer we are storing in the class.

Below is some code that creates a new `QuizProblem` object and uses it. This code is contained in a class different from the `QuizProblem` class.

```
public class Quiz
{
    public static void main(String[] args)
    {
        Scanner scanner = new Scanner(System.in);
        QuizProblem prob = new QuizProblem("What is the capital of France?", "Paris");
        System.out.println(prob.getQuestion());
        System.out.print("Answer: ");
        String guess = Scanner.nextLine();
        if (prob.isCorrect(guess))
            System.out.println("Right!");
        else
            System.out.println("Wrong. Answer was " + prob.getAnswer());
    }
}
```

In the code above, we create a new `QuizProblem` object, print out the question part of it, ask the program's user for the answer, and then use the `QuizProblem` class's `isCorrect` method to check the answer.

All of this is a bit much for a program that asks the user what the capital of France is. However, we could use this `QuizProblem` class in a variety of different programs without having to change anything. For instance, suppose we have a bunch of questions and answers stored in a file called `quiz.txt`, say with question 1 on the first line followed by its answer on the next, with question 2 following that, etc. The code below could be used to read all of the questions into a list.

```
List<QuizProblem> problems = new ArrayList<QuizProblem>();
Scanner textFile = new Scanner(new File("quiz.txt"));

while (textFile.hasNext())
{
    String q = textFile.nextLine();
    String a = textFile.nextLine();
    problems.add (new QuizProblem(q, a));
}
```

We could then use code like the code below to run a quiz game that goes through all the questions.

```
Scanner scanner = new Scanner(System.in);
for (QuizProblem prob : problems)
{
    System.out.println(prob.getQuestion());
    System.out.print("Answer: ");
    String guess = scanner.nextLine();
    if (prob.isCorrect(guess))
        System.out.println("Right!");
    else
        System.out.println("Wrong. Answer was " + prob.getAnswer());
}
```

One benefit of using the object here is our program just has one list of problems as opposed to separate lists of questions and answers that we would have to keep in sync. Another benefit is it separates our program into a couple of different pieces, making it easier to keep track of what does what and make changes. For instance, if we wanted to improve the answer checking so that it could handle common misspellings or something like that, we would just go directly to the `isCorrect` method.

2. Here is a class that represents a playing card.

```
public class Card
{
    private String suit;
    private int value;

    public Card(int value, String suit)
    {
        this.value = value;
        this.suit = suit;
    }

    public int getValue()
```

```
        {
            return value;
        }

        public String getSuit()
        {
            return suit;
        }

        public void setValue(int value)
        {
            this.value = value;
        }

        public void setSuit(String suit)
        {
            this.suit = suit;
        }

        @Override
        public String toString()
        {
            String s = value + " of " + suit;
            s = s.replace("11", "jack");
            s = s.replace("12", "queen");
            s = s.replace("13", "king");
            s = s.replace("14", "ace");
            return s;
        }
    }
```

A playing card has a suit and a value, like the 3 of clubs or 10 of hearts. Thus our class has a string field for the suit and an integer field for the value. For simplicity, we use numbers for the values, so that a value of 11 would be a jack, a value of 12 would be a queen, etc.

The first method we have in the class is the constructor. As noted in the previous example, the constructor has the same name as the class and no return type. Our constructor just sets the values of the two fields. For instance, suppose in another class, someone creates a card with the following line:

```
Card card1 = new Card(3, "clubs");
```

This line calls the constructor, and the constructor sets the `value` field to 3 and it sets the `suit` field to `"clubs"`.

After the constructor, we have getters and setters for the two fields. These allow people using our class to read and modify the two fields. For instance, assuming someone has created a `Card` object (like in the line of code above), then they could read and modify the fields like below:

```
System.out.println(card1.getValue());
card1.setSuit("Hearts");
```

The first line calls the getter to read the `value` field from `card1`, and the second line calls the setter to change the card's suit. Getters and setters are optional. If you don't want people to be able to read or modify certain fields, then don't include getters and setters in the class.

The last method is a special one called `toString`. It returns a string containing a nicely formatted version of the card. The `toString` method is specifically used by Java whenever you try to print an object. For instance, the following two lines will print out `"king of spades"`:

```
Card c = new Card(13, "spades");
System.out.println(c);
```

You might wonder why make a class for a playing card? We could represent a card purely with a string, maybe with strings like `"3,clubs"`. But then we'd have to do some string manipulation to get at the suit and value. That leads to less readable code. The object-oriented approach uses `card.getValue()`, whereas the string approach would use `card.substring(0,card.index(","))`.

Also, the class gives us a nice place to store card-related code. For instance, the `toString` method above contains code for nicely printing out a card. It is contained in the same class (and same file) as other card code. We could also add more code to this class to allow us to do other things, like compare cards based on their values or another field that would allow us to treat aces differently from other cards. Since all the card code is in one class, whenever there is a problem with card code or some new card feature we need

to add, we know where to go. Further, this card class could be used as part of a variety of different games without having to copy/paste or rewrite anything. All we'd have to do is import the `Card` class or put a copy of the file in the same package as our game.

3. Here is a class that represents a player in a simple game, where players attack each other or monsters.

```
public class Player
{
    private String name;
    private int hitPoints;
    private int attackPower;

    public Player(String name, int hitPoints, int attackPower)
    {
        this.name = name;
        this.hitPoints = hitPoints;
        this.attackPower = attackPower;
    }

    public String getName()
    {
        return name;
    }

    public int attack()
    {
        Random random = new Random();
        return random.nextInt(attackPower);
    }

    public void takeDamage(int amount)
    {
        hitPoints -= amount;
    }

    public boolean isAlive()
    {
        return hitPoints > 0;
    }
}
```

We have class variables for the player's name, hit points, and attack power. There is a constructor that sets those values, a getter for the name, and no setters. There is a method called `attack` that returns a random attack amount based on the player's attack power. There is a method called `takeDamage` that is given an amount and adjusts the player's hit points down by that amount. Finally, there is a method called `isAlive` that returns whether or not the player is still alive (`true` if the player's hit points are positive and false otherwise). Here is how we might use this in another class:

```
Player player1 = new Player("Ogre", 20, 10);
Player player2 = new Player("Fighter", 30, 7);

while (player1.isAlive() && player2.isAlive())
{
    int attack = player1.attack();
    player2.takeDamage(attack);
    System.out.println(player1.getName() + " deals " + attack + " damage.");

    attack = player2.attack();
    player1.takeDamage(attack);
    System.out.println(player2.getName() + " deals " + attack + " damage.");
}
if (player1.isAlive() && !player2.isAlive())
    System.out.println(player1.getName() + " wins.");
else if (player2.isAlive() && !player1.isAlive())
    System.out.println(player2.getName() + " wins.");
else
    System.out.println("Nobody wins!");
```

In the interest of keeping the code short, we have made a bit of a boring game, but you could imagine this as being the start of a more interesting game.

4. Here is a `Deck` class that represents a deck of playing cards. It uses the `Card` class.

```
public class Deck
{
    private List<Card> cards;

    public Deck()
    {
        cards = new ArrayList<Card>();
        String[] suits = {"diamonds", "hearts", "clubs", "spades"};
        for (int i=0; i<4; i++)
            for (int j=2; j<=14; j++)
                cards.add(new Card(j, suits[i]));
    }

    public void shuffle()
    {
        Collections.shuffle(cards);
    }

    public Card getNext()
    {
        Card card = cards.get(0);
        cards.remove(0);
        return card;
    }

    public boolean hasNext()
    {
        return !cards.isEmpty();
    }
}
```

The main part of this class is a list of `Card` objects that represents the deck. The constructor fills up that deck using nested for loops to put all 52 standard cards into the deck. Notice how we declare the list as a field, and in the constructor we do `deck = new ArrayList<Card>()` to initialize that list. Forgetting to do this will lead to a null pointer exception.

The `shuffle` method does exactly what it says by using `Collections.shuffle` on the list. The `getNext` method is perhaps the most interesting. It deals the next card out from the top of the deck and removes that card from the deck. The `hasNext` method returns `true` or `false` based on whether there is anything left in the deck. Here is how we might use the class to play a simple hi-lo card game:

```
Scanner scanner = new Scanner(System.in);

Deck deck = new Deck();
deck.shuffle();

Card oldCard = deck.getNext();
while (deck.hasNext())
{
    Card newCard = deck.getNext();
    System.out.println(oldCard);
    System.out.print("Higher or lower? ");
    String guess = scanner.nextLine().toLowerCase();

    if (guess.equals("lower") && newCard.getValue() < oldCard.getValue())
        System.out.println("Right");
    else if (guess.equals("higher") && newCard.getValue() > oldCard.getValue())
        System.out.println("Right");
    else
        System.out.println("Wrong"); //

    System.out.println();
    oldCard = newCard;
}
```

4.6 Object-oriented concepts

Constructors

A constructor is a special method that is called whenever someone creates a new object from a class. For instance, here is the constructor for the `Player` class above.

```
public Player(String name, int hitPoints, int attackPower)
{
    this.name = name;
    this.hitPoints = hitPoints;
    this.attackPower = attackPower;
}
```

When the following line is run from another class, the `Player` constructor is called.

```
Player player1 = new Player("Ogre", 20, 10);
```

The constructor's job is to set up the object and get it ready for whatever the class needs. Often that involves setting the class variables to values passed by the caller, but it could involve other things.

Just a note about terminology. A *class* is some Java code where we define some fields and methods. An *object* is a specific instance of that class. The class acts as a template for objects. For instance, `Player` is the class, while `player1` is an object. The process of creating an object (using the `new` keyword) is called *instantiating*.

Multiple constructors

There can be more than one constructor for a class, so long as each has a different list of parameters. For instance, suppose we have a `Ticket` class with `price` and `time` fields. Below we create two constructors: one that takes both fields as arguments, and another that takes just the time and sets a default price.

```
public Ticket(String price, String time)
{
    this.price = price;
    this.time = time;
}

public Ticket(String time)
{
    this.price = 8.50;
    this.time = time;
}
```

In fact, this sort of thing works for any functions in Java, not just constructors. For instance, we could create a function called `getRandom(String s)` that returns a random character from a string and we could create another function with the same name, `getRandom(List<Integer> list)` that returns a random item from a list.

Creating lists of objects

We can have lists of objects, like below, where we create an army of ogres:

```
List<Player> army = new ArrayList<Player>();
for (int i=0; i<100; i++)
    army.add(new Player("Ogre", 20, 10));
```

Getters and setters

Getters and setters (sometimes called *accessors* and *mutators*) are methods used by others to read and modify a class's fields. The `Card` class above provides an example. One of the class's fields is its suit (a string). Here are the getters and setters for that field:

```
public String getSuit()
{
```

```
    return suit;
}

public void setSuit(String suit)
{
    this.suit = suit;
}
```

Getters and setters will usually follow this form, though they can be more involved. For instance, the setter above might check to make sure the suit parameter is valid (hearts, clubs, spades, or diamonds), or it might convert the suit parameter to lowercase. However, if you just have a basic getter or setter to make, your IDE will be able to automatically generate it for you. In Eclipse and Netbeans, under the Source menu there is an option for generating getters and setters. It's under the Code menu in IntelliJ.

Getters and setters are not required. They are only needed if you want other classes to be able to read or modify your class's fields. If your fields are public instead of private (see later in this chapter for more on that), then your class's fields are automatically able to be read and modified without getters and setters, though this is usually frowned upon in Java.

toString

The `toString` method is a special Java method that works in conjunction with print statements to print out a representation of a class. Suppose we have a class called `Ticket` that represents a movie ticket and suppose the class has fields `price` and `time`. Suppose now, we create a new Ticket object called `ticket` and try to print it out as below:

```
System.out.println(ticket);
```

Java will give us something unhelpful like `Ticket@a839ff04`. However, if we add the following method to our class, things will print out more nicely:

```
@Override
public String toString()
{
    return "Ticket: price=$" + price + ", time=" + time;
}
```

Calling `System.out.println(ticket)` on a `Ticket` object called `ticket` with a time of 2:00 and price of $6.25 will print out the following:

```
Ticket: price=$6.25, time=2:00
```

In general, the `toString` method returns a string that is used by `print` statements to print a representation of an object. In order for it to work, it must be precisely named `toString` and must have a `String` return type and no arguments. The `@Override` part will be explained later.

equals

Like `toString`, `equals` is another special Java method. It is used to compare two objects for equality. To understand where it is used, suppose we have a class called `Coordinate` that has fields called `x` and `y`. The class represents coordinates, like (1,2) or (8,5). Suppose we have two `Coordinate` objects `c1` and `c2` and we want to test if they are equal. We might try the following:

```
if (c1 == c2)
```

This fails for the same reason that we can't compare two strings using ==. Java actually compares the objects based on whether they occupy the same memory location, not based on the values stored in the objects. The correct way to compare them is shown below:

```
if (c1.equals(c2))
```

But even this won't work automatically. We need to tell Java how to compare objects from our class. To do this, we have to write the code for the `equals` method. While, we can write it ourselves, it is much preferable to have

an IDE write it for us. Under the Source menu in Eclipse and Netbeans and the Code menu in IntelliJ is the option to generate `hashCode` and `equals`. The `hashCode` method is another special Java method whose exact function we won't worry about here, but it is recommended to be written whenever `equals` is written. Here is the code that Eclipse generates:

```
@Override
public int hashCode()
{
    final int prime = 31;
    int result = 1;
    result = prime * result + x;
    result = prime * result + y;
    return result;
}

@Override
public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Tuple other = (Tuple) obj;
    if (x != other.x)
        return false;
    if (y != other.y)
        return false;
    return true;
}
```

The key part of the code is the last five lines of the `equals` method. It compares the `Coordinate` objects based on their `x` and `y` coordinates. Notice also that the method must be precisely named `equals`, with a boolean return type and a single parameter of type `Object`. As mentioned, to get this right, it's easiest to have your IDE generate the code for you. You can choose which fields to use for the comparison. Not all fields have to be used. For instance, suppose we have a class representing a student, with fields for ID number, name, GPA, and classes. If we just want to compare students based on ID numbers, we could do that.

The `equals` method is used for comparing things in if statements, and it is also used by some Java list methods like `contains` and `remove` that call the class's `equals` method when searching a list for a specific item.

`main`

The main method is the special method that allows your class to actually run. Not all classes will need it, but in any program that you write, you need at least one class to have a main method so that your program will run. You can add main methods to other classes in your program and use them to test out the individual classes.

Here is a typical main declaration:

```
public static void main(String[] args)
```

The array `String[] args` is for command-line arguments. If you run the program from the command line, any values you pass in to the program are stored in that array.

Packages

Often you will be writing a program that consists of several related classes. A good idea is to group them into a *package* to keep them separate from other, unrelated programs. One example of a package you already know is `java.util`. It contains a variety of useful classes like `Random` and `ArrayList`. Another package is `java.io` that contains classes that are useful for working with files and other things.

To create a package in Eclipse, just right click on a project name and select new package to start a package in that project. Then right click the package name to create a new class in it. Other IDEs work similarly. Note that

package names, by convention, start with lowercase letters.

If a file is part of a package, then the first line of the file should be a line like below (your IDE will probably automatically do this for you):

```
package yourPackageName;
```

Public vs private

When declaring variables and methods in a class, you can determine who will have access to them. The options are `public`, `private`, `protected`, or nothing. The Java documentation has the following useful table indicating who has access to what:

	Class	Package	Subclass	World
public	x	x	x	x
protected	x	x	x	
no modifier	x	x		
private	x			

The most useful ones are `public`, which means the variable or method is visible to everyone, and `private`, which means it is visible to no one except the class itself. We will talk about `protected` later. Generally, class variables are made private unless there is a good reason not to.

To demonstrate the difference between public and private variables, suppose we have a class called `SimpleClass`, a `SimpleClass` object called `sc`, and suppose the class has a field called `x`. If that field is private, then other classes could read or modify `x` through getters and setters that we'd have to write. However, suppose `x` is declared public. Then another class can use the following to look at and change `x`:

```
System.out.println(sc.x);
sc.x = 23;
```

We can see that the getter/setter way of doing things is more verbose. For instance, the way to get the value of a public variable `x` is just to do `sc.x`, whereas if it is private, we would have to call `sc.getX()`. The extra characters can get tedious to constantly type, and it is tempting to just make everything public. However, the Java way of doing things is to stick with private variables. The Java community prefers the safety of using private variables to the ease of using public variables. In fact, some people recommend not even having getters and setters at all.

Methods, on the other hand, are usually public if you want other people to be able to use them (which is often). However, sometimes you'll need a method that does something internally to a class that the outside world doesn't need to know about, and in that case you'll make the method private.

Static methods

A static method is a method that is part of a class, but doesn't need a specific object to be created in order to be used. A good example is the `Math` class built into Java. If we want to print cos(0), we can do the following:

```
System.out.println(Math.cos(0));
```

Notice that even though `cos` is part of the `Math` class, we don't have to create a new `Math` object using something like `Math math = new Math()` in order to use `cos`. We can just use it. This is because it was declared as static in the `Math` class. Its declaration looks something like this:

```
public static double cos(double x)
```

On the other hand, the `size` method of a list is not a static method. The reason for this is that `size` does not make sense without an actual list to find the size of. We need to say something like `list.size()` to get the size of the list. It wouldn't make sense to call `size` by itself in the same way that we call `Math.cos` by itself.

Here are some examples of calling static functions that we have seen before:

```
Collections.addAll(list, 1, 2);
Collections.shuffle(list);
Thread.sleep(2);
Integer.parseInt("25");
```

Notice that the call is preceded by the name of the class itself (note the capital). On the other hand, here are some examples of calling non-static functions:

```
list.get(0);
s.length());
textFile.nextLine();
```

These methods need specific objects to work with. For instance, `list.get(0)` gets the first thing in a list, and it needs a specific list to get that thing from. It wouldn't make sense to say `ArrayList.get(0)`.

Here is an example of how we can write static methods.

```
public class MyClass
{
    private String str;

    public MyClass(String str)
    {
        this.str = str;
    }

    public static char getRandomChar1(String s)
    {
        Random random = new Random();
        return s.charAt(random.nextInt(s.length()));
    }

    public char getRandomChar2()
    {
        Random random = new Random();
        return str.charAt(random.nextInt(s.length()));
    }
}
```

The first random character method is static, while the second isn't. Here is how the first method would be used. Notice that we call it by using the class name followed by the method name.

```
char c = MyClass.getRandomChar1("abcdef");
```

Here is how the second method would be called. Since it is not static, we need a specific object to work with.

```
MyClass mc = new MyClass("abcdef");
char c = mc.getRandomChar2();
```

Ordinary methods usually operate on the data of a class, whereas static methods are often useful standalone utilities or methods that a class needs that don't rely on the data of the class. Here is an example of a class with a few static methods useful for working with random numbers:

```
public class RandomUtilities
{
    // returns a random integer from a to b (inclusive)
    public static int randint(int a, int b)
    {
        Random random = new Random();
        return random.nextInt(b-a+1) + a;
    }

    // returns a random even number from a to b (inclusive)
    public static int randomEven(int a, int b)
    {
        return 2 * randint(Math.ceil(a/2), b/2)
    }
}
```

Notice that these methods don't rely on any class variables. To call the first method from another class, we would use something like `RandomUtilities.randint(1,10)`.

Static variables

You can also declare a variable static, though you'll usually want to avoid doing so. When you declare a variable static, every instance of your class will share the same copy of that variable. Consider the following class that has a static variable x:

```
public class StaticVariableExample
{
    public static int x;

    public StaticVariableExample(int x)
    {
        this.x = x;
    }

    public String toString()
    {
        return "x = " + x;
    }
}
```

Suppose we test the class as follows:

```
StaticVariableExample e1 = new StaticVariableExample(1);
System.out.println(e1);

StaticVariableExample e2 = new StaticVariableExample(100);
System.out.println(e2);

System.out.println(e1);
```

Here is the output:

```
x = 1
x = 100
x = 100
```

We create two objects, `e1` and `e2`. Creating `e2` changes what happens when we print out `e1`. This is because all objects from our class share the same copy of `x`. Sometimes, this is what you want, but usually not. It can be a source of hard to find bugs if you are not careful. Sometimes a Java warning or error message will seem to indicate that you should make a variable static, but that's usually not the solution you want.

Constants and the `final` keyword

If you want to create a constant in your program, use the `final` keyword. That keyword makes it impossible to modify the value of the variable. Here is an example.

```
public static final int MAX_ARRAY_SIZE = 4096;
```

Here `static` is appropriate since constants are usually the same for all instances of a class. By convention, constants are usually written in all caps.

In general, declaring a variable `final`, whether it is a class variable, a local variable, or whatever, means that the variable cannot be reassigned to. For instance, the following is an error:

```
final int x = 3;
x = 4;
```

Some people recommend making variables final when possible.

The `this` keyword

The keyword `this` is used by a class to refer to itself. It is usually used if you need to distinguish between a variable or method of a class and something else with the same name, like a parameter of a function. For instance, in the constructor below, we use `this` to tell the difference between the class variable `x` and the

parameter x:

```
private int x;
public MyClass(int x)
{
    this.x = x;
}
```

The `this` keyword is also sometimes used to pass references of the current class to methods that might need it. For instance, in some GUI programs, you may see the following line:

```
addMouseListener(this);
```

The mouse listener needs to know what class to listen to, and if we want it to be the current class, we use the `this` keyword to do that. There are other occasional uses of `this` that you might see, such as if a nested class (class within a class) needs to refer to its outer class.

4.7 Inheritance

Sometimes you will write a class and then need another class that is extremely similar to the one you already wrote but maybe with just one method changed or added. Rather than copying and pasting all the code from that class into the new one, you can use inheritance. Inheritance is a kind of parent/child relationship where you have a parent class and a child class that "inherits" the functionality of the parent and possibly adds a few new things to it or changes some things.

Here is a simple example. First the parent class:

```
public class MyClass
{
    protected int x;

    public MyClass(int x)
    {
        this.x = x;
    }

    public void printHello()
    {
        System.out.println("Hello from MyClass");
    }

    public void printBye()
    {
        for (int i=0; i<x; i++)
            System.out.print("Bye ");
        System.out.println();
    }
}
```

One thing to notice right away is that the variable x is protected instead of private. If we make it `protected`, the child classes that inherit from this class will have access to it. If it were private, then only the parent would have access to it.

Here is a child class:

```
public static class ChildClass extends MyClass
{
    public ChildClass(int x)
    {
        super(x);
    }

    @Override
    public void printHello()
    {
        System.out.println("Hello from ChildClass");
    }
}
```

We notice a few things here. First, to indicate that we are inheriting from MyClass, we use the `extends` keyword. Next, our ChildClass constructor has `super(x)`. This calls the parent class's constructor and is required. The child class's constructor might go on to do other things, but the first line must be a call to `super`. We also see that the child class chooses to override the parent class's `printHello` method. The `@Override` annotation is used to indicate that we are overriding a method. The annotation is not strictly required, but is a good habit to get into.

