

Introduction to Programming with Java

By G. Ridout

Version 3.1

© September 2014

Table of Contents

Preface.....	iv
Chapter 1 - Introduction to Computer Languages and Java	1
1.1 Computer Programming Languages	1
1.2 The Java Programming Language	3
1.3 Translating Java Code to Machine Code	4
1.4 The JDK and Java IDEs	5
1.5 A Simple Java Program in Eclipse.....	6
1.6 Problem Solving with Computers	11
1.7 Chapter Summary	12
1.8 Questions, Exercises and Problems	12
Chapter 2 – Variables, Input/Output and Calculations	13
2.1 Declaring Variables and Primitive Types	13
2.2 Declaring Strings in Java	15
2.3 Using the Scanner Class to Input Information.....	17
2.4 Formatted Output using <code>System.out.printf()</code>	20
2.5 Arithmetic Expressions in Java.....	22
2.6 Type Conversion (Casting and Promotion)	25
2.7 Math and Character Class Methods and Constants.....	26
2.8 Chapter Summary	28
2.9 Questions, Exercises and Problems	29
Chapter 3 - Java Control Structures	33
3.1 Boolean Expressions (Conditions).....	33
3.2 Selection Structures	34
3.3 Conditional Loops.....	37
3.4 Counter Loops.....	39
3.5 Example Program using Selection and Loop Structures.....	40
3.6 Pseudocode and Flowcharts	42
3.7 Chapter Summary	45
3.8 Questions, Exercises and Problems	45
Chapter 4 –Methods	51
4.1 Introduction to Methods.....	51
4.2 Instance Methods vs. Static Methods.....	52
4.3 Writing Your Own Static Methods	53
4.4 Method Overloading	58
4.5 More on Method Parameters.....	59
4.6 Chapter Summary	60
4.7 Questions, Exercises and Problems	60

Chapter 5 - Working with Strings.....	65
5.1 Creating and Initializing a New String Object.....	65
5.2 Comparing Strings	66
5.3 Looking at Strings, One Character at a Time	67
5.4 Creating Substrings.....	68
5.5 Searching Strings	68
5.6 Other String Methods.....	70
5.7 Converting Other Types to and from Strings.....	70
5.8 Using Scanners to break up a String	72
5.9 The StringTokenizer Class.....	73
5.10 Chapter Summary	75
5.11 Questions, Exercises and Problems	76
Chapter 6 – Creating New Classes in Java	81
6.1 Encapsulation.....	81
6.2 Example: A Car class	82
6.3 Another Example: Designing a Time object	87
6.4 Java code for a Time object	89
6.5 Inheritance.....	91
6.6 Chapter Summary	93
6.7 Questions, Exercises and Problems	93
Chapter 7 - Arrays.....	97
7.1 Introduction to Arrays.....	97
7.2 Example Program Using Arrays	99
7.3 Initializing Arrays	100
7.4 Related (Parallel) Arrays.....	101
7.5 Arrays of Objects (An Alternative to Related Arrays)	103
7.6 Two Dimensional Arrays (Arrays of Arrays)	105
7.7 Arrays and Methods	107
7.8 Chapter Summary	108
7.9 Questions, Exercises and Problems	109
Chapter 8 - Robot World.....	119
8.1 Creating a New Robot World.....	119
8.2 Creating a New Robot	121
8.3 Overview of Robot Methods	123
8.4 Working with Items	126
8.5 RobotPlus: Extending the Robot Class.....	128
8.6 RobotTrack: Example Code for an extended Robot	131
8.7 Chapter Summary	133
8.8 Questions, Exercises and Problems	134

Appendix A -- Java Style Guide	139
A.1 Naming Identifiers	139
A.2 Program Comments.....	140
A.3 Code Paragraphing and Format.....	142
Appendix B -- Finding and Correcting Program Errors	143
B.1 Types of Errors in Java Programs	143
B.2 Tips to Avoid Errors.....	146
B.3 Using Test Data to Find Errors	147
Appendix C -- Unicode Characters	148
C.1 Unicode Characters Displayable in the Eclipse Console	148
Appendix D -- The Math and Character Classes	149
D.1 The Math Class in Java.....	149
D.2 The Character Class in Java.....	150
Appendix E -- Reading from and Writing to Text Files.....	151
E.1 Using a Scanner to read from a text file	151
E.2 Using a BufferedReader to read from a text file	152
E.3 Using a PrintWriter to write to a text file.....	153
Appendix F -- Robot World Methods and Constants	154
F.1 The World Class	154
F.2 The Robot Class	154
F.3 The Direction Class.....	156
F.4 The Item Class.....	156
Appendix G -- Exceptions and Exception Handling.....	157
References.....	159

Preface

The material presented in this text was originally used to introduce the Java programming language to Grade 11 Computer Science students. It has also been used in Grade 12 classes as a resource for students using Java for the first time. Although experience in another programming language such as Turing would be helpful in working through this text, it is not required.

