A Quick Intro to Java

©2015 Brian Heinold
Licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License
Preface

his 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.

Last updated: May 20, 2015.
# Contents

1 Basics ............................................. 1
   1.1 Installing Java ......................................... 1
   1.2 Basics of Java syntax ................................. 2
   1.3 Output ............................................... 3
   1.4 Variables ............................................ 3
   1.5 Operators ............................................ 4
   1.6 For loops ............................................. 4
   1.7 If statements ......................................... 5
   1.8 Importing things ................................. 6
   1.9 Input ............................................... 6
   1.10 Math functions .............................. 6
   1.11 Random numbers ........................... 7
   1.12 While loops ....................................... 7
   1.13 Strings ............................................ 7
   1.14 ASCII codes and escape characters ........ 10
   1.15 Arrays .............................................. 11
   1.16 Lists ............................................... 12
   1.17 Two-dimensional arrays ............... 14
   1.18 Arrays versus lists .......................... 15
   1.19 Reading and writing to text files .......... 16

2 Miscellaneous Programming Topics ............ 18
   2.1 Type casts ......................................... 18
   2.2 The split method ................................... 18
   2.3 Uses for scanners .................................. 19
   2.4 Short-circuiting ..................................... 20
   2.5 Scope of variable declarations .............. 20
   2.6 Infinite loops ..................................... 20
   2.7 Switch-case .......................................... 20
   2.8 printf .............................................. 21
   2.9 The break and continue statements ........ 22
   2.10 The ternary operator ......................... 23
   2.11 Do...while Loop ................................... 23
   2.12 Boolean expressions ......................... 23
2.13 Useful character methods .................................................. 24
2.14 Lists of lists ....................................................................... 24
2.15 Sets .................................................................................. 24
2.16 Maps ................................................................................ 25
2.17 StringBuilder .................................................................. 27
2.18 Timing and sleeping ............................................................. 27
2.19 Dates and times ................................................................. 28
2.20 Threads ............................................................................. 29

3 Common Programming Techniques ......................................... 31
3.1 Counting ............................................................................ 31
3.2 Summing ............................................................................ 31
3.3 Maxes and mins .................................................................. 31
3.4 The modulo operator ........................................................... 32
3.5 Nested loops ....................................................................... 33
3.6 Flag variables ..................................................................... 34
3.7 Working with variables ........................................................ 35
3.8 Using lists to shorten code .................................................. 35

4 Object-Oriented Programming .................................................. 37
4.1 Introduction ......................................................................... 37
4.2 Functions .......................................................................... 37
4.3 Examples of functions ......................................................... 38
4.4 More details about functions ............................................... 39
4.5 Classes ............................................................................. 40
4.6 Object-oriented concepts ..................................................... 44
4.7 Inheritance ......................................................................... 51
4.8 Wrapper classes and generics .............................................. 55
4.9 References and garbage collection ...................................... 56
4.10 Interfaces .......................................................................... 58
4.11 Nested classes ................................................................... 59
4.12 Exceptions ........................................................................ 60

5 GUI Programming .................................................................... 62
5.1 A template for other GUI Programs ..................................... 62
5.2 A Hello World program ....................................................... 63
5.3 Customizing labels and other widgets ................................. 64
5.4 Buttons .............................................................................. 65
5.5 Text fields .......................................................................... 66
5.6 Layout managers ............................................................... 68
5.7 Checkboxes ....................................................................... 71
5.8 Radio buttons ..................................................................... 72
5.9 Sliders ............................................................................... 73
5.10 More about ActionListeners ............................................. 75
5.11 Simple graphics .......................................................... 76
5.12 Timers ................................................................. 78
5.13 Keyboard input ............................................................ 79
5.14 Miscellaneous topics ....................................................... 80
5.15 Example GUI programs .................................................. 83
5.16 Further GUI programming ............................................. 92

6 Common Gotchas ............................................................... 93
   6.1 Simple debugging ......................................................... 93
6.2 Common exceptions ....................................................... 93
6.3 Lengths of strings, arrays, and lists ................................... 95
6.4 Misplaced semicolons ...................................................... 95
6.5 Characters and strings .................................................... 95
6.6 Counting problems ......................................................... 95
6.7 Problems with scanners .................................................. 96
6.8 Problems with logic and if statements ................................. 96
6.9 Problems with lists ......................................................... 97
6.10 Functions that should return a value but don’t ....................... 98
6.11 Problems with references and variables ............................. 98
6.12 Numbers ................................................................ 99