Here is some code to test both classes:

```
MyClass myClass = new MyClass(3);
ChildClass childClass = new ChildClass(5);

myClass.printHello();
myClass.printBye();

childClass.printHello();
childClass.printBye();
```

We see that the child class has access to the parent method's `printBye` method. Here is the output of the code:

```
Hello from MyClass
Bye Bye Bye
Hello from ChildClass
Bye Bye Bye Bye Bye
```

Another example

Earlier, we created a class called `Player` representing a `Player` in a simple fighting game. Here is the class again, with its fields changed to `protected`.

```
public class Player
{
    protected String name;
    protected int hitPoints;
    protected int attackPower;

    public Player(String name, int hitPoints, int attackPower)
    {
        this.name = name;
        this.hitPoints = hitPoints;
        this.attackPower = attackPower;
    }

    public String getName()
    {
        return name;
    }

    public int attack()
    {
        Random random = new Random();
        return random.nextInt(attackPower);
    }

    public void takeDamage(int amount)
    {
        hitPoints -= amount;
    }

    public boolean isAlive()
    {
        return hitPoints > 0;
    }
}
```

We can use this class to create different players with varying attack power and hit points, but what if we want each character to have its own unique special power? This special power would be defined in its own method. Suppose we want two players: one called an `Ogre` whose special power is an attack that is very powerful, but tends to miss a lot, and another called `Mage` whose special power is to heal all its hit points, but it can only use

that power once. These players would have all the other characteristics of a `Player` object, just with these special powers. Inheritance will help us with this. Here is the `Ogre` class:

```
public class Ogre extends Player
{
    public Ogre(String name)
    {
        super(name, 10, 50);
    }

    public int specialPower()
    {
        Random random = new Random();
        int r = random.nextInt(10);
        if (r == 0)
            return 99;
        else
            return 0;
    }
}
```

We start with the class above by saying it extends `Player`, which means it gets all the variables and methods of that class. Next comes the constructor. Its only action is to call its parent's constructor, and this will fix the Ogre's attack power at 10 and its hit points at 50. The name will be provided by the person who creates the `Ogre` object. For instance, the following line will create an `Ogre` named Tom:

```
Ogre ogre = new Ogre("Tom");
```

That Ogre (like all Ogres) will have 50 hit points and attack power 10. The only other part of the `Ogre` class is the `specialPower` method. It will return a 99 attack 10% of the time and a 0 attack the rest of the time.

Here now is the `Mage` class:

```
public class Mage extends Player
{
    private boolean usedSpecialPower;

    public Mage(String name)
    {
        super(name, 4, 100);
        usedSpecialPower = false;
    }

    public boolean healAll()
    {
        if (!usedSpecialPower)
        {
            hitPoints = 100;
            usedSpecialPower = true;
            return true;
        }
        else
            return false;
    }
}
```

The `Mage` class has a lot in common with the `Ogre` class, but the fact that the `Mage` can only use its special power once adds a new wrinkle. To accomplish this, we introduce a boolean variable `usedSpecialPower` into the class to keep track of whether the power has been used or not. That variable is set to `false` in the constructor, since initially the power hasn't been used, and it is set true in the `healAll` method.

The super keyword

The keyword `super` is used by a class to refer to its parent. It must be used in a child's constructor to build the parent object before doing anything extra that the child needs. For instance, let's suppose a parent class has two fields `x` and `y` that need to be set and the child has those two plus another one called `z`. Here is what the constructor would look like:

```
public ChildClass(int x, int y, int z)
```

```
    {
        super(x, y);
        this.z = z;
    }
```

The `super` keyword can also be used to refer to a method of the parent if you need to distinguish it from the child's method. For instance, if the child overrides one of its parent's methods called `someMethod` and you still need to refer to the parent's method, you could use `super.someMethod` to refer to the parent's method.

Modifiers on methods

If a method is declared as `protected`, then it is visible to and can be overridden by its subclasses. However, if it is `private`, then it is only visible to itself and not to its subclasses (or any other classes).

As noted earlier, if a variable is declared as `final`, then it cannot be reassigned. Similarly, if a method is declared `final`, it cannot be overridden. And if a class is declared with the `final` keyword, then it cannot be subclassed.

Abstract classes

An abstract class is a class that is abstract in the sense that you are not able to create objects directly from it. Rather, it is just to be used as a base class for other classes to inherit from. The Java API makes extensive use of abstract classes.

As an example, with the `Player` class we had earlier, suppose that each subclass should have a method called `specialAttack` that is some special type of attack, with a different type of power for each subclass, but each with the same declaration (say returning an `int` and taking no parameters. We could declare the `Player` class like below:

```
public abstract class Player
{
    protected String name;
    protected int hitPoints;
    protected int attackPower;

    public Player(String name, int hitPoints, int attackPower)
    {
        this.name = name;
        this.hitPoints = hitPoints;
        this.attackPower = attackPower;
    }

    public String getName()
    {
        return name;
    }

    public int attack()
    {
        Random random = new Random();
        return random.nextInt(attackPower);
    }

    public void takeDamage(int amount)
    {
        hitPoints -= amount;
    }

    public boolean isAlive()
    {
        return hitPoints > 0;
    }

    abstract int specialAttack();
}
```

Any class inheriting from this one would need to specify code for the `specialAttack` method. And we wouldn't

ever create a `Player` object; we would only create objects from subclasses of `Player`.

About inheritance

Inheritance is used a lot in the Java API. It is recommended not to overuse inheritance, especially as a beginner, as overly complicated inheritance hierarchies (A is a parent of B which is a parent of C which is a parent of ...) can become a mess to deal with unless they are very well designed.

4.8 Wrapper classes and generics

Wrapper classes

The data types int, long, double, char, and boolean (as well as others – short, float, and byte) are called *primitive data types*. Along with strings, these primitive types form the basis for all other data types. They are stored internally in a way that makes them work efficiently with the underlying hardware. They are not objects. They do not have methods and cannot be used as generic arguments to Java lists. However, there are classes, called *wrapper classes*, that have a few methods and allow integers, doubles, etc. to be used in lists and other places where objects are needed.

The wrapper class for `int` is `Integer`. The wrapper class for the other primitive data types are gotten by capitalizing the first letter of each type. For instance, the wrapper class of `double` is `Double`.

The `Integer` class contains two useful constants, `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, that give the largest and smallest values that can be held in an `int`. Many of the other classes have a similar field. These are useful if you need a value that is guaranteed to be greater than or less than any other value.

The most useful method of the Integer class is `Integer.parseInt`, which converts a string into an integer. Here is an example of it in action:

```
String s = "11/30/1975";
int month = Integer.parseInt(s.substring(0,2));
```

There are similar methods, e.g., `Double.parseDouble`, in the other wrapper classes.

To reiterate, wrapper classes are useful where you want to use something like `int` or `double`, but can't because an actual class is required, like when declaring a list:

```
List<Integer> list = new ArrayList<Integer>();
```

Generics

Generics are a feature added onto the Java language in version 5. A data type in slanted brackets <> indicates generics are being used. Every time you create a list using `List<Integer>` or `List<String>`, you are using generics. Before generics, if you were writing your own List class, you would have to create a whole bunch of separate classes like `IntegerList`, `StringList`, etc. to handle all the possible data types. Either that or you would have to work some magic using the `Object` class (which is the class from which all classes in Java ultimately inherit).

You can use generics when writing your own classes and methods. Here is an example of a method that reverses the elements of a list of any data type:

```
public static <T> List<T> reverse(List<T> list)
{
    List<T> reversedList = new ArrayList<T>();
    for (int i=list.size()-1; i>=0; i--)
        reversedList.add(list.get(i));
    return reversedList;
}
```

Here is an example of a really simple class with a variable x that can be of any data type:

```
public static class MyClass<T>
{
    private T x;

    public MyClass(T x)
    {
        this.x = x;
    }

    public T getX()
    {
        return x;
    }
}
```

You could create `MyClass` objects like below:

```
MyClass<Integer> myClass1 = new Myclass<Integer>();
MyClass<String> myClass2 = new Myclass<String>();
```

Although Java will sometimes let you leave off the type parameter, you shouldn't, as that can lead to hard to find problems in your programs.

4.9 References and garbage collection

Primitive data types like `int` and `double` behave differently than classes like `Scanner` or `ArrayList`. Consider the following code:

```
int x = 3;
x = 4;
```

Initially, the variable x refers to a specific memory location that holds the value 3. The second line goes to that memory location and replaces the value of 3 with 4.

Now consider the following code:

```
List<Integer> x = new ArrayList<Integer>();
x.add(3);
x = new ArrayList<Integer>();
```

What is different here is that there is no list stored at x. Rather, x holds a *reference* to a list that is stored elsewhere in memory. In other words, the value stored at x is just a memory location indicating where the list is stored in memory. There are various performance and efficiency reasons for this that we won't get into.

The third line then creates a new list in memory, separate from the original list and points x to that new list. So there are now two lists in memory, the original one with the 3 in it and the new empty list.

All classes are stored in the same way. When we create an object from a class, say like `Player player = new Player("Ogre", 20, 10)`, we are creating a player object in memory somewhere, and the `player` variable just stores a link (a reference) to that object in memory. On the other hand, whenever we create a primitive data type like `boolean b = true`, the value `true` is stored in at the precise memory location of the variable `b`.

Now consider the following code:

```
int a = 4;
int b = a;
b = 2;

List<Integer> x = new ArrayList<Integer>();
List<Integer> y = x;
y.add(4);
```

The first segment ends up with `a=4` and `b=2`, as we would expect. The second segment ends up with `x` and `y` both equal to the same list, `[4]`. This might be surprising unless you followed the previous discussion carefully. The reason has to do with how classes are stored. The first list line creates a an empty list in memory and the variable x points to that list. The second line does not create a new list. It just points `y` to the same list that `x` is

pointing to. So essentially x and y are two names for the same list in memory. Thus the third line modifies that shared list, making x and y both equal to [4].

This is something to be very careful of. In particular, the following will not work to make a temporary copy of a list called list:

```
List<Integer> copy = list;
```

This just makes an alias, with copy and list both referring to the same list in memory. Any changes to copy will affect list as well. Instead, do the following:

```
List<Integer> copy = new ArrayList<Integer>(list);
```

The new statement instructs Java to create a new list in memory, so now there are two copies of the list in memory, list referring to one and copy referring to the other.

Garbage collection

While your program is running, Java periodically checks to see if any of your objects are no longer being used. If an object is not being used any more, Java *garbage-collects* it, which is to say the portion of memory that is used to store the object is freed up to be used for other purposes. For instance, suppose we have the following:

```
List<Integer> list = new ArrayList<Integer>();
Collections.addAll(list, 2, 3, 4);
list = new ArrayList<Integer>();
```

The first two lines create the list [2,3,4] in memory. The third line points the list variable to a new empty list, leaving [2,3,4] still in memory, but with nothing pointing to it. Java's garbage collector will eventually notice this and free up the memory that the [2,3,4] list is occupying.

In small programs, you don't have to worry too much about garbage collection, but it can be important in larger programs, both in terms of the space saved by garbage collection and in terms of the time it takes for the garbage collector to do its work. Sometimes, if your program freezes at random times for fractions of a second or even seconds, it could be the garbage collector doing its thing.

Passing things to functions

Consider the following function:

```
public static void f(int x)
{
    x = 999;
}
```

Suppose we create a variable int y = 1 and call f(y). The function will have no effect on the value of y. The parameter x is stored in a completely different memory location from y. Its value is set to the value of y, but modifying it cannot affect y since x and y are stored in different locations.

Next consider the following:

```
public static void g(List<Integer> x)
{
    x.add(4);
}
```

Suppose we create a list called list and call g(list). This function will affect list by adding a 4 to the end of it. The reason is that the parameter x is a reference to the same list in memory that the variable list refers to. The lesson here is that if you are writing a function that takes a list or some other object, changes you make to that list/object can affect the caller's object, so you might want to make a copy of it if you need to change it.

However, consider the following:

```
public static void h(List<Integer> x)
{
    x = new ArrayList<Integer>();
}
```

If we create a list called `list` and call `h(list)`, we will find that the original list is not affected by the function. This is because `list` and `x` are separate variables that initially point to the same list in memory. When we set `x` to equal to the new list, what happens is `list` is still pointing to the original list, while `x` is now pointing to a new empty list.

4.10 Interfaces

An interface is the next level of abstraction up from a class. It is sort of like a template for classes in the same way that a class is a template for objects. For instance, if we have a class called `Investment` with `principal` and `interest` fields, that class serves a template for a variety of different `Investment` objects, with varying values of `principal` and `interest`. We might not know what those values are, but we do know that every `Investment` object has those fields.

With an interface, we create a template that says any class *implementing* that interface will have certain methods, though it is up to the class how it implements those methods. Here is a really simple example of an interface:

```
public interface MyInterface
{
    void printHello();
}
```

We can then create some classes that implement the interface:

```
public class Class1 implements MyInterface
{
    @Override
    public void printHello()
    {
        System.out.println("Hello from class 1");
    }
}

public class Class2 implements MyInterface
{
    @Override
    public void printHello()
    {
        System.out.println("Hello from class 2");
    }
}
```

Then suppose we have a program that tests everything. We might test it with the following code:

```
MyInterface a = new Class1();
MyInterface b = new Class2();

a.printHello();
b.printHello();
```

```
Hello from class 1
Hello from class 2
```

We know that anything that implements MyInterface will have a printHello method, though what that method does may vary from class to class.

One example of a familiar interface is the `java.util.List` interface. One of the classes implementing it is `ArrayList`, but there is another, called `LinkedList`. The idea is that any class that implements the `List` interface should have methods like `add`, `get`, and `set`, but it is up to the class exactly how it wants to implement those methods. For instance, an `ArrayList` internally stores the list elements in an array, whereas `LinkedList` stores list elements spread out through memory, connected by links.

In some applications `ArrayList` is faster, while in others `LinkedList` is faster. Since we know that `ArrayList` and `LinkedList` come from the same interface, we know that the both have `add`, `get`, and `set` methods, so if we change our minds as to which type of list we want to use, switching will only require changing the declaration line and nothing else.

The `List` interface also allows us to just write one function that operates on lists, saving us from having to write separate functions for each type of list.

The `Comparable` interface

One important Java interface is called `Comparable`. A Java class can implement this interface to make it possible to compare two different objects to see if one is greater than or less than another. For instance, say we have a card class like the one below.

```
public class Card
{
    private int value;
    private String suit;

    public Card(int value, String suit)
    {
        this.value = value;
        this.suit = suit;
    }
}
```

This a simplified version. A real card class would have some other methods. Anyway, suppose we want to be able to compare Card objects solely based on their `value` fields. Here is how we modify the class:

```
public class Card implements Comparable<Card>
{
    private int value;
    private String suit;

    public Card(int value, String suit)
    {
        this.value = value;
        this.suit = suit;
    }

    public boolean compareTo(Card other)
    {
        return value - other.value;
    }
}
```

The Comparable interface requires that we implement a method called `compareTo` that describes how to compare two objects. The method must return either a negative value, 0, or a positive value depending on whether current object is smaller than, equal to, or larger than the object it is being compared to. The expression `value-other.value` accomplishes this.

Now, if we have two card objects, `card1` and `card2`, we can compare them using an if statement like below, which asks whether `card1` is less than `card2`.

```
if (card1.compareTo(card2) < 0)
```

Here's something else we can do. Suppose we have a list of Card objects and we want to sort them in order. Since we have implemented the Comparable interface, we can call `Collections.sort` or `list.sort` to sort the list. These methods will sort a list of any type of objects, provided those objects implement the `Comparable` interface. It relies on the object's `compareTo` method to do the sorting.

This demonstrates one of the main ideas of interfaces, that if your class implements a certain interface, then other things (like `Collections.sort` can depend on that method being there and use it.

Note: In Java 8 and later, if you just need to sort an object by one of its fields, there is a shortcut. Here is how it would work with a list of Card objects:

```
list.sort((card1, card2) -> card1.value - card2.value);
```

4.11 Nested classes

A *nested* or *inner class* is a class inside of a class. Such a class can be public to be visible to outside classes or private to be something the class uses for its own purposes.

Often within a class you need a small class for something and that class would not be useful anywhere else or would just clutter up your project. In these cases, it might be worth just declaring that small class inside the other one. One place we use this extensively is in GUI programming. When we want to assign an action to happen when a button is clicked, Java requires us to create a class that specifies what happens. Often, that action is something really short and simple where we wouldn't want to create a whole separate file, so we use a nested class. See Section 5.4 for examples.

Note that if you need to refer to a nested class from inside a static method (like the `main` method), then it is necessary to declare the nested class itself as static.

4.12 Exceptions

When something goes wrong with your program, it can crash. For example, the following program will crash with an `ArithmeticException`:

```
int x = 0;
System.out.println(2 / x);
```

The problem is that we aren't allowed to divide by 0. We can imagine a scenario where a program is doing a complicated calculation and at some unpredictable point it ends up dividing by 0. If this is part of a program that we are writing for someone else, we would not want the program to crash. Java provides something called *exception handling* to allow a program to recover from errors without crashing. Here is an example of how it works:

```
int x = 0;
try
{
    System.out.println(2/x);
}
catch (ArithmeticException e)
{
    System.out.println("You tried to divide by 0.")
}
```

The try/catch block is what we use to recover from errors. The `try` block contains the code that might cause an error. The `catch` block is what we do in the case of an error, an `ArithmeticException` in this case. When we run this code, we will not see an exception in the console. Instead, we will just see a message telling us that we tried to divide by 0. The program then continues running instead of crashing.

There are a wide variety of exceptions. For example, the following code can be used to allow our program to recover if a file is not found:

```
Scanner scanner;
List<String> words = new ArrayList<String>();
try
{
    scanner = new Scanner(new File("wordlist.txt"));
    while (scanner.hasNext())
        words.add(scanner.nextLine());
}
catch (IOException e)
{
    System.out.println("Error reading file.  List will be empty");
}
```

Recall from Section 1.19 the other approach to exceptions is to add a `throws` statement, which essentially says you are ignoring the error and if things crash, then so be it.

It is possible to have multiple catch blocks, like below:

```
try
{
    scanner = new Scanner(new File("wordlist.txt"));
    while (scanner.hasNext())
        words.add(scanner.nextLine());
}
catch (IOException e)
{
    System.out.println("Error reading file.  List will be empty");
}
catch (Exception e)
{
    System.out.println("Something else went wrong. It was " + e);
}
```

In this example, the second catch just catches `Exception` which is a general class just standing for some unspecified type of exception.

Throwing your own exceptions

Your own classes can throw exceptions. For example, if you created your own list class and someone tries to delete from an empty list, you might have the following code raise an exception:

```
if (size < 0)
    throw new RuntimeException("The list is empty.");
```

You can also throw any exception already defined in Java, like `IndexOutOfBounds` exception or create your own exception classes by subclassing `Exception` or `RunTimeException`.

Chapter 5

GUI Programming

5.1 A template for other GUI Programs

GUI stands for Graphical User Interface. GUI-based programs in Java can have buttons, scrollbars, etc. Here is a simple program that we will use as a template for more involved programs:

```
import java.awt.Dimension;
import javax.swing.*;

public class GuiTemplate
{
    private JFrame frame;
    private JPanel mainPanel;
    // Declarations of labels, buttons, etc. go here...

    public GuiTemplate()
    {
        frame = new JFrame("This is the text in the title bar.");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new GuiTemplate();
            }
        });
    }

    private void initialize()
    {
        mainPanel.setPreferredSize(new Dimension(300,300));
        // Create labels, buttons, etc. and place them on the window here...
    }
}
```

This program creates an empty window. You can use this as a template for creating other GUI programs. Just copy it into whatever class you create, replacing the three occurrences of `GuiTemplate` with the name of your class.

Going through the code, the first thing we have are two variable declarations. The `JFrame` is the window itself, and the `JPanel` is an object that goes on the window whose job it will be to hold other things, like labels and buttons. After these two lines is where declarations of buttons, labels, and other things will go.

Next comes the constructor. It creates the `JFrame` and `JPanel` and does a bunch of other tasks. In any GUI program you write, you'll probably want most of these things here, but try experimenting with them to see how they work. The `setLookAndFeel` call is not essential but is designed to make the GUI look like an ordinary window on whatever operating system you are using. The `initialize` line calls our initialization function, which we'll talk about in a minute. The `setDefaultCloseOperation` call is important; if you delete it, then when you close your window, the program will still be running in the background. The `getContentPane` line adds the `JPanel` to the screen; don't delete it. The `setLocationByPlatform` line positions the window on the screen according to where your operating system prefers; it can be deleted. The `frame.pack` line tells Java to size the window properly, and the last line is required for your window to actually be visible on the screen.

Next comes the `main` method. It calls the `constructor`. The `SwingUtilities` stuff is the recommended way to start a GUI. At this point don't worry too much about it.

Finally, we have the `initialize` method. The first line sets the dimensions of the `JPanel` and therefore also the window itself. You can delete it if you want to let Java decide how big to make the window. After that line will be where we create our labels, buttons, etc. and add them to the window.

That's a lot to remember, so for now you can just copy the template each time you are making a new GUI program, replace `GuiTemplate` with your class name, and add things in the commented areas. Note that there are a lot of other ways to do GUIs in Java. This is just one relatively simple way. Eventually, you will probably want to use a drag-and-drop GUI builder. Each of the IDEs has one built in. The one in Eclipse is called WindowBuilder and may need to be installed, depending on which version of Eclipse you have.

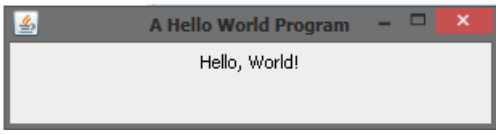
5.2 A Hello World program

Here is a Hello World program.

```
import java.awt.Dimension;
import javax.swing.*;

public class GuiHelloWorld
{
    private JFrame frame;
    private JPanel mainPanel;
    private JLabel label;  // Declare the label

    public GuiHelloWorld()
    {
        frame = new JFrame("A Hello World Program");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new GuiHelloWorld();
            }
        });
    }

    private void initialize()
```

```
        {
            mainPanel.setPreferredSize(new Dimension(300,50));
            label = new JLabel("Hello, World!");  // Create the label.
            mainPanel.add(label);  // Add the label to the screen.
        }
    }
```

This is what it looks like on my computer:



The program above has three lines added to the GUI template. The first is the declaration of the `JLabel` near the top of the program. A `JLabel` is a GUI object used to display stuff, usually text. The other two lines added are in the `initialize` method. These two lines create the label and place it on the window.

5.3 Customizing labels and other widgets

For this section assume we have created a label called `label`. We will look at how to customize the label. Most of what we do here applies to customize other things, like buttons and text fields.

Changing the font

We can change the font of the label like below:

```
label.setFont(new Font("Verdana", Font.PLAIN, 16));
```

The font name is a string. `Font.PLAIN` can be replaced with other things like `Font.BOLD` or `Font.ITALIC`. The last argument to the `Font` constructor is the size of the font.