This text makes no attempt to cover all of the material required for the Grade 11 Computer Science course. It does not cover the hardware, social issues, careers and problem-solving sections of this course. Also, since this is an introductory text, it does not cover Java applets or applications, although these topics are usually presented in the grade 11 course to help students with their final projects.

Since this text is a work in progress any corrections, comments or suggestions would be greatly appreciated.

Finally I would like to thank all of the students who I have taught and colleagues I have worked with at Richmond Hill High School who have given me valuable feedback, suggestions and comments on the material presented. I would also like to thank former students and colleagues at Thornhill S. S., Bayview S. S. and Dr. John M. Denison S. S. who have given me ideas and feedback over the years that have helped in developing this resource.

G. Ridout
September 2014

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Chapter 1 - Introduction to Computer Languages and Java

One of the things that make the electronic computer such a powerful and versatile tool is that it can be programmed to perform a variety of different tasks. When we look at a computer we see the hardware (monitor, keyboard and CPU etc.), but equally important is the software (programs) that instruct and control the hardware.

1.1 Computer Programming Languages

Each computer processor (e.g. Pentium) has a unique set of instructions that it understands. These instructions are used to tell the processor to perform calculations, make decisions and to interact with its memory and peripherals (input/output devices). Since computers work in binary (a series of 1's and 0's), the instructions given to the computer must eventually be translated into this binary **machine language**. In fact, early programs were written directly in machine language. However, trying to keep track of all of the instructions using only numbers as well as keeping track of memory locations in binary was very difficult. Therefore, programmers generally no longer program in machine language.

To make it easier for programmers to write programs, **assembly language** was developed. Assembly language is very close to machine language except it uses a series of mnemonic codes instead of binary codes for each instruction. The mnemonic codes are English short forms that help remind the programmer what each instruction does. For example the assembly language code to tell the processor to add two numbers is "add". A complete instruction, which includes what you want to add, would be:

```
add  ax, 12      ; add 12 to the ax register
```

In this case the `ax` register is a special memory location. Other examples of assembly language instructions are shown below:

```
mov  dx, 0       ; move 0 into the dx register
jmp  Start       ; jump to the section labelled Start
inc  [count]     ; add one to a memory location called count
```

In these instructions, the mnemonic codes `mov`, `jmp` and `inc` are short for move, jump and increment respectively.

Writing code using assembly language mnemonics is a lot easier than writing code directly in machine language. However, since the computer still only understands machine language, the assembly code needs to be translated into machine language using a special program called an **assembler**. In most cases each assembly language instruction translates one to one into a corresponding machine language instruction.

Since assembly language instructions directly translate into the machine instructions for a particular machine, they are processor specific. For example, the sample instructions given above are for an Intel processor. If you wanted to program a Motorola processor you would need to use a different set of assembly language instructions. Even though some of the instructions may be similar, an assembly language program written for one processor will not work on another processor.

Although assembly languages are not very portable, they allow the programmer to use processor specific commands that can give more direct control of the processor and other computer hardware. This is one of the main reasons why some programmers still use assembly language today. To make it easier to write sections of code in assembly language, high-level languages such as C and C++ allow you to put assembly language instructions directly into your programs.

In addition to their lack of portability, assembly language programs can be very cryptic and hard to follow. Furthermore, when writing assembly language programs, the programmer is forced to think of a problem in terms of how the computer will solve the problem rather than how the programmer would solve the problem. These disadvantages lead to the development of **high-level computer programming languages**.

High-level languages allowed programmers to write code that was closer to the way we think of a problem. For example:

How we think of a problem	High-Level Computer Code (Java)
If the total sales are greater than \$100, calculate a discount that is 10% of these sales.	<pre>if (totalSales > 100) discount = totalSales * 0.10;</pre>

Computer code that is closer to the way we think of a problem is both easier to write and to understand. However, since the computer still only understands machine language instructions, we must translate the high-level language instructions into machine language before we can run a program.

To translate high-level languages into machine language we use special programs called **compilers**. Since each single high-level instruction usually does more than a single assembly language instruction, each instruction could translate into many machine language instructions. For example, the above Java code would translate into more than 4 machine language instructions. Therefore, when translating high-level languages into machine language it is important to select a compiler that generates clean and efficient code.

Two early high-level languages were COBOL (COMmon Business Oriented Language) and FORTRAN (FORmula TRANslator). COBOL was mostly used for business programming and FORTRAN was used for scientific programming. Since many programs written in COBOL and FORTRAN are still in use today, these languages continue to be used.

Other high-level languages developed over the years include Ada, Algol, BASIC (Beginner All-purpose Symbolic Instruction Code), Pascal, C and Turing.

As programs became even larger, **object-oriented languages** were developed to make coding easier. Since the world we live in is made up of different objects, by using a programming language that can easily manipulate these objects, programming becomes more of a natural process.

To solve a problem using object-oriented programming (OOP), we first consider what objects are needed to solve the problem. Next, we define each object's characteristics and behaviour. Then we write the code to define each object. In some cases, we can re-use previously created objects without writing new code. In other cases, we can use a technique called **inheritance** to easily extend an existing object to create a new but similar object.