7 A Few Other Topics ............................................................ 101
   7.1 Java’s history ............................................................... 101
6.2 The Java Virtual Machine (JVM) ........................................ 101
6.3 Running your programs on other computers .......................... 102
6.4 Getting help on Java ....................................................... 102
6.5 Whitespace, braces, and naming conventions ....................... 102
Chapter 1

Basics

1.1 Installing Java

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


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”:

```java
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,

  ```java
  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,

```java
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:

```java
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:

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

To print variables, we can do something like below:

```java
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,

```java
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 \(-9 \times 10^{18}\) 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, 0.5) 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:

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

<table>
<thead>
<tr>
<th>Normal</th>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x + 1</td>
<td>x++</td>
</tr>
<tr>
<td>x = x - 1</td>
<td>x--</td>
</tr>
<tr>
<td>x = x + 3</td>
<td>x += 3</td>
</tr>
<tr>
<td>x = x * 3</td>
<td>x *= 3</td>
</tr>
</tbody>
</table>

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

1.6 For loops

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

```java
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:

```java
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:

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

### 1.7 If statements

Here is a typical if statement:

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

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

```java
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:

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

### Logical and comparison operators

The logical and comparison operators are below:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>not</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
</tr>
</tbody>
</table>
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:

```java
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:

```java
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:

```java
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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>absolute value</td>
</tr>
<tr>
<td>max(x, y)</td>
<td>returns the larger of x and y</td>
</tr>
<tr>
<td>min(x, y)</td>
<td>returns the smaller of x and y</td>
</tr>
<tr>
<td>round(x)</td>
<td>rounds x to the nearest integer</td>
</tr>
<tr>
<td>floor(x)</td>
<td>returns the greatest integer ≤ x</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>returns the least integer ≥ x</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>returns x^y</td>
</tr>
</tbody>
</table>

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:

```java
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:

```java
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.

```java
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:

```java
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>returns the string's length</td>
</tr>
<tr>
<td>charAt(i)</td>
<td>returns the character at index ( i )</td>
</tr>
<tr>
<td>indexOf(s)</td>
<td>returns the location of the first occurrence of the character/substring ( s )</td>
</tr>
<tr>
<td>equals(s)</td>
<td>returns whether the string equals ( s )</td>
</tr>
<tr>
<td>equalsIgnoreCase(s)</td>
<td>like above, but ignores upper/lowercase</td>
</tr>
<tr>
<td>substring(i)</td>
<td>returns the substring from index ( i ) to the end</td>
</tr>
</tbody>
</table>
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:
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:

```java
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:

```java
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:

```java
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:

```java
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:

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

Looping through strings

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

```java
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`:

```java
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.

```java
String capitalized = ""
for (int i=0; i<s.length(); i++)
```
```java
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.

```java
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.

<table>
<thead>
<tr>
<th>32</th>
<th>space</th>
<th>44</th>
<th>65–90</th>
<th>A–Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>!</td>
<td>45</td>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>46</td>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>47</td>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>48–57</td>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>58</td>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>59</td>
<td>96</td>
<td>'</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>60</td>
<td>97–122</td>
<td>a–z</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>61</td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>62</td>
<td>124</td>
<td>}</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>63</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>64</td>
<td>126</td>
<td>~</td>
</tr>
</tbody>
</table>

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

```java
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.

```java
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.

```java
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:
A few examples

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

```java
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`:

```java
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:

```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:

```java
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.

```java
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:

```java
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:

```java
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 be called with an index to insert an item into a list. Here is an example that inserts a 99 at index 1.

```java
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:

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

### Making a copy of a list

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

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

### List methods

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contains(x)</code></td>
<td>returns whether x is in the list</td>
</tr>
<tr>
<td><code>indexOf(x)</code></td>
<td>returns the location of the first occurrence of x in the list</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns whether the list is empty</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in the list</td>
</tr>
<tr>
<td><code>subList(i,j)</code></td>
<td>returns a slice of a list from index i to j-1, sort of like substring</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addAll(list,x1,...,xn)</code></td>
<td>adds x1, x2, and xn to list</td>
</tr>
<tr>
<td><code>binarySearch(list,x)</code></td>
<td>returns whether x is in list (list must be sorted)</td>
</tr>
<tr>
<td><code>max(list)</code></td>
<td>returns the largest element in list</td>
</tr>
<tr>
<td><code>min(list)</code></td>
<td>returns the smallest element in list</td>
</tr>
<tr>
<td><code>reverse(list)</code></td>
<td>reverses the elements of list</td>
</tr>
<tr>
<td><code>shuffle(list)</code></td>
<td>puts the contents of list in a random order</td>
</tr>
<tr>
<td><code>sort(list)</code></td>
<td>sorts list</td>
</tr>
</tbody>
</table>
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:

```java
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:

```java
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):

```java
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:

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

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

```java
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):

```java
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:

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

- Adding up the entries in the third column:
  ```java
  int total = 0;
  for (int i=0; i<a.length; i++)
    total += a[i][2];
  ```

- Adding up the entries along the diagonal:
  ```java
  int total = 0;
  for (int i=0; i<a.length; i++)
    total += a[i][i];
  ```

- Adding up the entries below the diagonal:
  ```java
  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:

- Arrays:
  ```java
  a[i]++;  
  total += a[i][j];
  ```

- Lists:
  ```java
  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.

- Arrays:
  ```java
  a[i][j]++;  
  total += a[i][j];
  ```

- Lists:
  ```java
  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:

```java
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:

```java
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:

```java
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:

```java
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.

```java
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:

```text
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:

```java
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:

```java
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:

```java
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):

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
String[] a = s.split(",|;"); // split at commas or semicolons
String[] a = s.split("\W+"); // 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:

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

```java
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:

```java
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 used 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:

```java
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

Suppose you need to check to see if \( a[i] \) is positive, where \( a \) is an array and \( i \) is a positive 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:

\[
\text{if} \ (i < a\text{.length} \ \&\& \ a[i] > 0)
\]

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 \( a\text{.length} \), 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:

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

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:

```java
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:

```java
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:

```java
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:

```java
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f</td>
<td>for floating point numbers</td>
</tr>
<tr>
<td>%d</td>
<td>for integers</td>
</tr>
<tr>
<td>%s</td>
<td>for strings</td>
</tr>
<tr>
<td>%g</td>
<td>chooses between regular or scientific notation for floats</td>
</tr>
</tbody>
</table>

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

```java
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);
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>17</td>
<td>2203</td>
</tr>
</tbody>
</table>

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.

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

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

```java
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
```
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:

```java
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:

```java
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:

```java
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:

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

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

```java
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:

```java
if (x == 1)
    System.out.println("There is 1 item.");
else
    System.out.println("There are " + x + "items.");
```

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

```java
System.out.println("There are " + 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:

```java
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.

```java
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:

```java
public static boolean isEvenString(String s)
{
    return s.length() % 2 == 0;
}
```
2.13 Useful character methods

To check if a character \( c \) is a digit, letter, both, or a whitespace character use the following:

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

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:

```java
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:

```java
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:

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

2.15 Sets

A set is a data structure that is a like a list with no repeated elements.

We declare a set in a similar way to how we declare a list:

```java
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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)</td>
<td>adds ( x ) to the set</td>
</tr>
<tr>
<td>contains(x)</td>
<td>returns whether ( x ) is in the set</td>
</tr>
<tr>
<td>difference(t)</td>
<td>returns a set of the elements of ( s ) not in the set ( t )</td>
</tr>
<tr>
<td>intersection(t)</td>
<td>returns a set of the elements in both ( s ) and ( t )</td>
</tr>
<tr>
<td>remove(x)</td>
<td>removes ( x ) from the set</td>
</tr>
<tr>
<td>union(t)</td>
<td>returns a set of all the elements that are in ( s ) or ( t )</td>
</tr>
</tbody>
</table>

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

```java
for (int x : set)
    // do something with \( x \)
```
A useful example

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

```java
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:

```java
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:

```java
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 \textit{bad} / \textit{dab} or \textit{diaper} / \textit{repaid}. We can use the code above to fill the list/set. The code below will then find all the desired words:

```java
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 \textit{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:

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

Checking if a list contains repeats

Suppose we want to know if an integer list called \textit{list} contains repeats. The following line uses a set for this purpose:

```java
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 we check 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:

```java
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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>containsKey(k)</td>
<td>returns whether k is a key in the map</td>
</tr>
<tr>
<td>containsValue(v)</td>
<td>returns whether v is a value in the map</td>
</tr>
<tr>
<td>get(k)</td>
<td>returns the value associated with the key k</td>
</tr>
<tr>
<td>put(k,v)</td>
<td>adds the key-value pair (k,v) into the map</td>
</tr>
</tbody>
</table>

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

```java
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:

```java
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:

```java
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.