Changing the foreground color

To change the color of the text in the label, use something like the following:

```
label.setForeground(Color.YELLOW);
```

The `Color` class contains a number of constant color names. You can also specify the red, green, and blue components separately, like below.

```
label.setForeground(new Color(255, 120, 40));
```

When specifying a color this way, you are specifying how much red, green, and blue make up the color. The values range from 0 to 255. All zeros is black, while all 255s is white.

Changing the background color

Changing the background color of the label is similar, except that we have to change the background from transparent to opaque:

```
label.setBackground(Color.BLUE);
label.setOpaque(true);
```

Getting and setting the text

To change the text in the label, do something like the following:

```
label.setText("This is the new text.")
```

To get a string containing the label's text, use the following:

```
label.getText();
```

Note that there are getters for many other fields; for instance, `getForeground` and `getBackground` to get the label's colors.

Using an image instead of text

We can also have our label hold an image instead of text:

```
label = new JLabel(new ImageIcon("SomeImage.png"));
```

5.4 Buttons

Creating a button is similar to creating a label:

```
button = new JButton("Click me");
```

To make the button do something when clicked, we have to add something called an `ActionListener` to the button. To do this, we can add the following to our GUI class's constructor:

```
button.addActionListener(new ButtonListener());
```

`ButtonHandler` refers to a class that we now need to create. The class must implement the `ActionListener` interface. Here is an example that sets the text in a label whenever a button is clicked:

```
private class ButtonListener implements ActionListener
{
    @Override
    public void actionPerformed(ActionEvent arg0)
    {
        label.setText("Button has been clicked");
    }
}
```

The `actionPerformed` method must be implemented and contains the code that is executed when the button is clicked. This class could go in a separate file, but it usually easier to make it a nested class inside the current

Here is an example program that has a button and a label whose text changes once the button has been clicked:

```
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class ButtonExample
{
    private JFrame frame;
    private JPanel mainPanel;
    private JLabel label;
    private JButton button;

    public ButtonExample()
    {
        frame = new JFrame("Button Example");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
```

```
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new ButtonExample();
            }
        });
    }

    private void initialize()
    {
        mainPanel.setPreferredSize(new Dimension(200,30));
        label = new JLabel("Button not clicked yet");
        button = new JButton("Click me");
        button.addActionListener(new ButtonHandler());

        mainPanel.add(label);
        mainPanel.add(button);
    }

    private class ButtonHandler implements ActionListener
    {

        @Override
        public void actionPerformed(ActionEvent e)
        {
            label.setText("Button has been clicked.");
        }
    }
}
```

5.5 Text fields

To add a place for users to enter text, use a JTextField:

```
textEntry = new JTextField();
```

The size Java assigns to the text field might not be what you want. You can use something like the following to set how wide it is:

```
textEntry.setColumns(5);
```

To get the text that the user typed in, use the `getText` method like below:

```
String text = textEntry.getText();
```

It may be necessary to use `Integer.parseInt` or `Double.parseDouble` to convert the text to a number if you are getting numerical input.

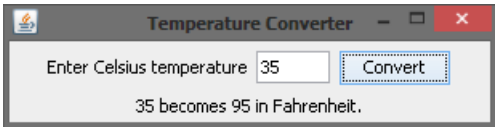
You can add an ActionListener to a text field. Whenever the user presses the Enter/Return key, the code from the ActionListener will execute.

Here is a simple GUI-based temperature converter that contains a text field.

```
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class Converter
{
    private JFrame frame;
    private JPanel mainPanel;
    private JLabel messageLabel;
    private JLabel answerLabel;
    private JTextField entry;
    private JButton convertButton;
```

```
    public Converter()
    {
        frame = new JFrame("Temperature Converter");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new Converter();
            }
        });
    }

    private void initialize()
    {
        mainPanel.setPreferredSize(new Dimension(350,60));
        messageLabel = new JLabel("Enter Celsius temperature");
        answerLabel = new JLabel();

        convertButton = new JButton("Convert");
        convertButton.addActionListener(new ButtonListener());

        entry = new JTextField();
        entry.setColumns(5);
        entry.addActionListener(new ButtonListener());

        mainPanel.add(messageLabel);
        mainPanel.add(entry);
        mainPanel.add(convertButton);
        mainPanel.add(answerLabel);
    }

    private class ButtonListener implements ActionListener
    {
        @Override
        public void actionPerformed(ActionEvent e)
        {
            int value = (int)Math.round(Integer.parseInt(entry.getText()) * 9 / 5.0 + 32);
            answerLabel.setText(entry.getText() + " becomes " + value + " in Fahrenheit.");
        }
    }
}
```

Here is what the program looks like on my computer:



Going quickly through the program code, we start with declarations of all the widgets that will be used. The constructor follows. It is the same code as in the template, except with the title text changed. The `main` method is the same as in the template. The `initialize` method creates all the widgets and puts them on the window. The Action Listener is where the interesting stuff happens. It is called whenever the button is clicked or when the enter key is pressed in the text field. In the Action Listener, we read the text from the text field using the text field's `getText` method. We convert that string to an integer, do the conversion, and display the result in `answerLabel` by using the label's `setText` method.

5.6 Layout managers

The examples up to this point have used the default layout of the `JPanel` class, which is something called `FlowLayout`. The way it works is it puts things on the screen in the order that you add them, starting at the upper left, moving right until there is no more room horizontally, and then moving down typewriter style. If you resize the screen, things will move around. This is not usually what we want.

There are a few options for better layouts. One option is to use a drag-and-drop GUI builder. Such a program will generate the GUI layout code for you. Here, however, we will cover how to hand-code simple layouts.

GridLayout

GridLayout lays out widgets in a grid. It is declared like below, where `r` and `c` should be replaced with the number of rows and columns in your grid:

```
mainPanel.setLayout(new GridLayout(r, c));
```

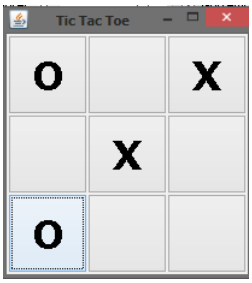
Then, when you put widgets on the window, they are still added typewriter style, going left to right and then top to bottom, but when the screen is resized, the number of rows and columns will stay fixed.

Here is a Tic Tac Toe program that uses GridLayout:

```
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class TicTacToe
{
    private JFrame frame;
    private JPanel mainPanel;
    private JButton[][] buttons;
    private String player;  // keeps track of if it is X's or O's turn

    public TicTacToe()
    {
        frame = new JFrame("Tic Tac Toe");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new TicTacToe();
            }
        });
    }

    private void initialize()
    {
        mainPanel.setLayout(new GridLayout(3,3));
        mainPanel.setPreferredSize(new Dimension(200,200));

        // Create the 3 x 3 array of buttons and place them on screen
        buttons = new JButton[3][3];
        for (int i=0; i<3; i++)
```

```
            {
                for (int j=0; j<3; j++)
                {
                    buttons[i][j] = new JButton(" ");
                    buttons[i][j].setFont(new Font("Verdana", Font.BOLD, 32));
                    buttons[i][j].addActionListener(new ButtonListener());
                    mainPanel.add(buttons[i][j]);
                }
            }
            player = "X";
        }

        private class ButtonListener implements ActionListener
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                // Figure out which button was clicked, put in X or O, and change player
                JButton button = (JButton)(e.getSource());
                if (button.getText().equals(" "))
                {
                    button.setText(player);
                    player = player.equals("O") ? "X" : "O";
                }
            }
        }
    }
```

Here is what it looks like on my machine:



The GridLayout code shows up in the `initialize` method. We set the grid to be 3 rows and 3 columns and use a 3×3 array of buttons. It is also worth looking at the ActionListener code. We use `e.getSource()` to figure out which button is being clicked. We then look at the text currently in that button. If it's currently a space (i.e., not already set to be an X or O), then we set its text to the `player` variable. The `player` variable is a class variable that is initially set to X and gets changed whenever a player takes their turn. The code that changes it uses the ternary operator as a one-line if/else statement (see Section 2.10).

One note about GridLayout is the widgets are resized so that the overall grid takes up the entire available window. This means that the buttons grow to be rather large. This may be desirable or not. If not, the trick is to combine layouts, which we will cover in a bit.

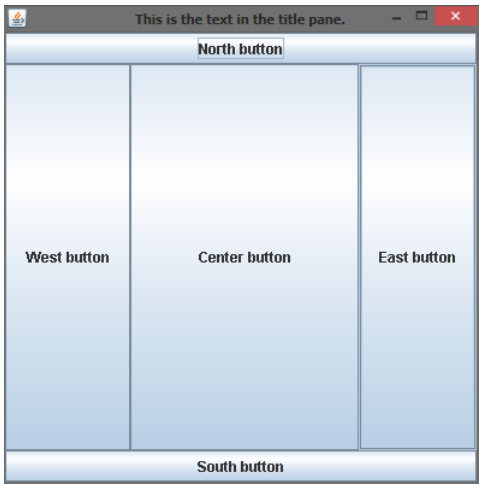
BorderLayout

BorderLayout divides the screen up into five regions: North, South, East, West, and Center. Here is an example:

```
mainPanel.setLayout(new BorderLayout());

button1 = new JButton("North button");
button2 = new JButton("South button");
button3 = new JButton("East button");
button4 = new JButton("West button");
button5 = new JButton("Center button");

mainPanel.add(button1, BorderLayout.NORTH);
mainPanel.add(button2, BorderLayout.SOUTH);
```

```
mainPanel.add(button3, BorderLayout.EAST);
mainPanel.add(button4, BorderLayout.WEST);
mainPanel.add(button5, BorderLayout.CENTER);
```

The figure below shows the what this would look like:



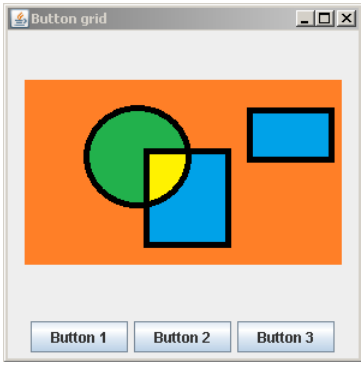
What happens is that the center will expand horizontally and vertically to fill as much available space as it can before it runs into the other areas. The north and south expand horizontally but not vertically, while the east and west expand vertically but not horizontally.

Combining layouts

One way to get things to look right is to combine multiple layouts. In order to that, we will use multiple JPanels. JPanels are a kind of generic widget that are often used to hold groups of other widgets. Here is an example:

```
mainPanel.setLayout(new BorderLayout());

buttonPanel = new JPanel();

imageLabel = new JLabel(new ImageIcon("image.png"));
button1 = new JButton("Button 1");
button2 = new JButton("Button 2");
button3 = new JButton("Button 3");

buttonPanel.add(button1);
buttonPanel.add(button2);
buttonPanel.add(button3);

mainPanel.add(buttonPanel, BorderLayout.SOUTH);
mainPanel.add(imageLabel, BorderLayout.CENTER);
```

The overall GUI window has a border layout, while the button panel has the default FlowLayout. The button panel is put into the south portion of the overall GUI window. The buttons will now be reasonably sized. Here is what it looks like (using a goofy image I made in the center):



You can use multiple panels with varying layouts and nest panels inside other panels to get the look you are after.

Note that a common mistake that leads to a null pointer exception is to forget the `buttonPanel = new JPanel()` line. Also, note that there are a variety of other layouts that we won't cover here, including `GridBagLayout`, `CardLayout`, `SpringLayout` and more. See the official Java documentation on layout managers at http://docs.oracle.com/javase/tutorial/uiswing/layout/using.html.

5.7 Checkboxes

JCheckBox is used for a checkbox. Here is code that creates a checkbox:

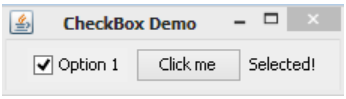
```
box = new JCheckBox("This is a check box");
box.addItemListener(new BoxListener());
```

The `isSelected` method is used to tell whether or not the box is checked. Here is a simple checkbox program:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class CheckBoxDemo
{
    private JFrame frame;
    private JPanel mainPanel;
    private JLabel label;
    private JButton button;
    private JCheckBox checkBox;

    public CheckBoxDemo()
    {
        frame = new JFrame("CheckBox Demo");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new CheckBoxDemo();
            }
        });
    }

    private void initialize()
```

```
        {
            label = new JLabel("Not selected!");

            button = new JButton("Click me");
            button.addActionListener(new ButtonListener());

            checkBox = new JCheckBox("Option 1");

            mainPanel.add(checkBox);
            mainPanel.add(button);
            mainPanel.add(label);
        }

        private class ButtonListener implements ActionListener
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                if (checkBox.isSelected())
                    label.setText("Selected!");
                else
                    label.setText("Not selected!");
            }
        }
    }
```

This is what it looks like on my machine:



In particular, notice the ActionListener code that uses the `isSelected` method to determine if the checkbox is selected or not.

In this code, the label is only updated when the big button is clicked. If we want the label to update whenever the checkbox is checked or uncheck, we can do something a little like adding an ActionListener to an ordinary button, except that we use an ItemListener. To add this to the above code, put the following line in the `initialize` method:

```
checkBox.addItemListener(new BoxListener());
```

And then add the ItemListener code near the end of the class:

```
private class BoxListener implements ItemListener
{
    @Override
    public void itemStateChanged(ItemEvent e)
    {
        if (checkBox.isSelected())
            label.setText("Selected!");
        else
            label.setText("Not Selected!");
    }
}
```

5.8 Radio buttons

Radio buttons are a series of objects similar to checkboxes, where only one of them can be selected at a time. Here is the code that sets up some radio buttons:

```
b1 = new JRadioButton("Button 1");
b2 = new JRadioButton("Button 2");
b3 = new JRadioButton("Button 3", true);

buttonGroup = new ButtonGroup();
```

```
buttonGroup.add(b1);
buttonGroup.add(b2);
buttonGroup.add(b3);
```

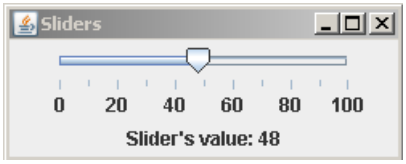
There are a few things to note above. First, the `true` argument for the third button says that is the button that is initially selected. Second, in order to tie the buttons together so that only one of them could be selected at a time, we use something called a ButtonGroup.

Radio buttons also work with ItemListeners, just like checkboxes, and the `isSelected` method is used to tell if the button is selected or not.

Here is what radio buttons look like in a simple program:

5.9 Sliders

Sliders are GUI elements that you can slide with a mouse to change their values, like below:



Here is the code to create the slider above:

```
slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 40);

slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(10);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
```

The first argument to the JSlider constructor specifies its orientation, the second one specifies the minimum and maximum values, and the last one specifies the starting value. The constructor creates a plain slider. The next four lines jazz it up a bit.

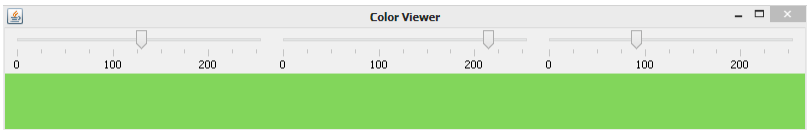
The get the numerical value represented by the slider, use the `getValue` method. To schedule something to happen whenever the slider is used, add a ChangeListener, which is the slider equivalent of an ActionListener. The `getValue` method is used to get the slider's value. The example below demonstrates all of these concepts. It is a color-chooser, where the user can use sliders to specify the red, green, and blue components of a color and the resulting color is displayed in a large label.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class ColorViewer extends JFrame
{
    private JFrame frame;
    private JPanel mainPanel;
    private JLabel colorLabel;
    private JPanel sliderPanel;
```

```
    private JSlider redSlider, greenSlider, blueSlider;

    public ColorViewer()
    {
        frame = new JFrame("Color Viewer");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new ColorViewer();
            }
        });
    }

    private void initialize()
    {
        mainPanel.setPreferredSize(new Dimension(800,100));

        mainPanel.setLayout(new BorderLayout());
        sliderPanel = new JPanel();
        sliderPanel.setLayout(new GridLayout(1,3));

        redSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 128);
        greenSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 128);
        blueSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 128);

        redSlider.setMajorTickSpacing(100);
        redSlider.setMinorTickSpacing(25);
        redSlider.setPaintTicks(true);
        redSlider.setPaintLabels(true);

        greenSlider.setMajorTickSpacing(100);
        greenSlider.setMinorTickSpacing(25);
        greenSlider.setPaintTicks(true);
        greenSlider.setPaintLabels(true);

        blueSlider.setMajorTickSpacing(100);
        blueSlider.setMinorTickSpacing(25);
        blueSlider.setPaintTicks(true);
        blueSlider.setPaintLabels(true);

        sliderPanel.add(redSlider);
        sliderPanel.add(greenSlider);
        sliderPanel.add(blueSlider);

        colorLabel = new JLabel();
        colorLabel.setOpaque(true);
        colorLabel.setBackground(new Color(128,128,128));

        mainPanel.add(sliderPanel, BorderLayout.NORTH);
        mainPanel.add(colorLabel, BorderLayout.CENTER);

        SliderListener sh = new SliderListener();
        redSlider.addChangeListener(sh);
        greenSlider.addChangeListener(sh);
        blueSlider.addChangeListener(sh);
    }

    private class SliderListener implements ChangeListener
    {
```

```
            public void stateChanged(ChangeEvent e)
            {
                int r = redSlider.getValue();
                int g = greenSlider.getValue();
                int b = blueSlider.getValue();
                colorLabel.setBackground(new Color(r,g,b));
            }
        }
    }
```

Here is what the program looks like on my machine:



5.10 More about ActionListeners

Recall that in order for something to happen when a button is clicked, we usually use an ActionListener. This section contains a few tips and tricks for working with them.

Multiple buttons

If you have two buttons, you can either can have an ActionListener for each, or you can have the same ActionListener handle both buttons. For instance, suppose we have two buttons, `button1` and `button2`. If the buttons behave quite differently, two ActionListeners is probably appropriate, but if they behave similarly, then one ActionListener may be better. Here is an example using one ActionListener:

```
private class ButtonListener implements ActionListener
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource()==button1)
            label.setText("Button 1 was clicked");
        else
            label.setText("Button 2 was clicked");
    }
}
```

Maintaining state

Suppose we want to make a button that keeps track of how many times it was clicked. The trick to this is to create a class variable in the `ActionListener`. Here is what the `ActionListener` would look like:

```
private class ButtonListener implements ActionListener
{
    private int numClicks;

    public ButtonListener()
    {
        numClicks = 0;
    }

    @Override
    public void actionPerformed(ActionEvent arg0)
    {
        numClicks++;
        label.setText(numClicks + " clicks");
    }
}
```

Class variables like this can be used to keep track of lots of things, not just the number of clicks.

A Java 8 shortcut

In Java versions 8 and later, there is a one line shortcut that avoids having to create a new class. Here is an example:

```
button.addActionListener(e -> label.setText("Button was clicked"));
```

If the action to be performed consists of multiple statements, the shortcut can be used as shown below, though once things to be more than a couple of statements, it is probably best to just create the class.

```
button.addActionListener(e -> {label.setText("Clicked!"); label.setForeground(Color.RED);});
```

5.11 Simple graphics

Suppose we want to draw things line lines and circles on the screen. What we'll do create a class called DrawingPanel which is a subclass of JPanel, and we'll use that class to do our drawing. We'll treat our drawing panel as if it were any other widget when we add it to the main window. Here is the class:

```
private class DrawingPanel extends JPanel
{
    public DrawingPanel(int width, int height)
    {
        super();
        setBackground(Color.WHITE);
        setPreferredSize(new Dimension(width, height));
    }

    @Override
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        // code to actually draw stuff would go here
    }
}
```

The DrawingPanel constructor take two parameters, specifying the panel's width and height. The way the `paintComponent` method works is it takes a `Graphics` object as a parameter. That `Graphics` object has certain methods that draw things. For instance, these two lines added to the `paintComponent` method will draw a red rectangle:

```
g.setColor(Color.RED);
g.fillRect(50, 30, 10, 10);
```

The screen is arranged so that the upper left corner is $(0, 0)$. Moving right and down increases the x and y coordinates. The first two arguments to `fillRect` are the starting x and y coordinates The other two arguments are the width and height of the rectangle.

The `g.setColor` line is used to change the drawing color to red. That will affect the color of everything drawn until the next occurrence of `g.setColor`.

Some other methods of `Graphics` objects include `drawLine` for lines, `drawOval` for circles and ellipses, and `drawString` for text.

Anytime you want to force the screen to update itself, call the `repaint` method. That method calls our`paintComponent` method and does some other stuff.