Once the code for all of the objects has been created, we then write the code to get these objects to interact with each other. When properly designed and implemented object-oriented programs closely match the real life problem we were trying to solve. Code that is closer to the way we think is easier to write, debug (find errors in) and maintain.

Some examples of object-oriented languages include Smalltalk, Object Pascal, C++, C#, Java, Objective C, and Python.

1.2 The Java Programming Language

The Java language was developed at Sun Microsystems in 1995. The original idea was to develop a language that could be used to program the smart appliances of the future. According to an overview of the Java Language originally posted on Sun's web site, the Java language was described as follows:

"Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language".

One of the key advantages of Java is a runtime environment that provides platform independence. Therefore you can download the same code over the Internet to run on different platforms such as Windows, Solaris and Linux.

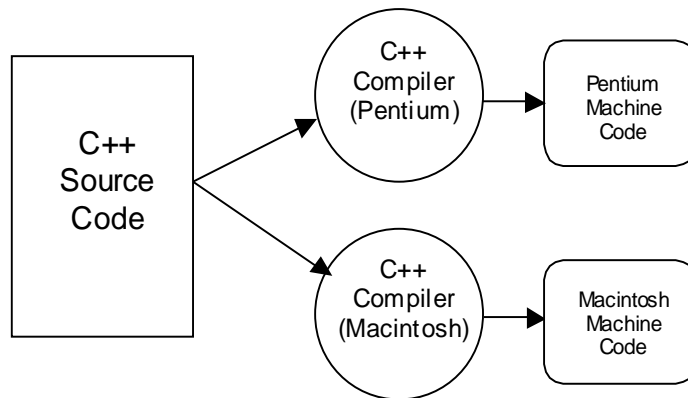
To make the transition to Java easier for programmers using C or C++ a lot of Java's syntax is based on these languages. Additional features were added to avoid some of the common problems found in C and C++. In particular, Java added automatic garbage collection (more on this later) and stronger type checking.

Although a lot of Java's success has been through the Internet, more recently it has returned to its roots with Java being the primary language used to write programs for cell phones and tablets using Google's Android Operating System. In 2010, Sun Microsystems was purchased by Oracle who is now responsible for maintaining the Java Language.

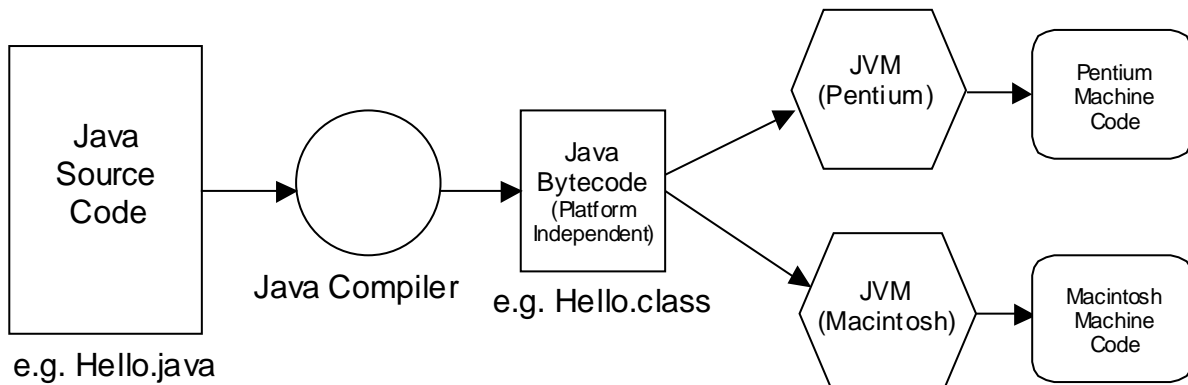
1.3 Translating Java Code to Machine Code

One of the big differences between Java and other languages is how the code is compiled into machine code.

When you translate (compile) a program written in C++ or most other high-level languages, the compiler translates your program (source code) into machine code (object code) – instructions that are specific to the processor your computer is running (e.g. Pentium). If you want to run your program on another platform (e.g. Macintosh) you must re-compile your source code to produce machine code specific to that platform. For example:



Since Java was designed to work on more than one platform, when you compile a Java program, you do not produce traditional machine code that is specific to one platform. Instead, the Java compiler converts your program into bytecode. This bytecode is like the machine code for the "Java Virtual Machine" (JVM). The JVM (also known as the Java Interpreter) is basically an abstract computer implemented in software. When you run the bytecode, it is the JVM's responsibility to convert it to the machine specific code required by each platform. For example:



The bytecode format is the same on all platforms because it runs in the same abstract machine -- the JVM. Each different platform will have its own version of the JVM that will translate the bytecode into the native machine code at runtime. Therefore this common bytecode can run on any platform that has a Java virtual machine. For example for Java applets, the virtual machine is either built into a Java-enabled browser or installed separately for the browser's use.

Unfortunately, because the JVM does its final conversion at runtime, Java programs are usually slower than native programs produced by languages such as C++. However, the big advantage, in many cases, is platform independence.