```java
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:

```java
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, 1);
}
```

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

```java
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:

```java
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.

```java
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:

```java
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:

```java
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:

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

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

```java
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:

```java
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:

```java
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:

```java
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:

<table>
<thead>
<tr>
<th>Thursday December 25, 2014 1:45 PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/25/2014 13:45</td>
</tr>
</tbody>
</table>
Arithmetic with dates

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

\[
d = d.\text{plusDays}(30));
\]

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

\[
\text{Period period = Period.between(} \text{LocalDate.of(2014, 12, 25), LocalDate.now());}
\]

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

\[
\text{Duration duration = Duration.between(} \text{LocalTime.of(} 14, 25)\text{), LocalTime.now());}
\]

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 \textit{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:

```java
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:

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

```java
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:

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

The technique used 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 by adding a space after every fifth character.

```java
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.

```java
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.

```java
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:

```java
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`:

```java
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:
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:

```java
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:

```java
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.

<table>
<thead>
<tr>
<th>(0,0)</th>
<th>0</th>
<th>(0,1)</th>
<th>1</th>
<th>(0,2)</th>
<th>2</th>
<th>(0,3)</th>
<th>3</th>
<th>(0,4)</th>
<th>4</th>
<th>(0,5)</th>
<th>5</th>
<th>(0,6)</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,0)</td>
<td>7</td>
<td>(1,1)</td>
<td>8</td>
<td>(1,2)</td>
<td>9</td>
<td>(1,3)</td>
<td>10</td>
<td>(1,4)</td>
<td>11</td>
<td>(1,5)</td>
<td>12</td>
<td>(1,6)</td>
<td>13</td>
</tr>
<tr>
<td>(2,0)</td>
<td>14</td>
<td>(2,1)</td>
<td>15</td>
<td>(2,2)</td>
<td>16</td>
<td>(2,3)</td>
<td>17</td>
<td>(2,4)</td>
<td>18</td>
<td>(2,5)</td>
<td>19</td>
<td>(2,6)</td>
<td>20</td>
</tr>
<tr>
<td>(3,0)</td>
<td>21</td>
<td>(3,1)</td>
<td>22</td>
<td>(3,2)</td>
<td>23</td>
<td>(3,3)</td>
<td>24</td>
<td>(3,4)</td>
<td>25</td>
<td>(3,5)</td>
<td>26</td>
<td>(3,6)</td>
<td>27</td>
</tr>
</tbody>
</table>

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.

**3.5 Nested loops**

Consider the following code:

```java
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:

```java
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:

```java
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.

```java
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.

```java
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:

```java
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:

```java
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:

```java
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:

```java
List<String> names = new ArrayList<String>();
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:

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

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

```java
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.

```java
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:

   ```java
   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):

   ```java
   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:

   ```java
   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:

   ```java
   printBlankLines(3);
   ```

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

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

   Here are some ways to call this function:

   ```java
   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.

   ```java
   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:

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

   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()
{
    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.

```java
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:

```java
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 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.

```java
public class Deck {
    private List<Card> cards;
```
The main part of this class is a list of Card objects that represents the deck. The constructor fills up that
desk 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:

```java
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.

```java
public Player(String name, int hitPoints, int attackPower)
```
When the following line is run from another class, the Player constructor is called.

```java
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.

```java
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:

```java
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:

```java
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:

```java
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:

```java
@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:

```java
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:

```java
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:

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

<table>
<thead>
<tr>
<th></th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>protected</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>no modifier</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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`:

```java
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 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:

```java
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:

```java
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:

```java
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:

```java
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.

```java
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.

```java
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.

```java
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:

```java
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`:

```java
public class StaticVariableExample
{
    public static int x;
    public StaticVariableExample(int x)
    {
        this.x = x;
    }
}
```


```java
{    this.x = x;
}

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

Suppose we test the class as follows:

```java
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.

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

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

myClass.printHello();
```

```java
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`.

```java
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:

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

    public int specialPower()
    {
        Random random = new Random();
```
```java
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.

53
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 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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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

```java
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

```java
boolean b = true;
```

the value true is stored in at the precise memory location of the variable b.

Now consider the following code:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

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

We can then create some classes that implement the interface:

```java
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:

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

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

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.