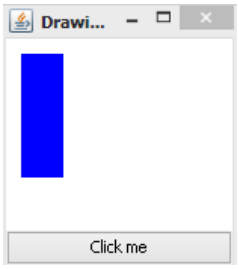
Here is a simple example program. It has a button and a drawing panel. Whenever the button is clicked, a random blue rectangle is drawn.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.*;

public class DrawingExample
{
    private JFrame frame;
    private JPanel mainPanel;
    private JButton button;
    private DrawingPanel drawingPanel;

    public DrawingExample()
    {
        frame = new JFrame("Drawing Example");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new DrawingExample();
            }
        });
    }

    private void initialize()
    {
        mainPanel.setLayout(new BorderLayout());

        button = new JButton("Click me");
        button.addActionListener(new ButtonListener());
        drawingPanel = new DrawingPanel(150, 150);

        mainPanel.add(button, BorderLayout.SOUTH);
        mainPanel.add(drawingPanel, BorderLayout.CENTER);
    }

    private class ButtonListener implements ActionListener
    {
        @Override
        public void actionPerformed(ActionEvent e)
        {
            drawingPanel.repaint();
        }
    }

    private class DrawingPanel extends JPanel
    {
        public DrawingPanel(int width, int height)
        {
            super();
            setBackground(Color.WHITE);
            setPreferredSize(new Dimension(width, height));
        }

        @Override
        public void paintComponent(Graphics g)
        {
            super.paintComponent(g);
            Random random = new Random();
            g.setColor(Color.BLUE);
```

```
                g.fillRect(10, 10, random.nextInt(100)+20, random.nextInt(100)+20);
            }
        }
    }
```

Here is what the program looks like on my machine:



Drawing images

To draw an image, use `drawImage`. First, though the image needs to be loaded, preferably in the `initialize` method:

```
try {
    myImage = ImageIO.read(new File("someImage.jpg"));
} catch (IOException e) {e.printStackTrace();}
```

Then in the `paintComponent` method, we would have something like this:

```
g.drawImage((Image)myImage, 0, 0, 20, 20,
            0, 0, myImage.getWidth(), myImage.getHeight(), null, null);
```

The first parameter is the image itself (cast to type `Image`). The next two parameters are where on the screen to place the image. Then comes the width and height that the image will fill on the screen (this can be used to resize the image). The next four parameters specify what part of the image to copy onto the screen (the parameters here will copy the entire image). The last two parameters we don't use and are set to `null`.

5.12 Timers

In Java GUIs, timers are used to schedule things to happen. Here is an example:

```
timer = new Timer(10, new TimerHandler());
timer.start();
```

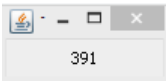
The 10 in the `Timer` constructor says to wait 10 milliseconds between each time the action is performed. The TimerHandler class is an ActionListener that contains the action to be performed. The `timer.start()` line starts the timer. Use `timer.stop()` to stop the timer.

Here is an example of timer program that updates a label with an ever increasing value:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class TimerExample
{
    private JFrame frame;
    private JPanel mainPanel;
    private JLabel label;
    private Timer timer;

    public TimerExample()
    {
        frame = new JFrame("Timer Example");
        mainPanel = new JPanel();
```

```
            try {
                UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
            } catch (Exception e) {}
            initialize();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.getContentPane().add(mainPanel);
            frame.setLocationByPlatform(true);
            frame.pack();
            frame.setVisible(true);
        }

        public static void main(String[] args)
        {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    new TimerExample();
                }
            });
        }

        private void initialize()
        {
            label = new JLabel("0");
            mainPanel.add(label);
            timer = new Timer(10, new TimerHandler());
            timer.start();
        }

        private class TimerHandler implements ActionListener
        {
            private int count = 0;
            @Override
            public void actionPerformed(ActionEvent e)
            {
                count++;
                label.setText("" + count);
            }
        }
    }
```

Here is what the program looks like on my machine:



5.13 Keyboard input

To get keyboard input, like to check if the arrow keys are pressed in a game, we use KeyListeners. One way to use them is to add something like the following line to the `initialize` method:

```
frame.addKeyListener(new KeyHandler());
```

Then create a class that implements the `KeyListener` interface. In it goes the code that says what to do when certain keys are pressed. An example program is shown below:

```
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import javax.swing.*;

public class KeyListenerExample
{
    private JFrame frame;
    private JPanel mainPanel;
    private JLabel label;

    public KeyListenerExample()
    {
```

```
            frame = new JFrame("Key Listener Example.");
            mainPanel = new JPanel();
            try {
                UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
            } catch (Exception e) {}
            initialize();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.getContentPane().add(mainPanel);
            frame.setLocationByPlatform(true);
            frame.pack();
            frame.setVisible(true);
        }

        public static void main(String[] args)
        {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    new KeyListenerExample();
                }
            });
        }

        private void initialize()
        {
            label = new JLabel("No keys have been pressed yet.");
            mainPanel.add(label);
            frame.addKeyListener(new KeyHandler());
        }

        private class KeyHandler implements KeyListener
        {
            @Override
            public void keyPressed(KeyEvent e)
            {
                int key = e.getKeyCode();

                if (key == KeyEvent.VK_LEFT)
                    label.setText("Left arrow pressed!!!!");
                else
                    label.setText(KeyEvent.getKeyText(key) + " was pressed.");
            }

            @Override public void keyTyped(KeyEvent e) {}
            @Override public void keyReleased(KeyEvent e) {}
        }
    }
```

The KeyListener requires that we implement three methods: `keyPressed`, `keyTyped`, `keyReleased`. However, we only need one of them, so we have left the others blank. The main thing we do in the KeyListener is get the key's code using `e.getKeyCode()` and check it. The code for the left arrow is given by `KeyEvent.VK_LEFT`. See the Java API documentation for the names of all the keys.

Another way we will use in some of the programs in Section 5.15 will be to make our `mainPanel` itself implement the `KeyListener` interface.

5.14 Miscellaneous topics

Dialog boxes

A simple way to open a dialog box is shown below:

```
JOptionPane.showMessageDialog(new JFrame(), "I'm a message");
```

There are lots of other types of dialog boxes, such as ones that ask the user to pick from a list of options or enter some input. See https://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html for a nice introduction. It is also not too hard to open fancier dialogs. For instance, `JFileChooser` opens a file-picking dialog.

Exiting a program and closing windows

The following can be used to stop a GUI-based program (or any other program):

```
System.exit(0);
```

To just close a window (say in a program that has multiple windows open), use the following.

```
frame.dispatchEvent(new WindowEvent(frame, WindowEvent.WINDOW_CLOSING));
```

Here `frame` is the `JFrame` variable associated with the window.

Text areas and scroll panes

If you want a space where the user can enter more than one line of text, use `JTextArea`. Here is a way to create a text area that is 80 columns and 20 rows:

```
textArea = new JTextArea("", 80, 20);
```

To attach a scrollbar to the area, use the following:

```
scrollpane = new JScrollPane(textArea);
```

Then, when adding the text area to the screen, add the ScrollPane instead of the JTextArea. For instance:

```
mainPanel.add(scrollpane, BorderLayout.CENTER);
```

Scrollbars can be added to other widgets as well, like combo boxes.

List and combo boxes

One the left side of the window below is a JList and on the right is a JComboBox.

Here is the code to create them:

```
String[] options = {"abc", "def", "ghi", "jkl"};
list = new JList(options);
clist = new JComboBox(options);
```

Use the `getSelectedValue` method to get the item that is currently selected in a JList and use the `getSelectedValue` method to get the item that is currently selected in a JComboBox.

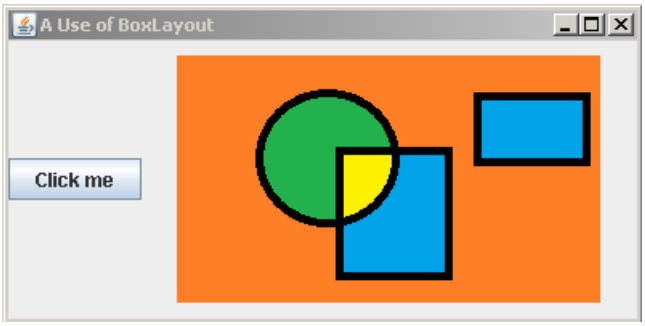
A JList is listened to by a ListSelectionListener and a JComboBox is listened to by an ItemListener.

BoxLayout

BoxLayout is like a more flexible version of FlowLayout that allows you to specify spacing and positioning. It would be a little tedious to go over all the features of BoxLayout here. Instead here's a simple example that can be used to center a button on a widget. Assume that the window has a BorderLayout and an image label in the center. Here is the BoxLayout portion of the code:

```
button = new JButton("Click me");

buttonPanel = new JPanel();
buttonPanel.setLayout(new BoxLayout(buttonPanel, BoxLayout.PAGE_AXIS));
buttonPanel.add(Box.createVerticalGlue());
buttonPanel.add(button);
buttonPanel.add(Box.createVerticalGlue());

mainPanel.add(buttonPanel, BorderLayout.WEST);
```

Here is the result:



If we had used FlowLayout instead, the button would be at the top of the west part of the window.

Positioning the window

In the constructor, use the following line to locate the upper left corner of the window at coordinates (x, y).

```
frame.setLocation(x,y);
```

Changing the icon in the title bar

In the upper left corner of the window, in the title bar, of our GUI programs is the Java coffee cup logo. To change it to something else, use `setIconImage`, in the constructor, like below:

```
frame.setIconImage(new ImageIcon("someImage.png").getImage());
```

Adding a background image

One approach to using a background image is to use the `DrawingPanel` class we created earlier. We will actually make the `mainPanel` be a drawing panel instead of a JPanel. In its `paintComponent` method, we load the background image. In the constructor, when we add things, instead of adding them directly to the window, we add them onto the drawing panel. Here is a basic example:

```
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.*;

public class BackgroundImageDemo
{
    private JFrame frame;
    private DrawingPanel mainPanel;
    private BufferedImage bgImage;
    private JLabel label;
    private JButton button;

    public BackgroundImageDemo()
    {
        frame = new JFrame("Background Image Demo.");
        mainPanel = new DrawingPanel(100,100);
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new BackgroundImageDemo();
            }
        });
    }

    private void initialize()
    {
        try {
            bgImage = ImageIO.read(new File("someImage.jpg"));
        } catch (IOException e) {}
        label = new JLabel("Hello world");
        button = new JButton("Click me");
        mainPanel.add(label);
        mainPanel.add(button);
    }

    private class DrawingPanel extends JPanel
    {
        public DrawingPanel(int width, int height)
        {
            super();
            setPreferredSize(new Dimension(width, height));
        }

        @Override
        public void paintComponent(Graphics g)
        {
            super.paintComponent(g);
            g.drawImage((Image)bgImage, 0, 0, this.getWidth(), this.getHeight(),
                        0, 0, bgImage.getWidth(), bgImage.getHeight(), null, null);
        }
    }
}
```

5.15 Example GUI programs

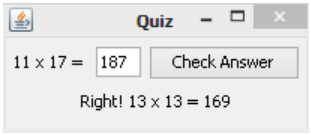
A multiplication quiz

Here is a graphical multiplication quiz. Players are given randomly generated multiplication problems to answer.

```
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.*;

public class MultiplicationQuiz
{
    private JFrame frame;
    private JPanel mainPanel;
    private JPanel topPanel;
    private JPanel bottomPanel;
    private JLabel promptLabel;
    private JLabel responseLabel;
    private JButton guessButton;
    private JTextField entry;
    private int num1;
    private int num2;

    public MultiplicationQuiz()
    {
        frame = new JFrame("Quiz");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new MultiplicationQuiz();
            }
        });
    }

    private void initialize()
    {
        // Generate the initial multiplication problem.
        Random random = new Random();
        num1 = random.nextInt(15) + 5;
        num2 = random.nextInt(15) + 5;

        // Create the GUI elements.
        promptLabel = new JLabel(num1 + " x " + num2 + " =");
        responseLabel = new JLabel();

        guessButton = new JButton("Check Answer");
        entry = new JTextField();
        entry.setColumns(3);

        guessButton.addActionListener(new ButtonHandler());
        entry.addActionListener(new ButtonHandler());

        // Lay things out on the screen.
        mainPanel.setPreferredSize(new Dimension(200,60));
```

```
            mainPanel.setLayout(new GridLayout(2,1));

            topPanel = new JPanel();
            topPanel.add(promptLabel);
            topPanel.add(entry);
            topPanel.add(guessButton);

            bottomPanel = new JPanel();
            bottomPanel.add(responseLabel);

            mainPanel.add(topPanel);
            mainPanel.add(bottomPanel);
        }

        private class ButtonHandler implements ActionListener
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                // Read guess from entry box, check if it's right, generate a new problem.
                int userGuess = Integer.parseInt(entry.getText());
                if (userGuess == num1 * num2)
                    responseLabel.setText("Right! " + num1 + " x " + num2 + " = " + num1*num2);
                else
                    responseLabel.setText("Wrong. " + num1 + " x " + num2 + " = " + num1*num2);
                Random random = new Random();
                num1 = random.nextInt(15) + 5;
                num2 = random.nextInt(15) + 5;
                promptLabel.setText(num1 + " x " + num2 + " = ");
                entry.setText("");
            }
        }
    }
```

Here is what the program looks like on my machine:



A few things to note about the code: First, we create two panels, `topPanel` and `bottomPanel` to help with laying things out nicely on the screen. The top panel holds the question, textfield, and button. The bottom panel holds the response. Second, notice that most of the interesting code goes into to the ActionListener. GUI programming is *event-driven*, which is to say that the program sits around and waits for things, like button presses, to happen, and then does things once such an event is triggered. In our code, whenever the button is pressed or enter is pressed in the textfield, we jump into the ActionListener, which determines if the answer is correct and then generates a new problem. We use class variables to remember the random numbers that we generate.

A virtual keyboard

Here is a keyboard made of buttons. Clicking the buttons adds text to a label.

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;
import javax.swing.*;

public class VirtualKeyboard
{
    private JFrame frame;
    private JPanel mainPanel;
    private JPanel buttonPanel;
```

```
    private List<JButton> buttons;
    private JLabel outputLabel;
    String text; // Keeps track of the text built up by clicking buttons.

    public VirtualKeyboard()
    {
        frame = new JFrame("Virtual Keyboard");
        mainPanel = new JPanel();
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new VirtualKeyboard();
            }
        });
    }

    private void initialize()
    {
        text = "";

        mainPanel.setLayout(new BorderLayout());
        buttonPanel = new JPanel(new GridLayout(5,6));

        outputLabel = new JLabel();

        // Creates all the letter buttons on the screen
        buttons = new ArrayList<JButton>();
        String alphabet = "abcdefghijklmnopqrstuvwxyz <";
        for (int i=0; i<alphabet.length(); i++)
        {
            JButton button = new JButton(alphabet.substring(i,i+1));
            button.addActionListener(new ButtonListener());
            buttons.add(button);
            buttonPanel.add(button);
        }

        mainPanel.add(buttonPanel, BorderLayout.CENTER);
        mainPanel.add(outputLabel, BorderLayout.SOUTH);
    }

    private class ButtonListener implements ActionListener
    {
        @Override
        public void actionPerformed(ActionEvent e)
        {
            // First get the letter pressed.
            String letter = ((JButton)(e.getSource())).getText();

            // if it's a backspace, delete last character; otherwise add letter to text.
            if (letter.equals("<") && text.length() > 0)
                text = text.substring(0, text.length()-1);
            else if (!letter.equals("<"))
                text += letter;
            outputLabel.setText(text);
        }
    }
}
```

Here is what the program looks like on my machine:

The buttons are created using a loop in the `initialize` method. Other than that, the only interesting part of the code is the ActionListener. We use `e.getSource` to figure out which button was clicked, and use `getText` to get the letter that is stored in the button's text. That letter is added to a class variable that holds the text, unless the backspace key is pressed, in which case the letter is removed from that variable.

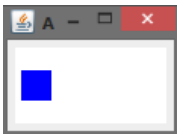
A simple animation demonstration

Here is a program that uses a timer and a drawing panel to animate a box moving across the screen.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class AnimationDemo
{
    private JFrame frame;
    private DrawingPanel mainPanel;
    private Timer timer;
    private int xpos; // Keeps track of box's horizontal position on screen

    public AnimationDemo()
    {
        frame = new JFrame("Animation Demonstration");
        mainPanel = new DrawingPanel(100, 50);
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new AnimationDemo();
            }
        });
    }

    private void initialize()
    {
        timer = new Timer(10, new TimerHandler());
        timer.start();
    }

    private class TimerHandler implements ActionListener
    {
        @Override
        public void actionPerformed(ActionEvent e)
        {
            // Every 10 milliseconds, this moves the box 2 pixels forward, wrapping around
            // at the end of the screen. We call repaint() to force a screen update.
```

```
                xpos += 2;
                if (xpos > mainPanel.getWidth())
                    xpos = 0;
                mainPanel.repaint();
            }
        }

        private class DrawingPanel extends JPanel
        {
            public DrawingPanel(int width, int height)
            {
                super();
                setBackground(Color.WHITE);
                setPreferredSize(new Dimension(width, height));
            }

            @Override
            public void paintComponent(Graphics g)
            {
                // This part draws the rectangle on the screen.
                super.paintComponent(g);
                g.setColor(Color.BLUE);
                g.fillRect(xpos, 15, 20, 20);
            }
        }
    }
```

Here is what the program looks like on my machine:



We use a class variable to keep track of the box's position. That position is updated every 10 milliseconds in the timer handler. That handler calls the drawing panel's `repaint` method, which internally calls the `paintComponent` method to draw the box on the screen.

One other thing to note is that we've made the `mainPanel` into a DrawingPanel. We could have just created a separate DrawingPanel and put that onto `mainPanel`, but that seemed unnecessary since we don't have any other buttons or other widgets.

A box-avoiding game

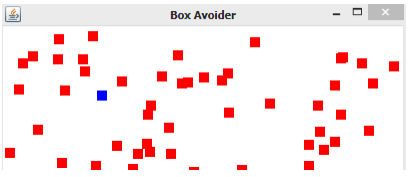
The following program is a simple interactive game. The player moves a blue box around on the screen trying to avoid randomly appearing red boxes.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import javax.swing.*;

public class BoxAvoider
{
    private JFrame frame;
    private JPanel mainPanel;
    private Timer timer;

    // Coordinates of the player
```

```
private int xpos;
private int ypos;

// Lists of coordinates of each of the randomly appearing blocks
private List<Integer> blockX;
private List<Integer> blockY;

public BoxAvoider()
{
    frame = new JFrame("Box Avoider");
    mainPanel = new DrawingPanel(400, 400);
    try {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    } catch (Exception e) {}
    initialize();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(mainPanel);
    frame.setLocationByPlatform(true);
    frame.pack();
    frame.setVisible(true);
}

public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new BoxAvoider();
        }
    });
}

private void initialize()
{
    xpos = ypos = 0;
    blockX = new ArrayList<Integer>();
    blockY = new ArrayList<Integer>();

    timer = new Timer(100, new TimerHandler());
    timer.start();
    mainPanel.setFocusable(true);
}

private class TimerHandler implements ActionListener
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        // Every 100 ms this is called to randomly add another box onto the screen.
        Random random = new Random();
        int x = random.nextInt(mainPanel.getWidth()-10);
        int y = random.nextInt(mainPanel.getHeight()-10);

        blockX.add(x);
        blockY.add(y);
        mainPanel.repaint();
    }
}

private class DrawingPanel extends JPanel implements KeyListener
{
    public DrawingPanel(int width, int height)
    {
        super();
        setBackground(Color.WHITE);
        setPreferredSize(new Dimension(width, height));
        addKeyListener(this);
    }

    @Override
    public void paintComponent(Graphics g)
    {
        // Draws the player and the random red blocks
```

```
            super.paintComponent(g);
            g.setColor(Color.BLUE);
            g.fillRect(xpos, ypos, 10, 10);

            g.setColor(Color.RED);
            for (int i=0; i<blockX.size(); i++)
                g.fillRect(blockX.get(i), blockY.get(i), 10, 10);
        }

        @Override
        public void keyPressed(KeyEvent e)
        {
            // The arrow keys move the player around.
            int key = e.getKeyCode();
            if (key == KeyEvent.VK_LEFT)
                xpos -= 2;
            else if (key == KeyEvent.VK_RIGHT)
                xpos += 2;
            else if (key == KeyEvent.VK_UP)
                ypos -= 2;
            else if (key == KeyEvent.VK_DOWN)
                ypos += 2;

            // Loop through the boxes and check to see if we've crashed into anything.
            for (int i=0; i<blockX.size(); i++)
            {
                int x = blockX.get(i);
                int y = blockY.get(i);

                if (((xpos+10>=x && xpos+10<=x+10) || (xpos>=x && xpos<=x+10)) &&
                    ((ypos+10>=y && ypos+10<=y+10) || (ypos>=y && ypos<=y+10)))
                {
                    timer.stop();
                    JOptionPane.showMessageDialog(new JFrame(), "You lose");
                    System.exit(0);
                }
            }

            // Player wins if they get to bottom right corner.
            if (xpos>=this.getWidth()-10 && ypos>=this.getHeight()-10)
            {
                timer.stop();
                JOptionPane.showMessageDialog(new JFrame(), "You win!");
                System.exit(0);
            }

            this.repaint();
        }

        @Override public void keyReleased(KeyEvent e) {}
        @Override public void keyTyped(KeyEvent e) {}
    }
}
```

Here is what the program looks like on my machine:



Briefly, the program works as follows: We have variables xpos and ypos to keep track of the player's location and lists to keep track of the locations of each of the randomly appearing red blocks. We set a timer so that every 100 milliseconds a new red block is generated. Then in the key listener, we check for arrow key presses and move the player accordingly. The trickiest part of the code is checking for a collision between the player and one of the randomly appearing boxes.

Mouse demonstration

Here is a demonstration of how to work with mouse input. The program creates a rectangle that can be moved around and resized.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseWheelEvent;
import java.awt.event.MouseWheelListener;
import javax.swing.*;

public class MouseDemo
{
    private JFrame frame;
    private JPanel mainPanel;

    public MouseDemo()
    {
        frame = new JFrame("Mouse Demo");
        mainPanel = new DrawingPanel(300, 100);
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {}
        initialize();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(mainPanel);
        frame.setLocationByPlatform(true);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new MouseDemo();
            }
        });
    }

    private void initialize()
    {
        mainPanel.setFocusable(true);
    }

    private class DrawingPanel extends JPanel implements MouseListener, MouseMotionListener,
                                                          MouseWheelListener
    {
        private int mouseX, mouseY;
        private int boxX, boxY;
        private int shiftX, shiftY;
        private double size;
        private boolean dragRectangle;

        public DrawingPanel(int width, int height)
        {
            super();
            setBackground(Color.WHITE);
            setPreferredSize(new Dimension(width, height));
            shiftX = shiftY = 0;
            size = 20;
            boxX = boxY = 50;
            dragRectangle = false;
            addMouseListener(this);
            addMouseMotionListener(this);
            addMouseWheelListener(this);
        }
```

```
        @Override
        public void paintComponent(Graphics g)
        {
            super.paintComponent(g);
            boxX += shiftX;
            boxY += shiftY;
            g.fillRect(boxX-(int)size/2, boxY-(int)size/2, (int)size, (int)size);
        }

        @Override
        public void mouseWheelMoved(MouseWheelEvent e)
        {
            int notches = e.getWheelRotation();
            if (notches > 0)
                size *= (1 - .1 * notches);
            else
                size /= (1 + .1 * notches);
            repaint();
        }

        @Override
        public void mousePressed(MouseEvent e)
        {
            int x = e.getX();
            int y = e.getY();

            if (x>=boxX-size/2 && x<=boxX+size/2 && y>=boxY-size/2 && y<=boxY+size/2)
            {
                mouseX = x;
                mouseY = y;
                dragRectangle = true;
            }
        }

        @Override
        public void mouseReleased(MouseEvent e)
        {
            dragRectangle = false;
        }

        @Override
        public void mouseDragged(MouseEvent e)
        {
            if (dragRectangle)
            {
                int x = e.getX();
                int y = e.getY();

                shiftX = x - mouseX;
                shiftY = y - mouseY;

                mouseX = x;
                mouseY = y;

                repaint();
            }
        }

        @Override public void mouseEntered(MouseEvent e) {}
        @Override public void mouseClicked(MouseEvent e) {}
        @Override public void mouseMoved(MouseEvent e) {}
        @Override public void mouseExited(MouseEvent e) {}
    }
}
```

To explain what is going on, first the listeners require us to implement a variety of different methods specifying what to do when the mouse is clicked, released, dragged, etc. We choose to do some of these but not others. In terms of the variables, `boxX`, `boxY` and `size` specify the location and size of the rectangle. The other variables are used for dragging.

The way dragging works is in the `mouseDragged` method, we get the current mouse coordinates and subtract the old ones from them to find out how much to move the box by. The `repaint` method at the end of the `mouseDragged` method essentially calls the `paint` method to redraw the panel. The `dragRectangle` variable is used to make sure that we can only drag while the mouse pointer is on the rectangle.