Because smaller devices such as cell phones and tablets are not as powerful as desktop computers, their JVM's have a different design. The Dalvik VM, Google Android's implementation of the JVM, is optimized for mobile devices such as cell phones and tablets. To help deal with the limited memory and processing power on mobile devices, the Dalvik VM runs `.dex` files which are more compact and efficient versions of the `.class` files.

1.4 The JDK and Java IDEs

To write programs in Java you normally need the JDK (Java Development Kit). The JDK includes a Java compiler, a JVM and all of the basic Java class libraries. The latest version of the JDK (version 8) can be downloaded for free from Oracle (www.oracle.com). Since your Java source code (`.java` file) is just a text file, you could use a text editor such as notepad to write your java programs. Then using commands provided in the JDK you could compile your source code to bytecode (`.class` file) and then pass this bytecode to the JVM to run your program.

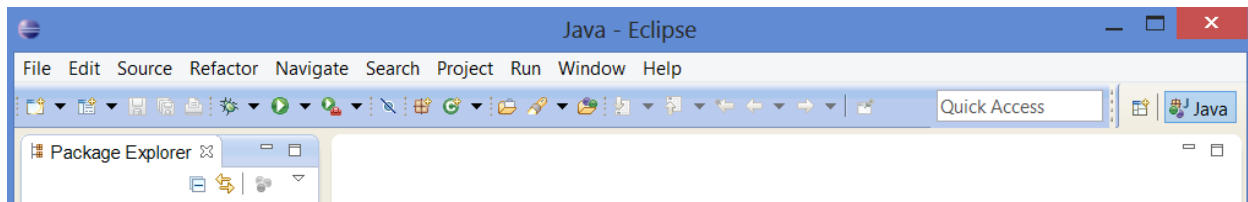
Although we can create, compile and run Java programs following the above steps, it is usually easier to use an IDE (Integrated Development Environment) that brings all of these steps together. There are many IDE's available for Java including Eclipse, Netbeans, Dr. Java, JCreator, BlueJ and Ready to Program with Java. In this course we will be using the Eclipse IDE.

Eclipse (available from www.eclipse.org) is a professional IDEs that can be used to edit, compile, run and debug your Java code. Eclipse has many features to make writing code easier including syntax highlighting, code completion, refactoring, nice error and warning messages, quick fix and even a spell check for your program comments. Even though it is a complete IDE that can be used to write programs in many programming languages including Java, Eclipse does not include the Java compiler and the Java class libraries so you will still need to download and install the Java JDK before installing Eclipse. Both the JDK and Eclipse are available for multiple platforms including Windows, Mac OS and Linux and in 32 bit and 64 bit versions.

Let's start by writing a simple Java program to output a message into a Console window. We will be using the Eclipse IDE to enter, edit and run our programs. To use Eclipse at home you will need to install both the Java JDK and the Eclipse IDE (see links on the course Moodle).

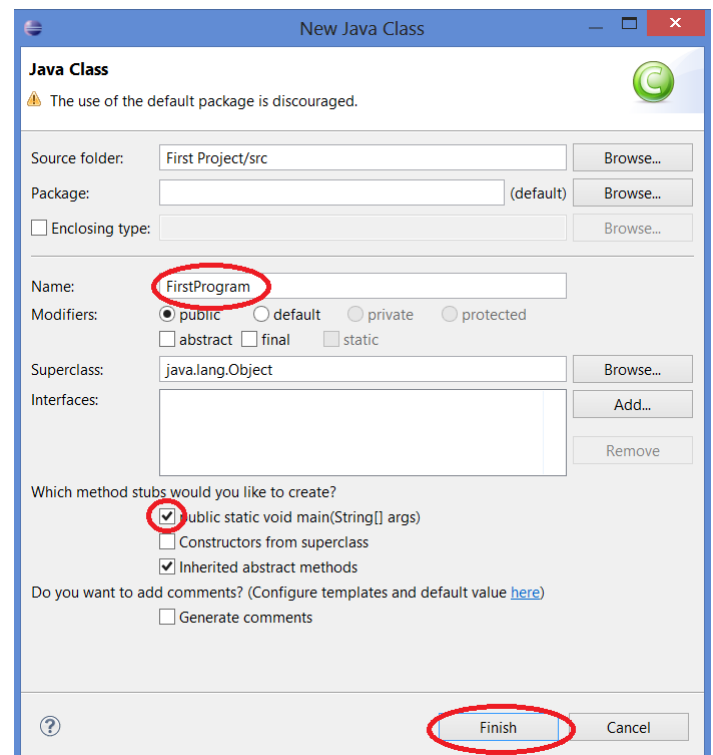
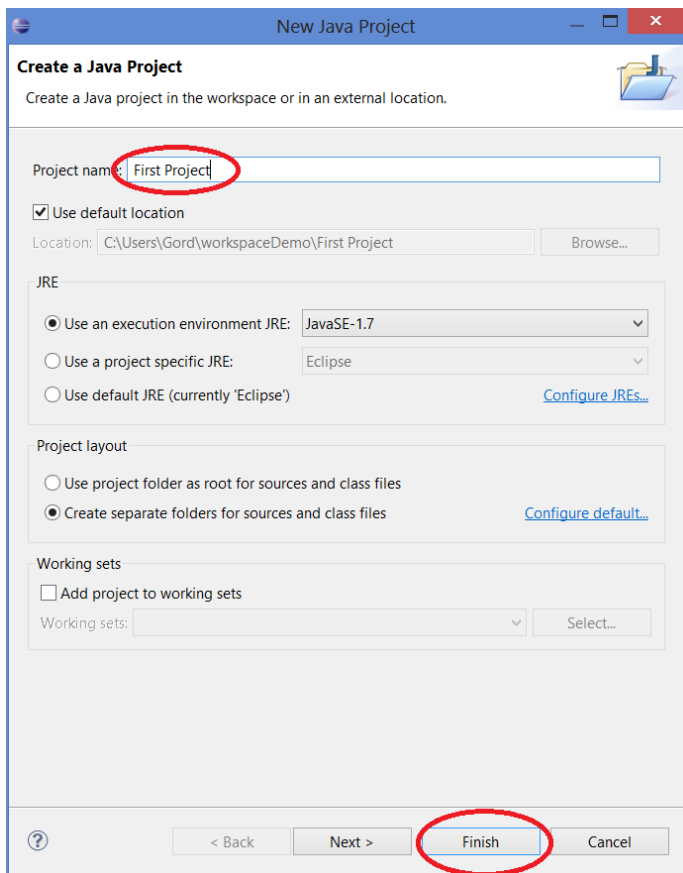
1.5 A Simple Java Program in Eclipse