```java
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:

```java
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 ((Integer)value).compareTo(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. Since we are comparing cards based on their values, we use a sneaky trick here to piggyback off the Integer class's compareTo method.

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`.

```java
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 to sort the list. Collections.sort 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.

Interfaces are used a lot in GUI programs, as we will see.

### 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. It 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:

```java
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:

```java
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:

```java
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:

```java
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("Error reading file. List will be empty");
}
```
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:

```java
if (size < 0)
    throw new RuntimeException("The list is empty.");
```

You can also throw any exception already defined in Java, like `IndexOutOfBoundsException` 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:

```java
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.

```java
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:

![Image of Hello, World! label]

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:

```java
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:

```java
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.

```java
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:

```java
label.setBackground(Color.BLUE);
label.setOpaque(true);
```

#### Getting and setting the text

To change the text in the label, do something like the following:

```java
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:

```java
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 class.

Here is an example program that has a button and a label whose text changes once the button has been clicked:

```java
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);
    }

    private class ButtonListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent arg0) {
            label.setText("Button has been clicked");
        }
    }
}
```
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 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()
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:

```java
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:

```java
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:

![Grid layout with Xs and Os](image_url)

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:

```java
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);
```
mainPanel.add(button3, BorderLayout.EAST);
mainPanel.add(button4, BorderLayout.WEST);
mainPanel.add(button5, BorderLayout.CENTER);

The figure below shows what this would look like:

![Image](image.png)

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 do 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:

```java
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:

```java
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:

```java
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)
        {
            // Handle button click event
        }
    }
}
```
if (checkBox.isSelected())
    label.setText("Selected!");
else
    label.setText("Not selected!");

This is what it looks like on my machine:

![Image of Checkbox Demo]

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:

```java
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:

```java
    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.

To 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:

![Image of the GUI interface showing sliders and a color label](image-url)
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:

```java
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:

```java
private class ButtonListener implements ActionListener {
    private int numClicks;
    public ButtonListener() {
        numClicks = 0;
    }
    @Override
    public void actionPerformed(ActionEvent arg0) {
        numClicks++;
        label.setText(numClicks + " 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:

```java
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.

```java
button.addActionListener(e -> {
    label.setText("Clicked!");
    label.setForeground(Color.RED);
});
```
5.11 Simple graphics

Suppose we want to draw things like lines and circles on the screen. What we'll do is 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:

```java
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 takes 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:

```java
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.

```java
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:

```java
try {
    myImage = ImageIO.read(new File("someImage.jpg"));
} catch (IOException e) {e.printStackTrace();}
```

Then in the `paintComponent` method, we would have something like this:

```java
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:

```java
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:

```java
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()
    {
```
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:

```java
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:

```java
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 implementsKeyListener
{
    @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:

```java
TextArea = new JTextArea("", 80, 20);
```

To attach a scrollbar to the area, use the following:

```java
scrollpane = new JScrollPane(textArea);
```

Then, when adding the text area to the screen, add the ScrollPane instead of the JTextArea. For instance:

```java
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.

![Image of list and combo box]

Here is the code to create them:

```java
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:

```java
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 \((x,y)\).

```java
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:

```java
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:

```java
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();
    }
}
```
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.setVisible(true);
    }
}
```java
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 the guess from the entry box, check if it's right or not, and 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("");
    }
}
```
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 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.

```java
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));

// 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, then 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);
}

g: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.

```java
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:

![Screenshot of the program](image)

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.

```java
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;
        }
    }
}
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:

```java
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:

```java
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 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:

```java
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:

```java
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.

```java
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.

```java
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:

```java
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:

```java
for (String word : words)
{
    int count = 0;
    for (int i=0; i<words.length(); i++)
```
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:

```java
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 redefine the scanner.

.Scanner.nextLine( ) vs Scanner.next()

When asking the user to enter a string, use Scanner.nextLine:

```java
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:

```java
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:

```java
if (s.equals("abc"))
```

if instead of else if

Consider the following program:

```java
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:

```java
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:

```java
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:

```java
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:

```java
for (int x : list) // do something with x
for (int i=0; i<list.size(); i++) // 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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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. 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:

```java
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).

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:

```java
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 $-9 \times 10^{18}$ to $9 \times 10^{18}$.

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⋯. 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 0.5999999999999998.

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.

```java
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.

```java
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:

```java
if (Math.abs(x-y)<.000000001)
```

The code above considers two doubles as equal provided they are within .000000001 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
```
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.”