The mouse wheel code uses the `getWheelRotation` method to determine how many "notches" the wheel has rotated through and uses that to resize the component by a certain percentage.

5.16 Further GUI programming

One thing to get used to with GUI programming is that it is event-driven. What this means is that the program is basically running in a perpetual loop, waiting for things to happen. Almost all of your program's interesting code is contained in event listeners, like ActionListeners that do something once something like a button click or a keypress occurs.

Programs that would have required a while loop when run in the console may not require one when run as a GUI-based program. For instance, a guess-a-number program for the console would require a while loop that runs until the player guesses the number or runs out of turns. When we write it as a GUI-based program, we don't need the while loop. The GUI program is already essentially in an infinite loop, waiting for button clicks and what-not. One of those buttons would have an event listener where we would check to see if the guess is right or not and exit the program if they player runs out of guesses.

We have just scratched the surface here to give you some simple tools to create graphical programs. There are a lot of different approaches to doing things graphically and we deliberately did not chosen the more sophisticated ones in the interest of keeping things simple. There are lots of web sources where you can learn more if you want.

Chapter 6

Common Gotchas

6.1 Simple debugging

There is a debugger that comes with the JDK that you can use to examine the values of your variables while your program is running. Another technique is to use print statements in your code. If your code is not behaving, add print statements to print out the values of some of the variables or just to see if a section of code is being reached. This can be a fast way to isolate where a problem lies.

6.2 Common exceptions

Index out of bounds exceptions

If you work with strings, arrays, and lists, sooner or later you will accidentally try to work with an index beyond the bounds of the object. For instance, if you have a 10-character string, and you try to access the character at index 15, Java will raise a runtime exception. You will see a message like the following:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 15
at java.lang.String.charAt(Unknown Source)
        at Test.main(Test.java:13)
```

It tells you the bad index (15 in this case) that you tried to access. That is incredibly important for debugging. It also tells the function, `charAt`, that caused the problem and the line number where the exception occurred. You can usually click on that line number to jump to that location in your code.

Null pointer exceptions

One of the most common errors in Java programs is a the null pointer exception. Here is an example:

```
import java.util.List;

public class MyClass
{
    List<String> list;

    public MyClass()
    {
        list.add("a");
    }

    public static void main(String[] args)
    {
        MyClass mc = new MyClass();
    }
}
```

This leads to the following error message:

```
Exception in thread "main" java.lang.NullPointerException
        at MyClass.<init>(MyClass.java:9)
        at MyClass.main(MyClass.java:14)
```

The problem here is that we declare a list variable, but we did not actually create a new object. There is no `new` statement. So there is no list in memory that the `list` variable is pointing to. Instead of storing the memory location of a list somewhere in memory, that variable just stores the special value `null`. The solution is to make sure you always set up your objects with a `new` statement. In particular, the first line of the constructor should be `list = new ArrayList<String>()`.

Number format exception

Number format exceptions happen when you try to use `Integer.parseInt`, `Double.parseDouble`, etc. on something that can't be converted into the appropriate type. For example, `Integer.parseInt("23x")` will cause the following:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "23x"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at Test.main(Test.java:11)
```

The reason for this is "23x" cannot be converted to an integer because of the "x". Number format exceptions can happen when you are getting text from a user and they enter something wrong. They can also happen when you make a mistake converting something that you've read from a file. For instance, if you forget that there is a space and try to convert `" 23"`, Java will raise a number format exception.

Also, trying to do `Integer.parseInt("23.2")` will raise the a number format exception. Using `Double.parseDouble("23.2")` could fix the problem.

Concurrent modification exception

Suppose we want to remove all the zeros from a list that contains some zeros. The following attempt at this will cause Java to raise a concurrent modification exception:

```
for (int x : list)
    if (x == 0)
        list.remove(x);
```

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
        at java.util.ArrayList$Itr.next(ArrayList.java:851)
        at Test.main(Test.java:13)
```

The problem is that we are removing things from the list as we are looping through it. This messes up the looping process. Here is a similar problem:

```
for (int i=0; i<list.size(); i++)
    if (list.get(i) == 0)
        list.remove(list.get(i));
```

To see why this won't work, suppose the list is `[0,0,1,2]`. The first time through the loop `i=0` and we remove the first element of the list, leaving `[0,1,2]`. The second time through the loop, `i=1`, which means we are looking at the item at index 1 in the modified list, which is a 1, so we end up missing the 0. Here is one way to fix both of these problems:

```
int i = 0;
while (i < list.size())
{
    if (list.get(i) == 0)
        list.remove(list.get(i));
    else
        i++;
}
```

6.3 Lengths of strings, arrays, and lists

One little inconsistency in Java is that the ways to get the length of a string, an array, or a list are all different. Shown below are the three ways:

```
string.length()  // length of a string
array.length     // length of an array (no parens)
list.size()      // length of a list
```

6.4 Misplaced semicolons

The following code is supposed to print `Hello` if `x` is less than 5:

```
if (x<5);
    System.out.println("Hello");
```

But it doesn't work. The reason is the unintended semicolon after the if statement. The problem is that this is valid Java code and it will compile without an error. The same thing can happen with for loops:

```
for (int i=0; i<10; i++);
    System.out.println("Hello");
```

These errors can be really frustrating and difficult to spot, so be on the lookout for them if your code is behaving strangely. In particular, if you are absolutely positive that your code should be working, but for some reason it seems to be skipping over a for loop, if statement, etc., then check to make sure there isn't an accidental semicolon.

6.5 Characters and strings

Java contains both a data type called `char` that holds a single character and a data type called `String` that holds multiple characters. The `char` data type is a primitive data type that is stored efficiently in memory and has no methods associated with it. The `String` data type has methods associated with it.

Single quotes are used for characters and double quotes for strings, like below.

```
char c = 'a';
String s = "a";
```

You will probably do most of your work with strings, but characters are sometimes useful. Some string methods, like `charAt`, return a char. This can cause a common problem. Suppose you are building up a string character by character and you try the following to capitalize one of those characters.

```
newString += s.charAt(i).toUpperCase()
```

The result will be an error because `charAt` returns a char and `toUpperCase` is a String method, not a char method. Three possible solutions are shown below:

```
newString += s.substring(i,i+1).toUpperCase()
newString += String.valueOf(s.charAt(i)).toUpperCase();
newString += (""+s.charAt(i)).toUpperCase();
```

6.6 Counting problems

A common mistake when counting is to forget to reset the counter. Here is an example. Suppose we have a list of words called `words` and for each word, we want a count of how many times the letter *i* occurs in the word, and we'll add those counts to a list called `counts`. Here is the code:

```
for (String word : words)
{
    int count = 0;
    for (int i=0; i<words.length(); i++)
```

```
        {
            if (word.charAt(i) == 'i')
                count += 1;
        }
        counts.add(count);
    }
```

The common mistake would be just declare `count = 0` outside of both loops. But we need the counter to be reset after each new word, so it needs to be inside the words loop.

6.7 Problems with scanners

Reading multiple items

Consider the following code that reads an integer followed by some text from the user:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter a number: ");
int x = scanner.nextInt();
System.out.print("Enter some text: ");
String s = scanner.nextLine();
```

It actually won't work. What happens is when the user enters a number and presses enter, a newline character is stored, corresponding to the enter key. The call to `nextInt` does not read that character, so it is still there. The subsequent call to nextLine reads that character, and doesn't even allow the user to type anything in. One solution to this problem is to refresh the scanner with `scanner = new Scanner(System.in)` before the call to `nextLine`.

Resetting scanners when reading through files

A related problem when using a scanner to read through a text file more than once is forgetting to reset the scanner. After you read through a file, the `Scanner` object points to the end of the file. If you need to read back through the file, it is necessary to set the scanner to point back to the start of the file. One way to do that is to redeclare the scanner.

`Scanner.nextLine()` vs `Scanner.next()`

When asking the user to enter a string, use `Scanner.nextLine()`:

```
System.out.println("Enter your name: ");
String name = Scanner.nextLine();
```

It's not recommended to use `Scanner.next()` here as it will only read up to the first space.

6.8 Problems with logic and if statements

Mixing up ANDs with ORs

A common mistake is having && where you need || or vice-versa. For instance, if we want to do something as long as x is not a 3 or a 4, the proper if statement is shown below:

```
if (x != 3 && x != 4)
```

Sometimes these mistakes show up in conditions for while loops. That's something to look for if you have a while loop that seems like it should be running but isn't.

Also, be careful of precedence: && has a higher precedence than ||. The expression `A || B && C` is treated as `A || (B && C)`. A safe way to do things is to always use parentheses.

== versus equals()

When comparing two strings, you must use the `equals` method. Comparing with `==` might sometimes work, but sometimes not. The proper way to check if the string `s` is equal to the string `"abc"` is:

```
if (s.equals("abc"))
```

if instead of else if

Consider the following program:

```
if (grade>=90 && grade<=100)
    System.out.println("A");
if (grade>=80 && grade<90)
    System.out.println("B");
else
    System.out.println("C or lower");
```

If `grade` is in the 90s, then the program will print out `"A"` and it will also print out `"C or lower"`. The problem is the `else` block only goes with the second if statement. The first `if` statement is totally separate. This is easy to fix. Just change the second `if` to `else if`.

A common mistake with booleans

Here is a mistake I have seen a lot. Say we want to write a `contains` method that returns whether a string contains a certain character. Here is an incorrect way to do that:

```
public static contains(String s, char c)
{
    for (int i=0; i<s.length(); i++)
    {
        if (s.charAt(i) == c)
            return true;
        else
            return false;
    }
}
```

The problem is that if the first character does not match the character `c`, then the program will return `false` right away and will miss occurrences of `c` later in the string. A correct way to write this method is shown below:

```
public static contains(String s, char c)
{
    for (int i=0; i<s.length(); i++)
    {
        if (s.charAt(i) == c)
            return true;
    }
    return false;
}
```

6.9 Problems with lists

Combining two types of loops

Consider the following:

```
for (int i : list)
    // do something with list.get(i)
```

This is a mistake as it is combining two types of loops. Instead, use either of the loops below:

```
for (int x : list)
```

```
for (int i=0; i<list.size(); i++)
```

```
        // do something with x                    // do something with list.get(i)
```

The list remove method

Say we're working with a list of integers called `list`. If we call `list.remove(3)`, does it remove the first occurrence of the integer 3 in the list or does it remove the integer at index 3 in the list? It turns out to be the latter. If we want the former, we must use `list.remove((Integer)3)`. Basically, if `remove` is called with an **`int`** argument, it assumes the argument is an index, while if it is called with an object (including the `Integer` class), then it removes the first occurrence of that object.

6.10 Functions that should return a value but don't

Suppose we have a function that converts Celsius to Fahrenheit temperatures, like below:

```
public static void convertToFahrenheit(double c)
{
    System.out.println(9/5.0*c + 32);
}
```

There's nothing wrong with this function except that it just prints the converted value and doesn't allow us to do anything with it. Maybe we want to convert a temperature to Fahrenheit and then use the converted temperature in another calculation, like below:

```
double boilingFreezingDiff = convertToFahrenheit(100) - convertToFahrenheit(0);
```

The function we've written won't allow us to do that. Instead, we should write the function like below, where it returns a value:

```
public static double convertToFahrenheit(double c)
{
    return 9/5.0*c + 32;
}
```

6.11 Problems with references and variables

Copying lists

Reread Section 4.9 carefully. In particular, note that the following will *not* make a copy of a list called `list`:

```
List<Integer> copy = list;
```

It just makes an alias, which is to say that both `copy` and `list` refer to the same list in memory. Instead, use the following:

```
List<Integer> copy = new ArrayList<Integer>(list);
```

This creates a brand new copy of the list in memory. The same thing applies to copying any other type of objects. To copy them, you will either need to rely on a `copy` method provided by the class or manually create a new object.

Passing lists and objects to functions

Consider the following function that returns how many zeros there are at the start of a list:

```
public int numZerosAtStart(List<Integer> list)
{
    int count = 0;
    while (list.size() > 0 && list.get(0) == 0)
        list.remove(0);
    return count;
```

```
}
```

It works properly, but it has a serious problem—it messes up the caller's list by removing all the zeros from the front. This is because the parameter `list` is actually an alias for the list that the caller passes to the function, so any changes using `list` will affect the caller's original list, which is not something they would likely appreciate. To fix this problem, either make a copy of list at the start of the function or rewrite it so that it doesn't modify the list.

Accidental local variables in constructors

Suppose we have the following class:

```
public class MyClass
{
    public int x;
    public MyClass()
    {
        int x = 0;
    }
}
```

This does not set the x field to 0. Instead, it creates a local variable called x in the constructor and sets that to 0. The class variable is not affected. The solution is to just say `x=0` or `this.x=0` in the constructor.

6.12 Numbers

Integers and floating point numbers in Java are stored in binary. Their size is also limited so that calculations can be done efficiently in hardware. This can lead to some problems if you are not careful.

Integers

An `int` in Java is stored with 4 bytes. A byte consists of 8 bits, and each additional bit effectively doubles the amount of values that can be stored, so a total of $2^{32} \approx 4$ billion possible integers can be stored in 4 bytes. The actual range of values is from about -2 billion to +2 billion (if you ever need the exact values, you can get them from `Integer.MAX_VALUE` and `Integer.MIN_VALUE`). These are useful if you need a value that is guaranteed to be greater than or less than any other value.

If you try to store a value greater than the maximum possible, you will end up with garbage. For instance, if we do the following, we will end up with x equal to -294967296, which is obviously not right:

```
int x = 2000000000 * 2;
```

This is sometimes called an *integer overflow*. As another example, `int x = Integer.MAX_VALUE + 1` will set x to -1. The reason for these bizarre results has to do with the way integers are stored in memory. Once we overflow past the maximum value, the sign bit (+/-) accidentally gets overwritten.

If you need larger values than an `int` can provide, use a `long`, which uses 8 bytes, for a range of values between about -9×10^{18} to 9×10^{18}.

If you need still larger values, use the `BigInteger` class. That class can handle arbitrarily large integers (like things several hundred digits in size). However, `BigInteger` cannot take advantage of the underlying hardware in the same way that `int` and `long` can, so it is much slower.

Floating Point values

A Java `double` is stored using 8 bytes. Internally, it is stored using a format called the IEEE Standard 754. The number is stored in a form similar to scientific notation that gives about 15 digits of precision and a wide range of exponents (from about 10^{-300} for numbers close to 0 to about 10^{300} for really large numbers). 15 digits of

precision is sufficient for most purposes, but Java does provide a `BigDecimal` class in the event that you need more. However, just like `BigInteger`, it can be slow.

One common problem with using binary to store floating point numbers is that many numbers that we can represent exactly in our decimal system cannot be represented exactly in binary. For instance, 1/5 can be represented as .2 in our decimal system, but its representation is binary is the repeating expansion $.001100110011\cdots$. That repeating expansion has to be cut off at some point, which means a computer can't store the exact value of .2. In fact, it stores the binary number equivalent to 0.20000000000000001. Similarly, .4 is actually 0.40000000000000002 and .6 is 0.59999999999999998.

This can have a number of undesirable effects:

1. Sometimes, when printing out a floating point number, you might expect to see .6, but you will actually see 0.59999999999999998. This can be managed using formatting codes (like with `printf`) to limit the number of decimal places.

2. Small errors can actually accumulate. For example, we might expect the following code to print out 200000, but it actually prints out 200000.00000266577.

   ```
   double sum = 0;
   for (int i=0; i<1000000; i++)
       sum += .2;
   System.out.println(sum);
   ```

   This is something to be aware of. If you need perfect precision, use the `BigDecimal` class.

3. Comparing floating points using == is a bad idea. It does work sometimes, but it often fails. For instance, both `x` and `y` in the code below, mathematically speaking, should equal .2, but because of how floating point numbers are stored, the values don't agree.

   ```
   double x = 1/5.0;
   double y = 1-.8;

   if (x == y)
       // do something
   ```

   The way to fix it is to change the if statement to something like the one below:

   ```
   if (Math.abs(x-y)<.0000000001)
   ```

   The code above considers two doubles as equal provided they are within .0000000001 of each other. That value can be adjusted up or down, depending on the precision needed. Another option would be to use the `BigDecimal` class.

Chapter 7

A Few Other Topics

7.1 Java's history

Java was developed in the early 1990s by a team at Sun Microsystems led by James Gosling. It was originally designed to be used in embedded systems, like in mobile or home electronics. As such, it was a good fit with the early internet. Its popularity exploded in the second half of the 1990s and it is still (as of 2015) one of the most popular and widely used programming languages.

Java's syntax was based off the syntax of the C programming language. Once you know Java, it is fairly straightforward to learn other languages with a similar syntax, including C, C++, C#, and Objective C, all of which are (as of this writing) in wide use.

Java is used in web development, Android development, and the development of desktop applications.

7.2 The Java Virtual Machine (JVM)

Originally, the way programming languages worked is that code was compiled directly into machine language. The program would have to be tweaked to run on different CPUs and on different operating systems. The approach Java uses is that Java code is compiled to an intermediate language called Java byte code, and that byte code is translated into machine code by the JVM. The JVM handles all the low level details so that the programmer can just write one program and have it run on a variety of operating systems. The JVM is also there while your program is running helping to manage memory and provide debugging information, among other things.

The JVM has become popular enough that a number of other languages now compile directly to Java byte code to take advantage of the JVM and all the libraries that have been written for Java. Below is an example of the Java byte code generated from a program that prints the integers from 1 to 100.

```
public class Numbers extends java.lang.Object{
public Numbers();
  Code:
   0:   aload_0
   1:   invokespecial   #8; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   iconst_1
   1:   istore_1
   2:   goto    15
   5:   getstatic       #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   8:   iload_1
```

```
    9:   invokevirtual   #22; //Method java/io/PrintStream.println:(I)V
    12:  iinc    1, 1
    15:  iload_1
    16:  bipush  100
    18:  if_icmple       5
    21:  return
}
```

7.3 Running your programs on other computers

If you want to give others your Java programs to use, they will need Java installed on their computers. They just need the JRE and not the full JDK that you need to develop Java programs. Chances are they already have the JRE.

You will probably want to give them a .jar file. To do this in Eclipse, select the file or project you want to distribute. Right click, go to `Export` and then select `Java` and then `Create Runnable JAR`. Choose a location to save the file and you're done. There is a similar process in IntelliJ. It's a little easier on NetBeans, since NetBeans automatically creates .jar files and saves them in the `dist` subdirectory of your project.

7.4 Getting help on Java

One of the best ways to get help is to just use the features of your IDE. The entire Java API documentation is also online. Google and StackOverflow are also good since chances are whatever problem you are having is a problem someone else has already encountered.

7.5 Whitespace, braces, and naming conventions

There are certain conventions regarding whitespace, names, and other things that people try to follow to make reading programs easier. Here are a few things:

- Blocks of code that make up if statements, for loops, functions, etc. are indented (usually two or four spaces).
- Long lines of code (greater than 80 characters) should be broken up into multiple lines.
- There are a couple of common brace styles in use. There's the one I use in these notes and then there's one that looks like below:

  ```
  for (int i=0; i<10; i++) {
      System.out.println("Hi");
      System.out.println("Hi again");
  }
  ```

  I think more people use this way than the way I go with. Go with whichever way you like better, but know that people, especially on the internet, feel especially strongly about one way or the other and are eager to argue about why their way is the one, true way.
- If there is only one statement that makes up the body of a loop or if statement, the braces are optional. It's generally recommended that you use them, even if they are optional (though I personally don't follow that rule).
- Class and interface names are capitalized, variable names and package names are lowercase, and constant names are all uppercase.
- If a name consists of multiple words, the first letter of each word (except possibly the first) is capitalized, like `newArraySize` or `maxValue`. Use underscores to separate words in constant names.

To see all the conventions, do a web search for "Java code conventions."

Chapter 8

Exercises

Here are exercises for much of the material covered in the previous chapters. Exercises are ordered by topic. Within each section exercises start out relatively straightforward and get a little more challenging toward the end. Each section builds on the previous, so that in the section on string exercises, loops and if statements are required for some problems. See http://faculty.msmary.edu/heinold/java_text_files.zip for text files needed for some of the exercises.

8.1 Exercises involving variables, loops, and if statements

1. Write a program that declares four integer variables equal to 2, 4, 5, and 10 and prints out their average (which should be 5.25).
2. Write a program that uses a for loop to obtain the output below on the left. Then write a program using a for loop to obtain the output on the right.

```
A                    1. A
A                    2. A
A                    3. A
A                    4. A
A                    5. A
```

3. The Body Mass Index, BMI, is calculated as

$$\text{BMI} = \frac{703w}{h^2},$$

where w is the person's weight in pounds and h is the person's height in inches. Write a program that asks the user for their height their weight and prints out their BMI.

4. Write a program that asks the user how many credits they have taken and prints their class standing. The credit ranges are 0-23 for freshmen, 24-53 for sophomores, 54-83 for juniors, and 84 and over for seniors.
5. The power in Watts across part of an electrical circuit is given by $P = \frac{V^2}{R}$, where V is the voltage and R is the resistance. Write a program that asks the user to enter the voltage and resistance. The program should then print out what the corresponding power is. It should also print out a warning if the power turns out to be greater than 1000 watts.
6. Write a program that uses a for loop to print out the numbers 8, 11, 14, 17, 20, ..., 83, 86, 89.
7. Write a program that uses a loop to ask the user to enter 10 numbers. Then the program should print the average of those 10 numbers.
8. This is a very simple billing program. Ask the user for a starting hour and ending hour, both given in 24-hour format (e.g., 1 pm is 13, 2 pm is 14, etc.). The charge to use the service is $5.50 per hour. Print out the user's total bill. You can assume that the service will never be used for more than 24 hours. Be careful to take care of the case that the starting hour is before midnight and the ending time is after midnight.

9. A company charges $12 per item if you buy less than 10 items. If you buy from 10 and 99 items, the cost is $10 per item. If you buy 100 or more items the cost is $7 per item. Write a program that asks the user how many items they are buying and prints the total cost.

10. Write a program that prints out on separate lines the following 1000 items: `file0.jpg`, `file1.jpg`, ..., `file999.jpg`, except that it should not print anything when the number ends in 8, like `file8.jpg`, `file18.jpg`, etc. [Hint: if `i % 10` is not 8, then `i` doesn't end in 8.]

11. It is possible to compute square roots using just multiplication, addition, and division. To estimate the square root of the number a to very high accuracy, start by setting $x = 2.0$. Then replace x with the average of x and a/x. Then replace this new x with the average of x and a/x (using the new x). Repeat this 50 times and print out the end result.