Your first step is to load up the Eclipse IDE. If it is your first time loading Eclipse you will need to select a workspace location to store all of your projects. You can accept the default location or choose your own location. The first time you load up Eclipse you will get a Welcome screen. After closing the Welcome screen you will get the following top menu and tool bar.



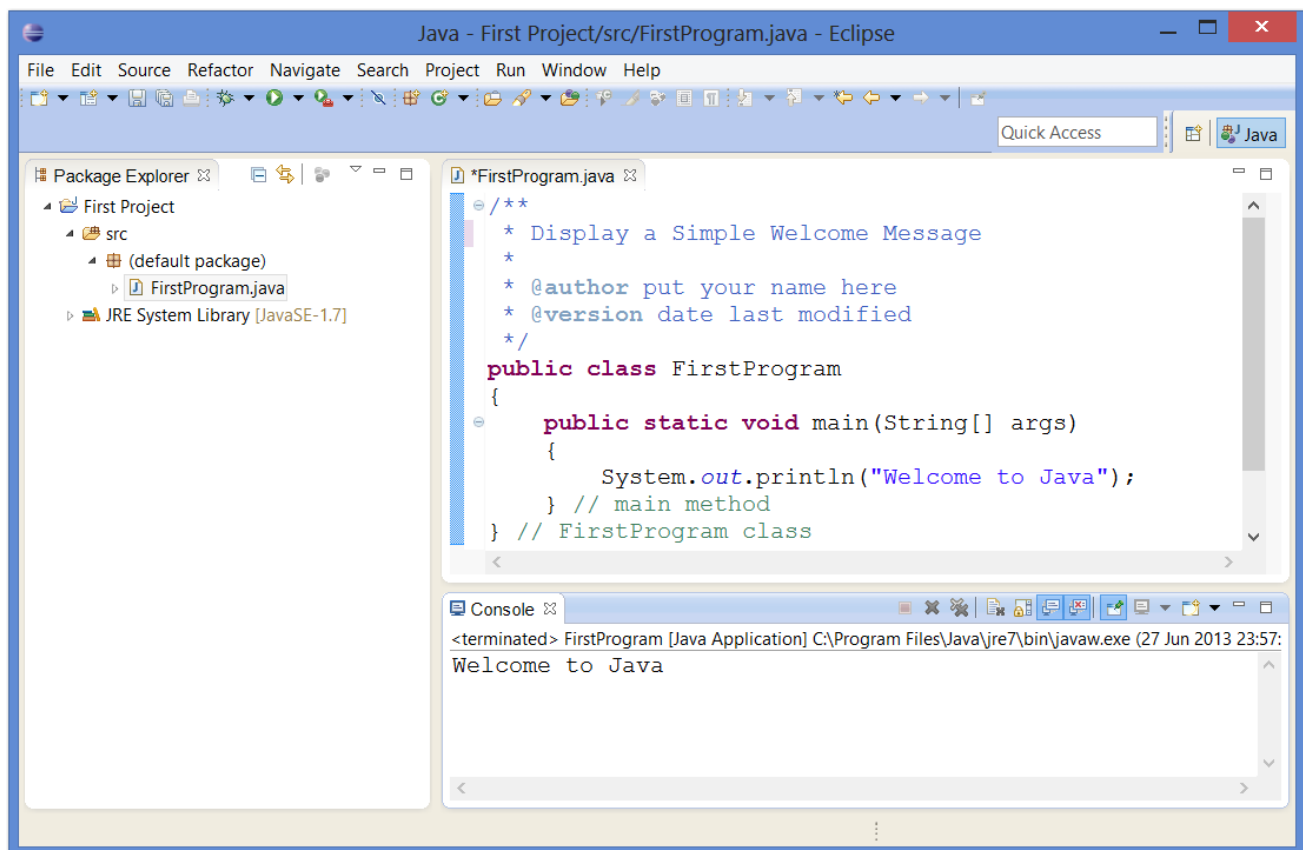
In Eclipse all code is part of a Project. To create a new Java Project, select **File->New->Java Project** to get the New Java Project dialog box shown below on the left. For most projects you can use the default settings, but you will need to enter a project name (e.g. `First Project`) and then select the **Finish** button. Once created, your new Project should appear in the Package Explorer (under the tool bar).