12. This problem is about finding the day of the week of any date since about 1583. Ask the user to enter the year y, month m, and day d separately. The following formulas give the day of the week:

$$p = \left\lfloor \frac{14-m}{12} \right\rfloor$$
$$q = y - p$$
$$r = q + \left\lfloor \frac{q}{4} \right\rfloor - \left\lfloor \frac{q}{100} \right\rfloor + \left\lfloor \frac{q}{400} \right\rfloor$$
$$s = m + 12p - 2$$
$$t = \left(d + r + \left\lfloor \frac{31s}{12} \right\rfloor \right) \bmod 7$$

    The $\lfloor\ \rfloor$ brackets indicate the floor function. In Java, you can do this simply by doing integer division. For instance, p can be calculated just by doing `(14-m)/12`, with m being an integer.

    The day of the week is given by t, with $t = 0$ corresponding to Sunday, $t = 1$ corresponding to Monday, etc. Please output your answer as a day name and not as a number.

13. Generally, a year is a leap year if it is divisible by 4. However, if it is also divisible by 100, then it is *not* a leap year, except in the case that it is actually divisible by 400, in which case it *is* a leap year. Write a program that asks the user for a year and prints out whether it is a leap year or not.

14. Write a program that asks the user to enter a value n, and then computes $(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) - \ln(n)$. The ln function is `Math.log` in Java.

15. Consider the following sequence: 1, 1, 3, 7, 17, 41, 99, 239, 577, 1393, ... The first two terms are 1 and each term in the sequence is obtained from the previous two terms by taking twice the previous term and adding it to the term before that. For example, $3 = 2 \cdot 1 + 1$, $7 = 2 \cdot 3 + 1$, and $17 = 2 \cdot 7 + 3$. In equation form, each term a_n is given by $a_n = 2a_{n-1} + a_{n-2}$. Write a program that asks the user for a value, n, and prints out the nth term of the sequence.

16. Consider the following sequence: 1, 2, 3, 2.0, 2.333, 2.444, 2.259, 2.345, ... The first three terms are 1, 2, 3 and each additional term in the sequence is obtained from the average of the previous three terms. For example, the fourth term is $(1 + 2 + 3)/3 = 2$, and the fifth term is $(2 + 3 + 2)/3 = 2.333$. In equation form, each term a_n is given by $a_n = (a_{n-1} + a_{n-2} + a_{n-3})/3$. Write a program that asks the user for a value, n, and prints out the nth term of the sequence.

17. Write a program that computes the factorial of a number. The factorial, $n!$, of a number n is the product of all the integers from 1 to n, including n. For instance, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. [Hint: Try using the multiplicative equivalent of the counting statement `count=count+1`].

18. Write a while loop that produces the exact same output as the for loop below.

```
for (int i=2; i<50; i++)
    System.out.print(i + " ");
```

19. Write a program that allows the user to enter any number of test scores. The user indicates they are done by entering in a negative number. Print how many of the scores are A's (90 or above). Also print out the average.

20. Write a program that asks the user to enter numbers from 1 to 10. The program should stop when the user enters a 5. After the loop is done, the program should print out a count of how many numbers were entered and print out *yes* or *no*, depending on whether the user entered any numbers less than 3.

21. The $3x + 1$ problem is one of the most famous unsolved problems in math. You start with an integer x. If x is even, divide it by 2 and if it is odd, compute $3x + 1$. Then do the same to this number, dividing it by 2 if it is even, and multiplying it by 3 and adding 1 if it is odd. Continue this process until you get a 1. For instance, $x = 11$ gives the sequence $11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$. It is conjectured that no matter what value of x you start with, you always end up with 1, but no one can prove it. Ask the user to enter a starting value, and print out the entire sequence on one line until the first 1.

22. Write a program that lets a the user play Rock-Paper-Scissors against the computer. There should be five rounds, and after those five rounds, your program should print out who won and lost or that there is a tie.

23. Write a program that given an amount of change less than $1.00 will print out exactly how many quarters, dimes, nickels, and pennies will be needed to efficiently make that change.

24. Write a program that finds and prints out all the solutions to the equation $x^2 - 2y^2 = 1$, where x and y are integers from 1 to 99.

25. In some board games, you roll four dice and have to pick two of them. For instance, if you roll 2, 3, 3, 5, and you needed to get a 7, you could pick the first and last dice in order to make 7. Write a program that asks the user for a specific sum from 2 to 12 and computes the probability that they will be able to get that sum. [You don't need to do any math. You can have the program determine the probability. The probability of a sum of 2 turns out to be 13.2% and the probability of a sum of 7 turns out to be 79.2%.]

26. Use loops to generate the following output:

```
0123456789
1234567890
2345678901
3456789012
4567890123
5678901234
6789012345
7890123456
8901234567
9012345678
```

27. Write a program that prints a triangle like the one below. The user should be allowed to specify the height. [Hint: You will need to use nested for loops here.)

```
*
**
***
****
```

28. Write a program that prints a diamond like the one below. The user should be allowed to specify the height.

```
  *
 ***
*****
 ***
  *
```

8.2 Exercises involving random numbers

29. Write a program that generates a random number, x, from 1 and 50; generates a random number, y, from 2 and 5; and computes x^y.

30. Write a program that generates and prints 100 random numbers from 50 to 60, all on the same line, separated by spaces.

31. Write a program that generates and prints 50 random zeroes and ones, all on the same line, with no spaces separating them.

32. Write a program that generates and prints 100 random numbers such that the first number comes from the range from 1 to 2, the second comes from 1 to 3, the third comes from 1 and 4, etc.

33. A Yahtzee is when you roll five dice and they all come out the same (five ones, five twos, etc.) Write a program that simulates rolling a set of five dice 1,000,000 times and outputs what percentage of those simulated rolls are Yahtzees.

    The percentage should be displayed to exactly three decimal places. Also, print out about how many rolls on average it takes to get a Yahtzee (you get this by doing 1,000,000 divided by the total number of Yahtzees).

34. Write a program that generates random floating point numbers from 1 to 10, each to two decimal places (e.g. 4.23, 3.39, 5.00).

35. Write the following text-based game. The player starts with 100 points. The program then picks a random number from 1 to 10 and asks the player to guess it. If they guess it right, they gain 100 points. If their guess is less than the number, then they lose 10 points. If their guess is greater, they lose 20 points. After the guess, print out the result of the guess and the player's amount of points. The computer then picks a new number and play continues in the same way until the player gets to 200 or more points (in which case they win) or 0 or less points (in which case they lose). Print out a message at the end indicating whether the player won or lost.

36. Program the following game. The player starts out with $50. They have to pick a number from 1 to 10. The computer then picks a random number from 1 to 10. If the player's number matches the computer's, then the player owes the computer $50 and the game ends. If not, the player's money doubles to $100 and they keep playing.

    This time the player and the computer pick numbers from 1 to 9, and if their numbers match, then the player owes the computer all the money ($100). Otherwise, the player's money doubles to $200 and the game continues, this time with numbers from 1 to 8. This process continues, with the range of numbers shrinking by 1 each turn, until the last turn with numbers only ranging from 1 to 2.

    If the player's and computer's numbers ever match, the game is over and the player has to pay the computer whatever the money amount has gotten to. At any point, instead of guessing a number, the player can type in 0 to quit the game and leave with their current winnings. Some sample output is below:

37. Consider a while loop that generates random numbers from 1 to 10 until a 6 comes up. Sometimes a 6 will come up right away and sometimes it will take quite a while. For instance, if we print out the numbers that the loop generates, we might see the sequence `2,3,1,3,6` or the sequence `2,1,9,9,2,2,10,8,9,6`. We are interested in how long it could possibly take a 6 to come up. Write a program that runs the while loop over and over again, a million times, and reports how long the longest sequence of numbers is.

8.3 String exercises

38. Write a program that asks the user to enter a string. The program should then print the following five things: the string converted to lowercase, how many characters are in the string, the location of the first `'z'`, the first three characters of the string, and the last three characters of the string.

39. Write a program that asks the user to enter a word and prints out whether that word contains any vowels.

40. IP addresses are important in computer networking. They consist of four numbers, each from 0 to 255, separated by dots. For instance, the IP address of www.google.com is 74.125.22.99. IP addresses of the following forms are considered special local addresses: 10.*.*.*, 192.168.*.*, 172.16.*.*. The stars can stand for any values from 0 to 255. Write a program that asks the user to enter an IP address and prints out whether it is a local address. You can assume that the user enters a valid address. [Hint: Consider the `startsWith` method.]

41. Write a program that does the following: It first asks the person to enter their first and last names (together on the same line). For simplicity, assume their first and last names are one word each. Then ask them for their gender (male/female). Finally ask them if they prefer formal or informal address. Then print a line that says hello to them, using Mr. or Ms. <last name> if they ask for formal address and just their first name otherwise. Here are two sample runs:

```
Name: Joe Montana
Gender (male/female): male
Formal/Informal: informal
Hello, Joe.
```

```
Name: Grace Hopper
Gender: Female
Formal/Informal: Formal
Hello, Ms. Hopper.
```

42. Write a program that asks the user to enter a string, converts that string to lowercase, removes all punctuation from it, and prints out the result. Here are the seven punctuation characters to remove: `? ! " . , ; :`

43. Ask the user to enter a string of at least 5 characters. Then change the third letter of the string to an exclamation point and print out the result.

44. Ask the user to enter a course code, like MATH 247 or CMSCI 125. The code consists of a department code followed by a course number. If the course number is not in the range from 100 to 499, then output that the user's entry is invalid.

45. People often forget closing parentheses when entering formulas. Write a program that asks the user to enter a formula and prints out whether the formula has the same number of opening and closing parentheses.

46. Write a program that asks the user to enter a height in the format *feet*`'`*inches*`"` (like `5'11"` or `6'3"`. Add 4 inches to the height and print the result in the *feet*`'`*inches*`"` format. For example, if the user enters `5'11"`, the result should be `6'3"`.

47. Write a program that asks the user to enter a word that contains the letter `'a'`. The program should then print the following two lines: On the first line should be the part of the string up to and including the first `'a'`, and on the second line should be the rest of the string.

48. In Python, `"a" * 10` will create the string `"aaaaaaaaaa"`. Java does not have such a feature, but you can still accomplish the same effect. Write a program that asks the user to enter a string and how many times to copy it, and then creates a string containing the user's string repeated the appropriate number of times. Don't just print out the repeated stuff, but actually store it in a string.

49. Write a program that asks the user to enter a string, replaces all the spaces with asterisks, then replaces every fifth character (starting with the first) with an exclamation point, and finally concatenates three copies of that result together. For example, `this is a test` would become `!his*!s*a*!est!his*!s*a*!est!his*!s*a*!est`.

50. Write a program that asks the user to enter their name in lowercase and then capitalizes the first letter of each word of their name.

51. Students' email addresses at a certain school end with `@email.college.edu`, while professors' email addresses end with `@college.edu`. Write a program that first asks the user how many email addresses they will be entering, and then has them enter those addresses. After all the email addresses are entered, the program should print out a message indicating either that all the addresses are student addresses or that there were some professor addresses entered.

52. Write a program that asks the user to enter a string of lowercase letters and prints out the first letter alphabetically that is not in the string and a message if the string contains every letter. For instance, if the user enters the string `"a big cat did eat food"`, the letter output should be `h`.

53. Write a program that asks the user to enter a word with an odd number of letters and then prints out a string where the middle letter is printed once, the letters next to the middle are printed twice, the letters next to them are printed three times, etc. all in the order given by the original word. For instance, `abcdefg` would become `aaaabbbccdeefffgggg`.

54. Write a program that asks the user to enter their name and replaces each character of their name with the letter immediately following it in the alphabet (with *a* following *z*).

55. Write a program that asks the user for their name and prints out the "numerical value" of their name, where $a = 1$, $b = 2$, ..., $z = 26$, and you add up the numerical values of each letter in the name. For instance, the numerical value of *dave* is $4 + 1 + 22 + 5 = 32$. You can assume the name is in all lowercase.

56. Write a program that asks the user to enter a sequence of integers separated by semicolons. Your program should print the sum of all the integers. For instance, if the user enters `4;20;3;9` the program should print out 36.

57. Write a program that asks the user to enter a date in the format `mm/dd` and outputs the date in long form. For instance, given an input of `01/24`, your program should output `January 24`.

58. Write a program that asks the user to enter a date in a form `month/day/year`, where the month and day can be one or two digits and the year can be two or four digits. For instance, `2/7/13` and `02/07/2013` are two possible ways of entering the same date. Write a program that uses the `split` method to print the date in long form. The date above would be `February 7, 2013` in long form. Assume all dates are in the 2000s.

59. Write a program that does the inverse of the above. That is, it takes a date in long form, and outputs a date in the `month/day/year` format. But make your program versatile, so that the user can enter Feb., feb, February, or even Febueree and the program will still recognize the month. One way to do this is to just check that the first three letters of the month are correct. Capitalization should not matter. Also, make it so that the user may choose to enter the comma between the day and year or leave it out.

60. Write a program that asks the user to enter an angle in degrees/minutes/seconds form, such as `49d33'22''` or `153d4'22''`, where the input will starts with the number of degrees (0 to 360), followed by the letter `d`, followed by the number of minutes (0 to 60), followed by an apostrophe, followed by the number of seconds (0 to 60), followed by two apostrophes.

    The program should convert that angle to decimal form, rounded to two decimal places. For instance, `49d33'22` should become 49.56 and `153d4'22''` should become 153.07. To convert from degrees/minutes/seconds to the decimal form, use $d + \frac{m}{60} + \frac{s}{3600}$, where d, m, and s are the numbers of degrees, minutes, and seconds, respectively.

61. Companies often try to personalize their offers to make them more attractive. One simple way to do this is just to insert the person's name at various places in the offer. Of course, companies don't manually type in every person's name; everything is computer-generated. Write a program that asks the user for their name and then generates an offer like the one below. For simplicity's sake, you may assume that the person's first and last names are one word each.

    ```
    Enter name: George Washington

    Dear George Washington,

    I am pleased to offer you our new Platinum Plus Rewards card
    at a special introductory APR of 47.99%.  George, an offer
    like this does not come along every day, so I urge you to call
    now toll-free at 1-800-314-1592. We cannot offer such a low
    rate for long, George, so call right away.
    ```

62. Write a censoring program. Allow the user to enter some text and your program should print out the text with all the curse words starred out. The number of stars should match the length of the curse word. For the purposes of this program, just use the "curse" words *darn*, *dang*, *frickin*, *heck*, and *shoot*. Sample output is below:

    ```
    Enter some text: Oh shoot, I thought I had the dang problem
    figured out. Darn it. Oh well, it was a heck of a frickin try.

    Oh *****, I thought I had the **** problem figured out.
    **** it. Oh well, it was a **** of a ******* try.
    ```

63. Deciding the math course a student at a certain school should take depends on the student's major, their SAT math score, and their score on the math placement test (if they have to take it).

    Here are the rules: A student with a 530 or above on their Math SAT does not have to take the math placement test, and they may immediately take whatever math course they need for the core or their

major. If the student has a 520 or below, then they take the math placement test. If the student fails the math placement test, then they have to take either Math 101 or Math 102. Math 102 is for Math, CS, and Science majors and Math 101 is for everyone else. For students that pass the math placement test or have a 530 or above on their Math SAT, the math class they take depends on their major. Math, CS, and Science majors take Math 247 (Calculus). Education majors take Math 108. Everyone else takes Math 105 (Stats).

This is a lot of information for advisors to remember. Write a program that asks the user to input the student's intended major and their SAT Math score. If the SAT score is under 530, the program should ask if the student passed the math placement test. The program should then output the recommended class. Sample output:

64. Write a program that determines whether a word is a palindrome or not. A palindrome is a word that reads the same backwards as forwards.

65. Write a program that asks the user to enter a string and then prints out the first letter, then the first two letters, then the first three letters, etc., all on the same line. For instance, if the user enters `abcde`, the program would print out `a ab abc abcd abcde`.

66. Write a program that generates and prints a random string consisting of 50 random lowercase letters (`a` through `z`.

67. Write a program that asks the user to enter a sentence that contains some positive integers. Then write a program that reads through the string and replaces each number n in the sentence with $n+1$. For instance:

    input: `I bought 4 apples and 17 oranges.`
    output: `I bought 5 apples and 18 oranges.`

    input: `My favorite number is 8.`
    output: `My favorite number is 9.`

68. A simple way of encrypting a message is to rearrange its characters. One way to rearrange the characters is to pick out the characters at even indices, put them first in the encrypted string, and follow them by the odd characters. For example, the string *message* would be encrypted as *msaeesg* because the even characters are *m*, *s*, *a*, *g* (at indices 0, 2, 4, and 6) and the odd characters are *e*, *s*, *g* (at indices 1, 3, and 5). Write a program that asks the user for a string and uses this method to encrypt the string.

69. In algebraic expressions, the symbol for multiplication is often left out, as in `3x+4y` or `3(x+5)`. Computers prefer those expressions to include the multiplication symbol, like `3*x+4*y` or `3*(x+5)`. Write a program that asks the user for an algebraic expression and then inserts multiplication symbols any time a number is followed by a letter or an open parenthesis. [Hint: `Character.isDigit(c)` can be used to check if the character `c` is a digit.]

70. Write a program that uses a loop to generate the 26-line block of letters, partially shown below.

```
abcdefghijklmnopqrstuvwxyz
bcdefghijklmnopqrstuvwxyza
cdefghijklmnopqrstuvwxyzab
...
yzabcdefghijklmnopqrstuvwx
zabcdefghijklmnopqrstuvwxy
```

71. Write a program that converts a time from one time zone to another. The user enters the time in the usual American way, such as `3:48 pm` or `11:26 am`. The first time zone the user enters is that of the original time and the second is the desired time zone. The possible time zones are `Eastern`, `Central`, `Mountain`, or `Pacific`.

```
Time: 11:48 pm
Starting zone: Pacific
Ending zone: Eastern
2:48 am
```

72. Write a program that prints out `2015` without using any numbers anywhere in your program.

8.4 Exercises involving lists and arrays

73. Write a program that creates an array of 100 random integers that are either 0 or 1. Then print out the array and how many 1s are in the array.

74. Do the same as the problem above, but use a list instead of an array.

75. Create an array that consists of the first 100 perfect squares: 1, 4, 9, ..., 10000.

76. Generate and print out a list of 10 random integers from 1 to 50. Then do the following:

    (a) Print out how many even numbers are in the list.

    (b) Print out *yes* or *no*, depending on whether or not the list contains any numbers greater than 45.

    (c) Create a new list that consists of the elements of the original list in reverse order.

    (d) Ask the user for an integer n. The program should reverse the first n things in the list and leave the others alone. For instance, if the list is `["A","B","C","D","E","F","G","H","I","J"]` and the user chooses 6, we get `["F","E","D","C","B","A","G","H","I","J"]`.

77. Write a program that generates an array of 50 random integers from 1 to 100. Once you have generated the array, replace each integer in the array with its square.

78. Write a program that creates a list containing the first 100 triangular numbers. The triangular numbers are the numbers 1, 1+2=3, 1+2+3=6, 1+2+3+4=10, etc.

79. Write a program that asks the user to enter 10 numbers and stores them in a list. Then print out the smallest number the user entered.

80. Write a program that contains a list of at least 10 verbs, a list of at least 10 nouns, and a list of at least 10 adjectives. Then have the program randomly generate and print a sentence of the form "The <*adjective*> <*noun 1*> <*verb*> the <*noun 2*>." All the words except "the" should be randomly pulled from the appropriate lists.

81. Write a program that asks the user to enter in 10 quiz scores, drops the two lowest scores, and prints out the average of the rest of them. Use lists to do this problem.

82. Do the problem above without lists.

83. Write program that contains a list of 10 names. Then do the following:

    (a) Print out a random name from the list (the name should vary each time the program is run). The original list should not be changed or reordered.

    (b) Have the program reorder the list into alphabetical order.

    (c) Ask the user for an integer n. The program should reverse the first n things in the list and leave the others alone.

84. Write a program that creates a array of 1000 random integers from 0 to 49. Then have the program create a new array of 50 integers whose nth element is the number of times that n appears in the array of random integers.

85. Write a program that generates an array of 50 random integers from 1 to 100. Then rotate the elements of the array so that the element in the first position moves to the second index, the element in the second position moves to the third position, etc., and the element in the last position moves to the first position.

86. Write a program that asks the user to enter a sentence and returns a count of how many 5-letter words are in the sentence. For this, you should be careful of punctuation. Assume punctuation consists of the following characters: `. ! ? ( ) , ; :`

87. Write a program that generates a list of 100 random numbers from 1 to 50. Print out the list and then write some code that changes the first two occurrences of 50 to -999. If there is just one 50 in the list, then just change it. If there are no 50s in the list, then print a message saying so.

88. Create a list that contains the numbers 1 through 20 in order. Then have the program randomly pick three pairs of elements in the list and swap them. Print out the resulting list. An element may be swapped more than once. A possible output is `[7,2,3,4,5,6,1,8,12,10,11,9,13,14,15,18,17,16,19,20]`. The swaps in this example are `1,7`, then `9,12`, and finally `16,18`.

89. Write a program that asks the user to enter a date in the format *month/day*. The program should print out whether the date is real. For instance, 14/10 is not a real date, and neither is 2/30. Do this by using a list that contains the number of days in each month.

90. Create and print a list that contains 10 random numbers from 1 through 99,999. Then print out the sum of the lengths of the numbers. For instance, if the list is `[8,53,124,4,2,88423,1010,455,111,2]`, the program should output 24, which is $1+2+3+1+1+5+4+3+3+1$, adding up the how many digits long each of those numbers are.

91. Create and print a list that contains 100 random 0s, 1s, and 2s. Then write some code the prints out all the indices i in the list where the element at index i is different from the element at index $i-1$. For instance, if the list is `[0,0,1,1,1,1,0,2,2,2,0,...]`, then the program should print out 2, 6, 7, and 10 since those are indices at which the elements of the list change.

92. Write a program that asks the user to enter a string of lowercase letters and prints out a count of how many times each letter appears in the string.

93. Write a simple quiz game that has a list of 10 questions and a list of answers to those questions. The game should give the player four randomly selected questions to answer. It should ask the questions one-by-one, and tell the player whether they got the question right or wrong. At the end it should print out how many out of four they got right.

94. Write a program that asks the user to enter a length in feet. The program should then give the user the option to convert from feet into various other units. The user will enter a 1 to indicate converting to feet, a 2 for yards, 3 for miles, 4 for millimeters, 5 for centimeters, 6 for meters and 7 for kilometers. While this can be done with if statements, instead use a list whose elements are the various conversion factors.

8.5 Exercises involving 2d arrays

95. Create and print a 10×10 array that consists of random integers from 1 through 5. Then print a count of how many fives are in the array.

96. Write a program that creates a 6×6 array of random integers from 0 to 9. Then do the following:

    (a) Print out the array.

    (b) Add up all the entries in the last column.

    (c) Create a new one-dimensional array from the two-dimensional array such that all the entries in row 0 of the original are followed by all the entries in row 1, which are followed by all the entries in row 2, etc. See the example below:

    ```
    2 4 9 1 1 0
    5 7 5 4 2 1
    6 6 3 5 9 2
    7 1 2 0 9 4
    7 0 2 1 8 7
    7 0 0 0 3 7

    [2,4,9,1,1,0,5,7,5,4,2,1,6,6,3,5,9,2,7,1,2,0,9,4,7,0,2,1,8,7,7,0,0,0,3,7]
    ```

    (d) Create a new two-dimensional array that swaps the rows and columns of the original. That is, row 0 of the original should become column 0 of the new one, row 1 of the original should become column 1 of the new one, etc.

97. The following is useful as part of a program to play *Battleship*. Suppose you have a 5×5 array that consists of zeroes and ones. Ask the user to enter a row and a column. If the entry in the array at that row and column is a one, the program should print `Hit` and otherwise it should print `Miss`.

98. Write a program that asks the user for an integer n and creates an $n \times n$ array of random integers from 1 to 9. Then:

    (a) Print the array.

    (b) Print the largest value in the third row.

    (c) Print the smallest value in the fourth column.

    (d) Print the average of all the entries in the array.

99. Write a program that generates and prints a 15×15 array of random zeros, ones, and twos, and then adds up all the entries on or below the main diagonal. These are all entries (r, c), where the column number c is less than or equal to the row number r.

100. Write a program that first creates and prints a 6×6 array of random ones, twos, and threes. Then ask the user to enter a starting row, ending row, starting column, and ending column, and print out the sum of all the entries in the array within the range specified by the user.

101. Write a program that asks the user for an integer n and creates and prints an $n \times n$ array whose entries alternate between 1 and 2 in a checkerboard pattern, starting with 1 in the upper left corner.

102. Write a program that first creates and prints a 6×6 array of integers from 0 to 5. Then write some code that determines if there are two consecutive zeros in the array, either horizontally or vertically. Try to do this with loops rather than dozens of if statements.

103. Ask the user for an integer n. Then create a $n \times n$ character array that consists of spaces everywhere except that the first row, last row, first column, last column, main diagonal, and off diagonal should all be # symbols. Print out the array. Below is an example of what the array should look like with $n = 10$. Remember that you are creating an array, not just printing characters.

```
##########
##      ##
# #    # #
#  #  #  #
#   ##   #
#   ##   #
#  #  #  #
# #    # #
##      ##
##########
```

104. Write a program that creates and prints a 10×10 array that has exactly 10 zeroes and 90 ones, all randomly ordered.

8.6 Exercises involving file I/O

105. The file *expenses.txt* consists of a bunch of amounts of money spent on things, one amount on each line. Have your program read the file and print out only those expenses that are $2000 or greater.

106. The files *first_names.txt* and *last_names.txt* are lists of common first and last names. Write a program that uses these files to print out 10 random full names, where a full name consists of a first and last name separated by space, like `"Don Knuth"`.

107. Each line of the file *population.txt* contains a country name and its population, separated by a tab. Write a program that uses the file to print out all the countries whose names start with *G* that have at least 500,000 people.

108. Write a program that reads through `wordlist.txt` and produces a new file `shortwordlist.txt` that contains all the words from `wordlist.txt` that are 5 or less letters long.

109. The file `temperatures.txt` contains several temperatures in Celsius. Write a program that reads the temperatures from the file, converts them to Fahrenheit, and prints out the converted temperatures.

110. Each line of the file *elements.txt* has an element number, its symbol, and its name, each separated by `" - "`. Write a program that reads this file and outputs a new file *symbols.txt*, where each line of the file contains just the element symbol and number, separated by a comma, like below. Remember that you are creating a new file, not just printing things on the screen.

```
H,1
He,2
Li,3
...
```

111. The file `nfl1978-2013.csv` contains the results of every regular-season NFL game from 1978 to 2013. Open the file to see how it is arranged. Allow the user to enter two team names (you can assume they enter the names correctly) and have the program print out all of the games involving the two teams where one of the teams was shut out (scored no points).

112. Write a program that asks the user to enter a chemical name and prints out the names of the elements that appear in the chemical name. For instance:

input: `NaCl` output: `Sodium, Chlorine`
input: `H2SO4` output: `Hydrogen, Sulfur, Oxygen`

The file `elements.txt` should be helpful. Element names are one to three letters long, with the first letter always a capital and the others always lowercase.

113. The file *continents.txt* contains a list of country names arranged by continent. Please look at the file to see how things are arranged. Using this file, write a quiz program that randomly selects a country and asks the user to guess what continent the country is on. Print out a message indicating whether or not they guessed correctly. Use `equalsIgnoreCase` when checking their answer.

114. This program asks you to read a file that contains multiple choice questions and *create a new file* that contains those same questions with the choices reordered. For instance, shown on the left is what a typical file would look like and on the right is what it might look like after reordering the choices for each question.

```
1. What is the capital of Estonia?
(a) Tallinn
(b) Vilnius
(c) Riga
(d) Sofia

2. After Asia, the largest continent is
(a) Africa
(b) North America
(c) South America
```

```
1. What is the capital of Estonia?
(a) Vilnius
(b) Sofia
(c) Tallinn
(d) Riga

2. After Asia, the largest continent is
(a) North America
(b) Africa
(c) South America
```

The file is structured such that there is a question on one line, followed by several choices, each on their own lines. There is an empty line between questions. Some questions may have more choices than others, and there is no limit on the number of questions in the file. Test your program with `multiple_choice.txt` and `multiple_choice2.txt`.

115. The file `first_names.txt` contains a large number of first names, each on a separate line, and the file `last_names.txt` that contains a large number of last names, each on a separate line. Write a program that reads these two files and uses them to generate 10 random names in the form `last name, first name` and outputs them to a file called `names.txt`, with each name on a separate line. The random names generated should be different each time the program is run.

116. Use either `wordlist.txt` or `crazywordlist.txt` to do the following. For all of these, *y* is not considered a vowel.

(a) Print all words ending in *ime*.

(b) Print how many words contain at least one of the letters *r, s, t, l, n, e*.

(c) Print all words with no vowels.

(d) Print all palindromes.

(e) Print out all combinations of the string "num" plus a five-letter English word. Capitalize the first letter of the three letter word.

(f) Print how many words that contain double letters next each other like *aardvark* or *book*, excluding words that end in *lly*.

(g) Print all words that contain a *q* that isn't followed by a *u*.

(h) Print all words that contain *ab* in multiple places, like *habitable*.

(i) Print all the words that contain at least nine vowels.

(j) Print all four-letter words that start and end with the same letter.

(k) Print all words that contain each of the letters *a*, *b*, *c*, *d*, *e*, and *f* in any order. There may be other letters in the word. Two examples are *backfield* and *feedback*.

(l) Print all three letter words where first and last letters are same and middle is a vowel, and all five possibilities are words. For example, *pat pet pit pot put*.

(m) Print all the words where the letters *m*, *b*, and *a*, appear in that order, though not necessarily one right after another. An example is the word *embrace*. In other words, there is an *m*, then possibly some letters, then a *b*, then possibly some more letters, and then an *a*.

(n) Print the word that has the most i's.

(o) Print all the words that contain four or more vowels (in a row).

117. The file `logfile.txt` contains lines that look like the one below.

```
Gosling 12:14 13:47
```

There is a one word username followed by a log-on time and a log-off time, both given in 24-hour format. All are separated by spaces. Write a program that finds and prints out all the users that were logged on for at least an hour. You can assume that each log-on and log-off occurs within the same day.

118. You are given a file called `students.txt`. A typical line in the file looks like:

```
walter melon    melon@email.college.edu    555-3141
```

There is a name, an email address, and a phone number, each separated by tabs. Write a program that reads through the file line-by-line, and for each line, capitalizes the first letter of the first and last name, changes the email address to be `@gmail.com`, and adds the area code 301 to the phone number. Your program should write this to a new file called `students2.txt`. Here is what the first line of the new file should look like:

```
Walter Melon    melon@gmail.com    301-555-3141
```

119. You are given a file `namelist.txt` that contains a bunch of names. Print out all the names from the file in which the vowels *a*, *e*, *i*, *o*, and *u* appear in order (with repeats possible). The first vowel in the name must be *a* and after the first *u*, it is okay for there to be other vowels. An example is *Ace Elvin Coulson*.

120. Write a quiz game where the questions and answers are stored in text files (either one file or multiple files – your choice). The program should randomize the order of the questions at the start of each game, ask all the questions, and print out an appropriate message each time the person answers a question. Print out the percent correct at the end of the game. The program should work for any size file.

121. One way to save the state of a game is to write the values of all the important variables to a file. Write a simple guess-a-number game, where the computer picks random numbers from 1 to 10, and the user has one try to guess the number. The program should keep track of the amount right and wrong. After each guess, the user has the option to save their progress and quit or to keep playing. To save their progress, write the amount right and wrong to a file. When the player starts the game, they should have the option to start a new game or continue an old one.

122. The file `romeoandjuliet.txt` that contains the text of a well-known Shakespearean play. Write a program that reads through the file and creates two ArrayLists. The first list should contain all the words used in the play. The items in the second list should correspond to the words in the first list and should be the number of times the corresponding word occurs in the play. I would recommend converting things to lowercase and stripping out all punctuation except apostrophes and hyphens.

123. The a file `scores.txt` contains results of every 2009-10 NCAA Division I basketball game. A typical line of the file looks like this:

```
02/27/2010 RobertMorris 61 MountSt.Mary's 63
```

Use the file to answer some of the following:

(a) Find the average number of points scored per game. In the game above, there were 124 points scored. Find the average of the points scored over all the games in the file.

(b) Pick your favorite team and scan through the file to determine how many games they won and how many games they lost.

(c) Write a program that finds the most lopsided game, the game with the largest margin of victory. You should print out all the information in the file about that game, including the date, teams, and scores.

(d) Write a program that uses the file to find something interesting. In order for this to count for credit, you must run your idea by me first so I can verify that it is not something too easy.

124. The file `nfl_scores.txt` contains information on how many times each final score has happened in an NFL game. Please look at the file to see how things are arranged. Using the file, print out all the final scores that have *never* happened, with the limitation that both teams' total points must be 0 or in the range from 2 to 49. For instance, the scores 4-2, 11-9, 35-29, and 46-0 have never happened. 20-1 and 77-55 have also never happened, but those fall outside of the specified range.

125. The file `high_temperatures.txt` contains the average high temperatures for each day of the year in a certain city. Each line of the file consists of the date, written in the month/day format, followed by a space and the average high temperature for that date. Find one of the 30-day period over which there is the biggest increase in the average high temperature (there are actually a few 30-day periods with that maximum increase).

126. An acronym is an abbreviation that uses the first letter of each word in a phrase. We see them everywhere. For instance, NCAA for National Collegiate Athletic Association or NBC for National Broadcasting Company. Write a program where the user enters an acronym and the program randomly selects words from a wordlist such that the words would fit the acronym. Below is some typical output generated when I ran the program:

```
Enter acronym: ABC
[addressed, better, common]

Enter acronym: BRIAN
[bank, regarding, intending, army, naive]
```

127. Write a program that reads a 2d array of integers from a file into a Java 2d array. The file will be structured something like below:

```
2 13 3 192 2
3 99 11 132 13
29 944 2 44 222
```

Each row of the array is on its own line, and entries are separated by spaces. Your program should be able to handle files of any size, not just 3×5 arrays like the one above. Remember that your program should actually store the values in an array. Please have your program print out the array. You can use the files `array1.txt` and `array2.txt` to test your program.

128. The site http://www.mbeckler.org/ncaa_tournament/ has a text file of NCAA tournament results from 1985-2009. The file is in a compressed format which is described near the bottom of the page. Use the file to determine what percentage of matchups between a #5 seed and a #12 seed were won by the #12 seed.

129. The word *part* has the interesting property that if you remove its letters one by one, each resulting step is a real word. For instance, *part* $\rightarrow$ *pat* $\rightarrow$ *pa* $\rightarrow$ *a*. You may remove the letters in any order, and the last (single-letter) word needs to be a real word as well. Find all eight-letter words with this property.

8.7 Miscellaneous exercises

130. Write a program that takes a list of ten product names and a list ten prices, applies an 11% discount to each of the prices, and prints the info with the product names left-justified and the discounted prices right-justified, like below:

```
apples           $   0.93
detergent        $  11.35
...
cordless drill   $ 125.35
```

You can put whatever you want in your lists; just make sure your printing code can work with whatever lists (of whatever size) might be used. You can assume product names will be no more than 30 characters and that prices will be under $10,000.

131. Write a program that figures out someone's typing speed. To do this, ask the user to enter some text. Time how long it takes them to enter the text. Count the number of words they entered and use that and the time to print out the number of words per minute that the user types.

132. The program you will write in this problem is for testing reaction times. The program should first ask the person if they are ready. When they press enter, then the program should pause for a random time from 1 to 3 seconds (use `Thread.sleep` for this). Then the program should print a message and the person must press enter again as soon as they can. Time how long it takes between when the message is printed and when the person presses enter, and print out the result.

133. Write a multiplication game that gives the player 10 randomly generated problems. The program should print each problem, ask the user for an answer, and print a message as to whether they got it right or wrong. The numbers generated should range from 1 to 50. Your program should also keep score.

Score the problems as follows: Harder problems are worth more points — 1 point if the answer is in the range 1-144, 5 points if it is in the range 145-400, and 15 points otherwise. You should also time how long it takes the person to answer the question. The point values above should be tripled if the person takes less than 5 seconds to answer, and the point values should be doubled if they take from 5 to 10 seconds to answer. There is also a 2-point penalty for a wrong answer.

134. The following method can be used to encrypt data: First ask the user to enter a string to be encrypted. Then generate a random number from 0 to 25 for each character in the message. That random number determines how many letters to move forward, wrapping around if necessary. For instance, to encrypt the message `"java"`, suppose the random numbers 3, 6, 10, 19 are generated. The encrypted message is `"mgft"` since `m` is 3 letters after `j`, `g` is 6 letters after `a`, `f` is 10 letters after `v` (wrapping around), and `t` is 19 letters after `a`. Print out the encrypted message.

135. Write a program that counts how many numbers from 1 to 10000 consist of only odd digits.

136. In number theory it is useful to know how many divisors a number has. Write a program that asks the user for a number and prints out how many divisors the number has.

137. Write a program that asks the user for a weight in ounces and returns the weight in pounds and ounces. For instance, 43 ounces becomes 2 pounds, 11 ounces.

138. A number is called a *perfect number* if it is equal to the sum of all of its divisors, not including the number itself. For instance, 6 is a perfect number because the divisors of 6 are 1, 2, 3, 6 and $6 = 1 + 2 + 3$. As another example, 28 is a perfect number because its divisors are 1, 2, 4, 7, 14, 28 and $28 = 1 + 2 + 4 + 7 + 14$. However, 15 is not a perfect number because its divisors are 1, 3, 5, 15 and $15 \neq 1 + 3 + 5$. Write a program that finds all four of the perfect numbers that are less than 10000.

139. Write a program that asks the user to enter a number n and displays a multiplication table of the numbers 1 through $n-1$ with the arithmetic done modulo n. For instance, if $n = 5$, to compute 3×3, we do $3 \times 3 = 9$ and then mod that by 5 to get 4. Use `printf` to right-justify table entries. Shown below are the tables you should get for $n = 5$ and $n = 11$

```
 1  2  3  4
 2  4  1  3
 3  1  4  2
 4  3  2  1
```

```
 1  2  3  4  5  6  7  8  9 10
 2  4  6  8 10  1  3  5  7  9
 3  6  9  1  4  7 10  2  5  8
 4  8  1  5  9  2  6 10  3  7
 5 10  4  9  3  8  2  7  1  6
 6  1  7  2  8  3  9  4 10  5
 7  3 10  6  2  9  5  1  8  4
 8  5  2 10  7  4  1  9  6  3
 9  7  5  3  1 10  8  6  4  2
10  9  8  7  6  5  4  3  2  1
```

140. Recall that checking to see if a set contains something turns out to be a lot faster than checking to see if a list contains something.

```
Set<String> words = new LinkedHashSet<String>();
```

Write a program that reads the contents of a wordlist into a set and then finds all words such that if you cut the word in half, both halves are words themselves. If the word has an odd number of letters cut it in half by not including the middle letter. Store all the words you find in a list and just print out the last 10 words you find.

After you do the above, try the program on `worlist.txt` and `crazywordlist.txt` using a list instead of a set to store the wordlist and compare the speed of that approach with the set approach.

141. Write a program that asks the user for a double a and an integer b and rounds a to the nearest multiple of b. For instance, if the user enters $a = 22.3$ and $b = 10$, then the program would round 22.3 to the nearest multiple of 10, which is 20. If the user enters $a = 44.12$ and $b = 5$, then the program would round 44.12 to the nearest multiple of 5, which is 45.

142. Ask the user to enter the numerator and denominator of a fraction, and the digit they want to know. For instance, if the user enters a numerator of 1 and a denominator of 7 and wants to know the 4th digit, your program should print out 8, because $\frac{1}{7} = .142857...$ and 8 is the 4th digit. [Warning: don't use doubles for this problem. They only have about 15 digits of precision and likely won't give the correct answer if the user wants anything past the 15th digit.]

143. This exercise is useful in creating a *Memory* game. Randomly generate a 5×5 array of assorted characters such that there are exactly two of each character. An example is shown below.

```
@ 5 # A A
5 0 b @ $
$ N x ! N
0 - + # b
- x + c c
```

144. Write a program that asks the user to enter some text. Then write some code that replaces any occurrence of a date with the text `***Date removed***`. For the purposes of this problem, a date is an expression the form `<month><sep><day><sep><year>`, where `<sep>` can be - or /, `<month>` can be any of the numbers 1 through 12, `<day>` can be any of the numbers 1 through 31, and `<year>` can be given in either two- or four-digit format. If given in four-digit format, it must begin with 19 or 20. [Hint: look up some info on regular expressions in Java.]

145. The following is useful as part of a program to play *Minesweeper*. You might want to try playing the game before attempting this problem if you are not familiar with how the game works.

(a) Suppose you have a two-dimensional array that consists of zeros and *M*'s. Write a function takes the array as an argument and modifies the array so that it has *M*'s in the same place, but the zeroes are replaced by counts of how many *M*'s are in adjacent cells (adjacent either horizontally, vertically, or diagonally). An example is shown below.

```
0 M 0 M 0          1 M 3 M 1
0 0 M 0 0          1 2 M 2 1
0 0 0 0 0          2 3 2 1 0
M M 0 0 0          M M 2 1 1
0 0 0 M 0          2 2 2 M 1
```

(b) Write a function that takes an array `a` in the form of the modified array above (the one on the right), another array `b` consists of zeros and ones (0 for cells that are hidden and 1 for cells that are showing), as well as a row `r` and a column `c` and does the following:

- If `a` contains an M in the selected row and column, print a message saying `You lost` or something like that.
- If `a` contains a positive integer in the selected row and column, change the state of `b` in that row and column to a 1.
- If `a` contains a zero in the selected row and column, change the state in `b` of not only that row and column, but all the other locations that should also be changed, according to the rules of Minesweeper.

146. Write a Java program that computes at least 100 digits of π. You will probably have to do some internet research on how to generate digits of π, but don't just copy someone else's code to do this.

8.8 Exercises involving functions

147. Write a function called `getBillTotal` that takes a bill amount and tip percentage (both doubles) and returns the total bill amount (also a double).

148. Write a function called `repeat` that takes a string `s` and an integer `n` and returns a string containing `n` copies of `s`.

149. Write a function called `count` that takes an integer array and an integer value as arguments and returns how many times that value appears in the array.

150. Write a function called `range` that takes integer `m` and `n` and returns a list containing the integers from `m` through `n-1`. For instance `range(3,8)` should return the list `[3,4,5,6,7]`.

151. Write a function called `midRange` that takes an integer list as its only argument and returns $(x + y)/2$, where x is the largest value in the list and y is the smallest.

152. Write a function called `repeat` that takes a string `s` and an integer `n` and returns the string repeated `n` times. For instance, `repeat("hello", 3)` should return `"hellohellohello"`. (Note that the function itself should not do any printing.)

153. Write a function called `swap01` that takes an integer list of 0s and 1s and returns a new list in which all the 0s have been changed to 1s and all the 1s have been changed to 0s. The original list should not be affected.

154. Write a function called `index` that takes an integer array and an integer value as arguments and returns the first index in the array at which value occurs. It should return `-1` if the value is not found.

155. Arrays don't have an `in` method. Write a function that takes an integer array and an integer value as arguments and returns a boolean `true` or `false` depending on whether the value is in the array.

156. Write a function called `printLines` that takes a file name (as a string) as its only argument and prints out all the lines of the file. It shouldn't return anything.

157. Write a function called `getRandomCapital` that takes no arguments and returns a random capital letter.

158. Write a function called `getRandomChar` that takes a string as an argument and returns a random character from that string.

159. Write a function called `reverse` that takes an integer array as an argument, reverses that array, and returns nothing. The caller's array should be modified.

160. Write a function called `closest` that takes an array of integers `a` and an integer `n` and returns the largest integer in `a` that is not larger than `n`. For instance, if `a=[1,6,3,9,11]` and `n=8`, then the function should return 6, because 6 is the closest thing in `a` to 8 that is not larger than 8. Don't worry about if all of the things in `a` are smaller than `n`.

161. The Euler ϕ function, $\phi(n)$, tells how many integers less than or equal to n are relatively prime to n. (Two numbers are said to be relatively prime if they have no factors in common, i.e. if their gcd is 1). Write a Java function called `phi` that implements the Euler ϕ function. Test your function with `phi(101)` which should return 100, and `phi(64)`, which should return 32.

162. Below is described how to find the date of Easter in any year. Write a function that takes year and returns a string with the date of Easter in that year. Despite its intimidating appearance, this is not a hard problem. Note that $\lfloor x \rfloor$ is the *floor* function, which for positive numbers just drops the decimal part of the number. In Java it is `Math.floor`.

$C =$ century (1900s $\rightarrow C = 19$)

$Y =$ year (all four digits)

$m = (15 + C - \lfloor \frac{C}{4} \rfloor - \lfloor \frac{8C+13}{25} \rfloor) \bmod 30$

$n = (4 + C - \lfloor \frac{C}{4} \rfloor) \bmod 7$

$a = Y \bmod 4$

$b = Y \bmod 7$

$c = Y \bmod 19$

$d = (19c + m) \bmod 30$

$e = (2a + 4b + 6d + n) \bmod 7$

Easter is either March $(22 + d + e)$ or April $(d + e - 9)$. There is an exception if $d = 29$ and $e = 6$. In this case, Easter falls one week earlier on April 19. There is another exception if $d = 28$, $e = 6$, and $m = 2, 5, 10, 13, 16, 21, 24$, or 39. In this case, Easter falls one week earlier on April 18.