Your next step is to add a new class to the project. Right click on your new Project and select **New->Class** to get the New Java Class dialog box shown below on the right. Once again, you can accept most of the defaults including the default package (even though you are warned not to). You will need to enter a class name (use `FirstProgram`) and you will also need to make sure that `public static void main (String [] args)` is selected. When done, select the **Finish** button to create your new class.



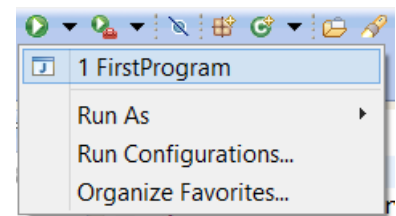
Your Eclipse IDE screen should now look like the screen below (with some of the code missing). Complete the program by typing in the rest of the code (including the comments) given below. Make sure that you change the `@author` tag to include your name and the `@version` tag to include the current date. To keep things simple you can remove all of the comments at the top of the `main` method including the `@param`. Once the program has been entered, you can run this program by right clicking on the Java program in the Package Explorer and selecting `Run As->Java Application`.

When completed, your program output should appear in the Console window at the bottom right of your screen (see below). If your program gives you errors make sure you check your code very carefully. In this example the Console window has been pinned to the program window (select the pin icon on the middle right of the Console window).



If you want to run your program a second time you can select the Run button in the tool bar (green circle with white arrow head) which will run the last program run. You can also choose `Run` from the `Run` menu or `Ctrl – F11`.

If you have more than one program that you are working on, you can select the drop down arrow on the Run button and then select the program that you want to run.



On the next few pages we will look at each part of this program in more detail.

Program Code:

```

/** Display a Simple Welcome Message
 *
 * @author put your name here
 * @version date last modified
 */

```

Introductory program comments tell anyone reading this code the purpose, author and date the code was last modified. `/**` and `*/` are used to start and end the comments.

```

public class FirstProgram
{

```

Heading for the `FirstProgram` class. A class is used to describe the framework for an object. In this case, the object is your program.

```

    public static void main (String [] args)
    {

```

Heading for main method

Displays a message in the System Console

```

        System.out.println ("Welcome to Java");
    }

```

```

    } // main method
}

```

```

} // FirstProgram class

```

start and end of the class code

start and end of main method

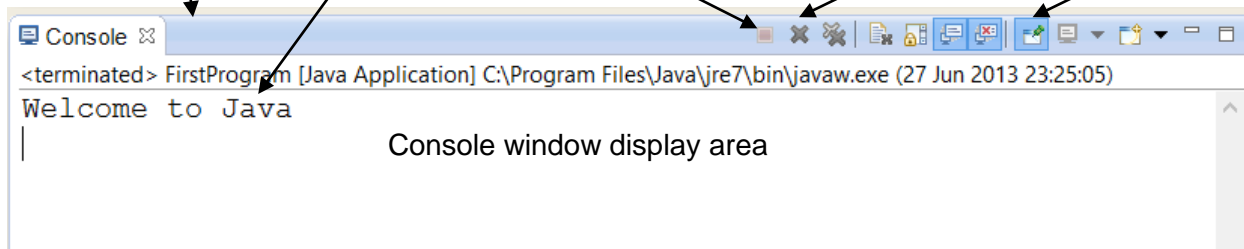
Program Output:

System Console window

Stop Application

Remove Launch

Pin Console



Console window display area

If you typed this code into Eclipse or if you are looking at this pdf file on the computer, you may have noticed that different sections of the code are shown in different colours. This feature is known as syntax colouring or highlighting. In most cases it makes your code easier to read and can be very helpful, especially when you are trying to find errors in your code. The given colours are fine but, if you want to, you can customize these colours in Eclipse.

On the next few pages a more detailed analysis of the above program is given. Please read it over very carefully.