163. It is possible to compute square roots using just multiplication, addition, and division. To estimate the square root of the number a to very high accuracy, start by setting $x = 2.0$. Then replace x with the average of x and a/x. Then replace this new x with the average of x and a/x (using the new x). Repeat this until consecutive estimates are within 10^{-10} of each other. Write a function that takes a double and uses this method to estimate the square root of it.

164. Write a function that takes an argument which is a list of names and prints out a randomized grouping of the names into teams of two. You can assume that there are an even number of names. The function shouldn't return anything. Here is sample output for the list `[A, B, C, D, E, F, G, H, I, J]`:

```
Team 1: D, J
Team 2: I, A
Team 3: B, C
Team 4: E, G
Team 5: F, H
```

165. Write a function called `matches` that takes two strings as arguments and returns an integer. The function should return -1 if the strings are of different lengths. Otherwise, it should return how many matches there are between the strings. A match is where the two strings have the same character in the same position. For instance, `"Java"` and `"lavatory"` match in the second, third, and fourth characters, so the function would return 3.

166. Write a function called `filter` whose arguments are an integer array `a` and two integers `x` and `y`. The function should return an integer array that consists of the elements of `a` that are between `x` and `y` (inclusive).

    For instance, if `a` is the array `[3,7,11,-2,6,0]`, then `filter(a,0,10)` should return the array `[3,7,6,0]`, an array with all the elements of `a` that are between 0 and 10.

167. Write a function that takes a list/array of strings and prints out only those strings that contain no repeated letters.

168. Write a function `randString(n)` that generates strings of `n` random A's, B's, and C's such that no two consecutive letters are the same. Each call to the function should produce a different random string with no consecutive letters being the same.

169. Write a function that is given a 9×9 potentially solved Sudoku (as a 2d array of integers) and returns `true` if it is solved correctly and `false` if there is a mistake. The Sudoku is correctly solved if there are no repeated numbers in any row or any column or in the any of the nine "blocks."

170. Our number system is called *base 10* because we have ten digits, 0, 1, ..., 9. Some cultures have used a base 5 system, where the only digits are 0, 1, 2, 3, and 4. Here is a table showing a few conversions:

10	**5**	**10**	**5**	**10**	**5**	**10**	**5**
0	0	8	13	16	31	24	44
1	1	9	14	17	32	25	100
2	2	10	20	18	33	26	101
3	3	11	21	19	34	50	200
4	4	12	22	20	40	51	201
5	10	13	23	21	41	124	444
6	11	14	24	22	42	125	1000
7	12	15	30	23	43	126	1001

Write a function called `convertToBase5` that converts a base 10 number to base 5. It should return the result as a string of digits. One way to convert is to take the remainder when the number is divided by 5, then divide the number by 5, and repeat the process until the number is 0. The remainders are the base 5 digits in reverse order. Test your function with 3794. The correct base 5 string should be `"110134"`.

8.9 Object-oriented exercises

171. Write a class called `BankAccount` that has the following:

- A private double field called `amount` that stores the amount of money in the account.
- A private double field called `interestRate` that stores the account's interest rate.
- A private string field called `name` that stores the name of the account holder.
- A constructor that just sets the values of the three fields above.
- Getters and setters for each of the three fields.
- A method called `applyInterest` that takes no arguments and applies the interest to the account. It should just modify the `amount` field and not return anything. For instance, if the account has $1000 in it and the interest rate is 3%, then the `amount` variable should be changed to $1030 ($1000 + 3% interest).

172. Write a driver class to test the class above. Use it to create a new `BankAccount` object for a user named `"Juan De Hattatime"` who has $1000 at 3% interest. Then do the following:

- Use the `applyInterest` method to apply the interest to the account.
- Print out how much money is now in the account after applying the interest.
- Change the account's interest rate to 2% (using the setter method).
- Use the `applyInterest` method to apply the interest to the account again
- Print out how much money is now in the account after applying the interest again.

173. Write a class called `IPair` with the following:

- private integer fields `x` and `y` and a constructor that sets them
- getters and setters for both `x` and `y`
- a `toString` method that returns a string representation of the class in the form `(x,y)`
- an `equals` method that lets you compare two IPair objects to see if they are equal (use your IDE to generate it for you)

174. Test out the `IPair` class in another class by doing the following:

- Create and print a list of IPair objects that contains all pairs of the form (x, y) where x and y range from 1 to 10.

- Write a couple of lines of code that prints out whether the pair $(3, 4)$ is in the list.

175. Use the `Card` and `Deck` classes from Section 4.5 to do the following: Deal out five cards and print what they are. Print out whether or not the hand is a flush (where all five cards have the same suit).

176. Write a class called `Item` that represents an item for sale. It should have the following:

    - Fields representing the name (string) and price (double) of the item
    - A constructor that sets those fields, as well as getters for those fields (but no setters)
    - A `toString` method that returns a string containing the item name and price, with the price formatted to exactly 2 decimal places [Hint: `String.format` can create a string with formatting codes like the ones `printf` uses]
    - An `equals` method that determines that two Item objects are equal if they have the same name (use your IDE to generate this)

177. Write a class called `ShoppingCart` that might be used in an online store. It should have the following:

    - A list of Item objects that represents the items in the shopping cart
    - A constructor that creates an empty list of items
    - A method called `add` that takes a name (string) and a price (double) and adds an Item object with that name and price to the shopping cart
    - A method called `total` that takes no arguments and returns the total cost of the items in the cart
    - A method called `removeItems` that takes an item name (a string) and removes any Item objects with that name from the shopping cart. It shouldn't return anything.
    - A `toString` method that returns a string containing info on all the items in the shopping cart
    - A `main` method that tests out the shopping cart as follows: (1) create a shopping cart; (2) add several items to it; (3) print the cart's total cost (using the `total` method); (4) remove one of the items; (5) print out the cart.

178. Write a class called `RestaurantCheck`. It should have the following:

    - Fields called `subTotal`, `salesTaxPercent`, `tableNumber`, and `serverName` representing the bill without tax added, the sales tax percentage, table number, and name of the server.
    - A constructor that sets the values of all four fields
    - Getters and setters for each
    - A method called `calculateTotal` that takes no arguments and returns the total bill including sales tax.
    - A method called `printcheck` that takes a filename as its only argument and should writes information about the check to that file, formatted like below:

```
Table Number: 17
Server: Sonic the Hedgehog
Sales tax: 6.0%
Subtotal: $23:14
Total: $24.53
```

179. Create a class called `WordList` that has the following:

    - A String list field called `words` (that will store a list of words).
    - A constructor that takes a filename (a string) as an argument and fills up `words` with the words from the file. You can assume the file just has one word per line.
    - A static method called `isPalindrome` that takes a string argument and returns a boolean indicating whether that string is a palindrome (reads same forward and backward).
    - A non-static method called `getPalindromes` that takes no arguments and returns a list of all the palindromes contained in `words`.
    - A main method that tests out the constructor and the two methods.
    - Add an *interesting* method or two to the WordList class.

180. Create a class called `MyScanner` that recreates some of the functionality of Java's Scanner class. The class should have the following:

    - A string field called `data` and an integer field called `location` (this keeps track of where we are in scanning the string).
    - A constructor that takes a string as a parameter, sets `data` to it, and also sets `location` to 0.
    - A `next` method that takes no arguments and returns a string consisting of all the characters of `data` from where the location variable is currently at, up until the next space (or the end of the string if there are no more spaces). This method will have to adjust the `location` variable.
    - A `hasNext` method that takes no arguments and returns `true` or `false` based on whether there is anything else left to scan in the string.
    - A `main` method that tests out the constructor and the two methods.

181. A three-dimensional vector is a mathematical object with three coordinates, like $\langle 1, 4, 6\rangle$ or $\langle 2, 2, 5\rangle$. The three integers are called the components of the vector. Write a class called `Vector` that has the following:

    - Private `int` fields `x`, `y`, and `z` representing the vector's components.
    - A constructor that sets all three fields
    - Getters for each field, but no setters
    - A method called `toString` that has no parameters and returns a string in the format of `<x,y,z>`. For instance, if `x`, `y`, and `z` are 1, 2, and 3, respectively, it would return the string `"<1,2,3>"`
    - A method called `length` that takes no arguments and returns the double, $\sqrt{x^2 + y^2 + z^2}$.
    - A method called `add` that takes another Vector object, adds the `x`, `y`, and `z` components of that vector with the corresponding components of the current vector and returns a new vector with added components. For instance, if `v` is the vector $\langle 1, 4, 6\rangle$ and `w` is the vector $\langle 2, 2, 5\rangle$, then `v.add(w)` would produce the vector $\langle 3, 6, 11\rangle$.
    - A static method also called `add` that takes two Vector objects and adds them as described above, returning a new vector.

182. Write a driver for the class above that does the following:

    - Create a vector $u = \langle 1, 2, 3\rangle$.
    - Create a list of 100 vectors, where the vectors in the list have random components with values from 1 through 5.
    - Print out the list. The list should display something like `[<1,4,2>, <2,4,4>, ...]`, not `[Vector@7c62, Vector@84a3,...]`.
    - Print out the value of just the x component of the last vector in the list.
    - Print out the length of the first vector in the list.
    - Create a vector $v = \langle 4, 5, 6\rangle$ and use both `add` methods to add it to u.

183. Write a class called `Ticket` that has the following:

    - A double field `cost` and a string field `time` (assume times are stored in a format like `"2:35 pm"`)
    - A constructor that sets those variables
    - Getters for those variables
    - A method called `isEveningTime` that returns `true` or `false` depending on whether the time falls in the range from 6 to 10 pm.
    - A method called `bulkDiscount` that takes an integer n and returns the discount for buying n tickets. There should be a 10% discount for buying 5 to 9 tickets, and a 20% discount for buying 10 or more. Return these percentages numbers as integers.

184. Write a class called `MovieTicket` that inherits from the `Ticket` class of the previous problem. It should have the following (in addition to all that it gets from the `Ticket` class):

    - A string field called `movieName`

- A constructor that sets `movieName` as well as `cost` and `time`
- A method called `afternoonDiscount` that returns an integer percentage of 10% if the ticket time falls in the range from noon until (but not including) 6 pm.
- Override the `bulkDiscount` method so that there is a 10% discount for 10 tickets or more and no other discounts

185. Write a class called `Course` that has the following:

    - A private String field `name` that is the name of the course, a private integer `capacity` that is the maximum number of students allowed in the course, and a private list of strings called `studentIDs` representing the students by their ID numbers (stored as strings).
    - A constructor that takes the name of the course and capacity and sets those fields accordingly. The constructor should also initialize the `studentIDs` list, but it should not take a list as a parameter. It should only have the course name and capacity as parameters.
    - A method called `isFull` that takes no arguments and returns `true` or `false` based on whether or not the course is full (i.e. if the number of students in the course is equal to the capacity).
    - A method called `addStudent` that takes a student ID number (a string) and adds the student to the course by putting their ID number into the list. If the student is already in the course, they must *not* be added to the list, and if the course is full, the student must *not* be added to the course.

186. Write a class called `IntegerListUtilities` that contains the following static methods:

    - `average` — takes an integer list as an argument and returns the average of its entries (as a double)
    - `sample` — takes an integer list as an argument and an integer `n` and returns a list of `n` random items from the caller's list. There should be no repeated items in the returned list unless there were repeats in the caller's list. You might want to use the `Collections.shuffle` method here, but be sure only do this on a copy of the user's list as otherwise your method will modify the caller's list and they might not like that.

187. Write a class called `Timer` used to time things. It should have the following:

    - A private long value `initialTime` that holds information about the time.
    - A method `start` that starts the timing by saving the value of `System.currentTimeMillis()` to the `initialTime` field. It takes no parameters and returns nothing.
    - A method `elapsedSeconds` that returns the amount of time in seconds (as a double) that has passed since the timer was started.
    - A static method `formattedTime` that takes a time in seconds (a double), rounds to the nearest whole second, and converts it to minutes and seconds, and returns that in a string with the minutes and seconds separated by a colon. For instance, `formattedTime(131)` would return `"2:11"`.

188. Write a class called `TimerDriver` that tests out the class above. It should have the following:

    - Create a timer right at the start of the program.
    - Ask the user to type something.
    - Print out the amount of time in seconds that took, using `elapsedSeconds`
    - Then create another timer.
    - Ask the user to type some more text.
    - Print out the amount of time in seconds that took.
    - Print out the total amount of time passed since the first timer was started, using `elapsedSeconds` and formatting it with `getFormattedTime`

189. Add pausing capability to the Timer class so the timer can be paused and unpaused.

190. Create a class called `Coordinate` that has the following:

    - Integer fields `x` and `y`
    - A constructor to set those fields as well as getters and setters for them

- An `equals` method that considers two `Coordinate` objects equal if both of their fields are equal
- A `toString` method that returns a string containing the two fields separated by a comma and with parentheses, like `(2,3)` or `(14,9)`

191. Create a class called `Polygon` that has the following:

    - A field `vertices` that is a list of `Coordinate` objects. (This is a list of the polygon's vertices.)
    - A constructor that takes a list of `Coordinate` objects and sets `vertices` to a *copy* of that list.
    - A method called `numberOfVertices` that has no parameters and returns how may vertices the polygon has.
    - A method called `overlaps` that takes another `Polygon` object as a parameter and returns `true` or `false` depending on whether that polygon shares any vertices with this one.

192. Write a class called `TicTacToeBoard` that has the following:

    - A field that is a 3×3 array of Strings called `board`. Unused squares are represented by spaces and the other squares are represented by X's and O's.
    - A constructor that fills the array with spaces.
    - A method `makeMove(int row, int column, String type)` that sets the appropriate location in the board to either an `X` or an `O`, whichever is specified in `type`
    - A method `getLocation(int row, int column)` that returns the value stored in `board` at that location
    - A method `printBoard()` that nicely prints what the board currently looks like
    - A method `getWinner()` that returns the empty string if there is no winner, an `X` if there are three `X`'s in a row vertically, horizontally, or diagonally, and an `O` if there are three `O`'s in a row

193. Create a class that uses the `TicTacToe` class above to play a tic-tac-toe game.

194. Write a class called `GeneralPuzzle` that has the following:

    - A field called `board` that is a 2-dimensional array of integers
    - A field called `size` that represents the size of the board (assume the board has the same number of rows as columns)
    - A constructor that takes a 2-dimensional array and sets the `board` field to a copy of the 2-dimensional array and the sets the `size` field to the number of rows of the array.
    - A method `rowSum(r)` that sums up all the entries in row `r`
    - A method `columnSum(c)` that sums up all the entries in column `c`
    - A method `mainDiagonalSum()` that sums up all the entries in the main diagonal
    - A method `offDiagonalSum()` that sums up all the entries in the off diagonal
    - A method `rowDifferent(r)` that returns `true` if all the entries in row `r` are different and `false` otherwise
    - A method `columnDifferent(c)` that returns `true` if all the entries in column `c` are different and `false` otherwise
    - A method `toString()` that returns a string containing the data in `board`, formatted with one row per line, with columns separated by spaces

195. Write a class called `PuzzleInheritanceExample` that inherits from GeneralPuzzle and has the following:

    - A String field called `extra`
    - A constructor that takes a 2-dimensional array and a string and sets `board` and `size` like above (use `super` for this) and sets `extra` to the string
    - Override the `rowSum` method so that it returns the row sum modulo 2.

    This class is pretty pointless. It is just gives practice with some basic inheritance stuff.

196. Write a class `MagicSquare` that inherits from `GeneralPuzzle`. You will want to read up on magic squares online. http://www.jcu.edu/math/vignettes/magicsquares.htm is a good resource.

- A method `isMagicSquare()` that returns whether the array represents a magic square or not
- A method `getMagicSum()` that returns the magic sum of the magic square (assuming that it is a magic square)
- A method `constructMagicSquare()` that uses the algorithm described at the link above to generate a magic square. It only needs to work if the number of rows and columns is odd.

197. Write a class called `Sudoku` that inherits from `GeneralPuzzle`. It should have a method called `isCorrect()` that verifies that the array represents a solved `Sudoku`. For regular credit, your class should work with 9×9 Sudokus. For extra credit, it should work with $n^2 \times n^2$ Sudokus in general.

198. Write a class called `Calendar` with the following:

- a private integer field called `year` and a constructor that sets the year
- a boolean method `isLeapYear()` that returns whether the year is a leap year (see Exercise 13)
- a method `firstDay(m)` that returns the day of the week of the first day of the month `m` in the given year [Hint: see Exercise 12 for how to do this.]
- a method `printCalendar(m)` that prints out a calendar for that month in the given year. The calendar should look something like the one below, which would be for this month:

```
     1   2   3   4   5   6
 7   8   9  10  11  12  13
14  15  16  17  18  19  20
21  22  23  24  25  26  27
28  29  30
```

199. Java has a class called `BigInteger` that allows you to use arbitrarily large integers. For this exercise you will write your own very simple version of that called `BInteger`. The class should have the following:

- A large integer can be stored as an array of digits, so the class should have an integer array field called `digits` that stores the digits of the number.
- There should be a `toString` method that returns a string representation of the number. This will just be the digits from the integer array, one one immediately after another.
- The class should have a static method called `add` that takes two `BInteger` objects and adds them.

8.10 GUI exercises

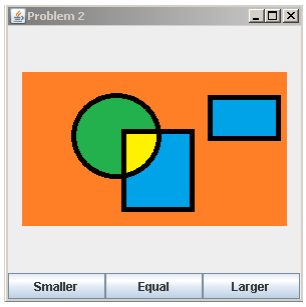
200. Write a GUI-based word-checking problem. There should be a place where the user enters a word and a button with the text, “Check word”. When the button is clicked, the program should read through a wordlist and output in a label whether or not the word is a real word.

201. Find or create images of the six sides of a die. Write a GUI program with a button that says "Roll" and a label that holds images of the die. When the user clicks the button, a random number from 1 to 6 should be generated and the label’s image should be changed appropriately.

202. Write a GUI program that has a place for the user to enter a bill amount and a place to enter a tip percentage (as a percentage, not a decimal). There should be a button labeled `Calculate` that calculates the total bill (bill + tip) and displays it in a label. Use `String.format` to format the amount to exactly two decimal places. (Getting user data will be covered in class on Monday 4/7 and is also in the online notes.)

203. Write a temperature converter program that allows the user to enter the temperature in a TextField. There should be two radio buttons that allow the user to decide whether the temperature should be converted from Fahrenheit to Celsius or vice-versa. The result should be displayed in a label.

204. Create a GUI that looks like the one below. It doesn’t have to do anything. In the center is an image from a file. Use any image file in its place.



205. Write a GUI-based program that allows the user to select what amount of red, green, and blue they want to use (from 0 to 255) and then displays the corresponding color. Your program should have the following:

    - Entry boxes for each of the three color types and labels to go with the entry boxes
    - A big label whose background color is the color given by the user's entries
    - A button that, when clicked, changes the color in that label.

206. Write a GUI program that creates 64 labels in an 8×8 grid. The labels should all have the text `"Hello"` and the colors of the label text should alternate in a checkerboard pattern between red and blue. Use lists or arrays.

207. Create a tri-colored checkerboard like the one below.

208. Write a program that draws a stick figure. Your stick figure should, at the very least, have a head, torso, two legs, and two arms.

209. Create a 9×9 board of buttons. You can use a list of 81 buttons or a 9×9 array of them. Each time a button is clicked its background color should alternate between red and blue.

210. Write a GUI program with checkboxes labeled 1, 2, 3, 4, and 5. Each time a box is checked or unchecked, a label should be updated. The label should display which boxes are checked and which are unchecked as a set. For instance, if boxes 2, 4, and 5 are checked and the others unchecked, the label should show $\{2, 4, 5\}$.

211. Write a GUI program with 26 checkboxes labeled `A` through `Z`. Each time a box is checked or unchecked, a label should be updated. The label should display which boxes are checked and which are unchecked. For instance, if boxes A, C, and Z are checked and the others unchecked, the label should show `ACZ`.

212. The following formula tells you how many payments are needed to pay off a loan of P dollars at interest rate r when making payments of A dollars at a time.

$$N = \frac{-\ln(1 - rP/A)}{\ln(1 + r)}$$

    Create sliders allowing the user to vary r, P, and A, and each time a slider is moved, display the new value of the formula in a label.

213. Write a converter program that allows the user to enter a length measurement. There should be two sets of radio buttons — one set to indicate the current unit and the other to indicate the desired unit. The result should be displayed in a label. Have several possible units.

214. Write a GUI day-of-the-week calculator. The program should have:

    - A message telling the user to input a date in the form mm/dd/yyyy.

- A button that when clicked calculates the day of the week of that date in history. See Exercise 12 for how to do that.
- A label displaying the result of the calculation.

215. Write a GUI program that has:

    - Five labels that represent dice. The labels hold numbers from 1 to 6 (or dice images if you prefer).
    - Five checkboxes, each one below the corresponding dice label.
    - A Roll button. When it's clicked, it will randomly "roll" the five "dice" by setting them to random numbers from 1 to 6.
    - However, if the checkbox under the corresponding dice label is checked, then the corresponding dice label should not be changed when the Roll button is clicked.

    It's a little like the game *Yahtzee*, where the player sets aside certain dice and only rolls the others. The user indicates they don't want to roll certain dice by selecting the checkboxes.

216. Write a GUI program where the board has a 5×5 grid of buttons, each initially filled with a random letter. Whenever the user clicks on a button, it adds the corresponding letter to a string and displays the string in a label. There should also be a Check Word button that the user can use to see if the string contains a real word. (This is basically halfway to a Boggle game.)

217. Write a GUI tic-tac-toe game. The board should be a 3×3 grid of cells that are buttons. The X player goes first, followed by the O player. At each player's turn, when a button is clicked, an X or an O should be put into the button's text.

218. Write a GUI-based guess-a-number game. The player should get a certain number of guesses and after each guess the program should tell them if their guess is higher, lower, or correct. At the end, there should be a message indicating whether the player has won or lost.

219. Add a replay button to the guess-a-number game. The game should keep track of how many times the player won and how many times they lost.

220. Write a GUI-based Rock-Paper-Scissors game that allows the user to play against the computer. The first one to win three rounds is the winner. Your program should say who wins each round and who wins the overall match.

221. Write a very simple, but functional, GUI calculator. It should have buttons for the numbers 0-9, a plus key, and an equals key.

222. Write a GUI-based multiplication game. Your game should do the following:

    - Randomly generate multiplication problems and display them prominently
    - Have an entry box for the user to enter their guess
    - Tell them if they got the problem right and what the answer was if they got it wrong
    - Display how many problems the user has tried and how many they've gotten right.

223. Write a program to play the following puzzle:

    - The board is a 4×4 grid of blue and yellow cells.
    - All the cells are yellow to start and your goal is to make all the cells blue.
    - Each time you click a cell, it changes to the other color.
    - But the trick is that all four of its neighbors (above, below, left, and right) also change their color.

224. Write a GUI-based slider puzzle game. The way the user interface should work is that when the user clicks on a number adjacent to the space, that number swaps places with the space.

Index