Now let us look at the actual Java code section by section

```
/** Simple Java Program
 * Displays a Welcome Message
 * @author put your name
 * @version put date here
 */
```

The above statements are introductory comments. Introductory comments include the purpose of the program, the author, and the version (date). Every program you write should include these introductory comments. Comments should be used to make your program code easier to follow. See the Java Style Guide in Appendix A for more details on program comments.

```
public class FirstProgram
```

The above line of code is the heading for the `FirstProgram` class. In Java we use classes to describe the framework for objects. Since a program is an object every program needs at least one class. Since the `FirstProgram` class is the main class in our code we make it a public class. You can only have one public class in a source file, but you can have any number of non-public classes. The name of your public class should clearly describe your program. Also, the name of your program file must match the name of the public class. One final point, class names should always begin with a capital letter. All other letters in the name should be small letters, except for the first letter in each word in the name. (See also Appendix A)

In Java, we use curly braces { and } to indicate the start and end of a block (section) of code. This { indicates the beginning of the section of code which describes the `FirstProgram` class (our program). At the bottom of the code, the } indicates the end of the `FirstProgram` class description. In this case, this last } also indicates the end of our program.

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

This line is the heading for the main program or method. Every program class needs a main program. Later you will see that we can write smaller subprograms within our program, but we will always need a main program to tell the computer where to start. Our main program heading also includes the `public` modifier since we want it to be available outside of the class. For now, don't worry about the `static void` or the `String [] args`. Just remember, every main program you write will start with this same heading.

Just like the class description begins and ends with curly braces, the main program also starts with a { and ends with a }. Every line of code between the two curly braces is part of the main program. In this case our main program is quite simple with only the following one line of code:

```
System.out.println ("Welcome to Java");
```

The statement given on the previous page tells the computer to print the words "Welcome to Java" on the screen in the System Console window. `println()` (short for print line) is the method (action) we are going to perform in the `System.out` window. The message "Welcome to Java" is a **parameter** to the `println` method telling it what to print. Like the name suggests, `println` displays the given `String` parameter followed by a new line. You can also use `System.out.print()` to display data without a new line character at the end. Try the following lines of code to see the difference between `print` and `println`:

```
System.out.print("First line ");
System.out.println("Still on first line");
System.out.print("On a new line because of previous println");
```

You may have also noticed the semi-colon (;) at the end of each of these statements. In Java (like C and C++) the semi-colon is used to indicate the end of a Java statement. Don't worry you will quickly get used to putting a semi-colon at the end of each statement. However, you must be careful because, not every line in your Java program requires a semi-colon. Note: class and method headings do not end with a semi-colon.

```
    } // main method
} // FirstProgram class
```

As mentioned earlier, the last two lines close off the main method and the `FirstProgram` class definition. Single line comments (beginning with `//`) have been added to make it clear which block of code these `}`'s end. Notice how the `}`'s are indented to match up with the matching `{`. By matching the curly brace pairs it is a lot easier to follow your program code. Luckily the Eclipse IDE will automatically indent and line up your code when you press `Ctrl-Shift-F`. If, after pressing `Ctrl-Shift-F`, your code is not lined up, your code probably has an error. In order to consistently format your programs to a similar standard, a custom formatting file (called `RHHS.xml`) can be easily imported into your Eclipse workspace. Later in the course, you can also customize the formatting options to your personal preferences.

After looking over this code, you may be thinking this Java language is a little too complicated. If you have studied a language such as Turing, you may recall the same program in Turing would be:

```
put "Welcome to Turing"
```

Turing is designed as a teaching language so generally it is a lot easier to work with. However Java is a professional language so even though it may be a little more complicated at first, it has more potential. Also, as you will see, most of the above code is pretty standard so we can cut and paste or use templates when creating our programs.

Although this was a very simple program, it contains a lot of important ideas about the Java language. Since we will be building on these ideas, it is important that you understand all of the material that was presented before moving on. If there is still something you don't understand, you should ask questions or seek extra help.

1.6 Problem Solving with Computers

As we start to write computer programs to solve specific problems, it is important to consider the following steps to make sure that you come up with the best solution. In many cases there will be overlap and backtracking between each of the following steps. As you work towards your final solution, it is important that you find and correct any errors or problems at the earliest possible stage.

1) Define the Problem

- read over the problem carefully and make sure that you understand what the problem is. Ask questions if the problem is not clearly understood.
- this is a very important first step since you can't possibly find the correct solution unless you have identified the correct problem.

2) Analyze the Problem (Initial Design)

In analyzing the problem you should identify the following:

- outputs (user requirements) and inputs (what information you are given)
- memory locations that you may need to store the information required to solve the problem.
- main steps required to solve this problem. For larger problems you may want to break the problem down into smaller sub-problems (see Chapter 4).

3) Design the Solution (More detailed design)

- create a comprehensive set of test cases (specific examples) and think about how to solve these example problems without a computer
- plan out the required steps (algorithm) to solve the problem
- determine the tasks to be performed and their sequence (input, calculations, decisions, output, loops)
- consider extreme cases to make sure your algorithm works for all possible cases
- use flowcharts or pseudo code to help clarify your algorithm (see Chapter 3).
- carefully check that the design produces correct results in an efficient way

4) Implement the Solution

- implement (code) the solution using the most appropriate tool (e.g. Java)
- if possible, use built in tools (e.g. objects and methods) to simplify your solution.
- find and correct any errors including compiler/syntax and logic errors
- code should adhere to a defined programming style and should include internal documentation (comments) (see Appendix A).

5) Final Testing and Debugging

- find and correct logic errors and run-time errors
- use the comprehensive set of test data created earlier to check that the final program is a robust and complete solution to the original problem
- finding and correcting errors at this stage can be more difficult so it is very important that you test your design and correct any errors as soon as possible.

1.7 Chapter Summary

Keywords/phrases introduced in this Chapter. You should be able to explain each of the following terms:

assembler	indenting (paragraphing)	object code
assembly code	IDE	OOL
binary	JDK	OOP
bytecode	JVM	parameter
class	machine code	platform
compile	main	program comments
compiler	method	software development life cycle
inheritance	mnemonic code	source code

Java key words/syntax introduced in this chapter:

{ and }	class	static
;	public	void

1.8 Questions, Exercises and Problems

- 1) Explain the importance of both hardware and software in a computer system.
- 2) Give an advantage of using assembly language to write programs.
- 3) Give two disadvantages of using assembly language to write programs.
- 4) Give and explain two similarities and one difference between an assembler and a compiler.
- 5) What is JVM short for?
- 6) What is bytecode and how is bytecode different than machine code?
- 7) Give and explain one advantage and one disadvantage of the way Java code is translated into machine code compared to other languages (e.g. C++).
- 8) What is IDE short for? What does an IDE do?
- 9) One feature of the Eclipse IDEs is code completion. What is code completion?
- 10) Why should we include comments in our Java programs? What are the minimum comments you should include in your programs? See also Appendix A.2.
- 11) Give and explain at least two advantages of syntax highlighting in the Eclipse IDE.
- 12) What is a parameter? Why do we use parameters with methods?

Chapter 2 – Variables, Input/Output and Calculations

One of the important functions of a computer is its ability to store data in memory. In Java, memory locations whose contents can be changed are called **variables**. In the following chapter, we will find out how to declare primitive and String variables. We will also investigate how to input and output these variables as well as how to manipulate their contents with arithmetic expressions.

2.1 Declaring Variables and Primitive Types

In Java, like Turing, variables must be declared before they can be used. To declare a variable, we must give it a name and indicate what type of information it will hold.

Even though Java is an object-oriented language, not every variable in Java refers to an object. For the basic types of variables, Java has a collection of "primitive" types. When you declare a variable using one of these types, the appropriate amount of memory is automatically allocated without using the `new` command.

Although there are 8 primitive types available in Java, we will only be considering 5 different types summarized in the table below:

Summary of Java Primitive Data Types

Type	Size	Range	Examples/ Other Info
boolean		false, true	true or false
char	16 bits	Unicode 0 to 65,535 ($2^{16} - 1$)	Unicode Characters 'A', '5', '%', '?', 'c', '\u0121'
int	32 bits	$-2,147,483,648$ (-2^{31}) to $2,147,483,647$ ($2^{31} - 1$)	most integers stored as signed numbers using two's complement
long	64 bits	(-2^{63}) to $(2^{63} - 1)$	larger integers stored as signed numbers using two's complement
double	64 bits	Big numbers: -10^{308} to $+10^{308}$ Fractions: -10^{-324} to $+10^{-324}$	numbers with decimals (any real number) 15 digit precision

The above primitive types behave just like variables in Turing. The following shows some examples of how you would declare each type in Java. Notice, unlike Turing, you don't use the keyword `var`. You may also notice that, in Eclipse, the names of primitive types appear in bold.

Declaring variables (give type and name):

```
boolean gameOver;  
char answer;  
int noOfStudents;  
long worldPopulation;  
double totalPrice, salesTax;
```

In some languages (C and Pascal) you need to declare your variables at the top of your program. In Java, you can declare your variables anywhere in your code as long as you declare them before they are used. When we are working with loops and selection structures you will also need to make sure that you declare your variables in such a way that you are not limiting their scope (more on this later). In most cases we will declare our variables on the same line that they are first initialized. For example:

```
int total = 0;  
boolean gameOver = false;  
int noOfStudents = keyboard.nextInt();
```

Which type of variable you need will depend upon what you are storing in memory. In the above examples we used a `boolean` variable to keep track of whether the game was over. This was appropriate since the game is either over or not (true or false). We used `char` for the `answer` variable since it represents the answer to a multiple-choice question, which is just a single character. We used an integer to keep track of the number of students since you can't have a fraction of a student. Because integers take up less space and are easier to calculate with, if possible, you should always use integers when storing a number. However, to keep track of money amounts such as `totalPrice` and `salesTax` we used a `double` since these amounts could have decimal values that cannot be stored in an integer variable.

Sometimes information in a memory location has a constant value. In Turing, we could declare constants using the keyword `const`. In Java, if a variable has a constant value, you can use the keyword `final` when declaring this variable. For example:

```
final double HST_RATE = 0.13;
```

A `final` behaves just like a variable except you can't change it while the program is running. Using constants makes your code easier to follow. For example the statement:

```
priceWithTax = price + price * HST_RATE    is clearer than:  
priceWithTax = price + price * 0.13
```

Also, if we need to change all occurrences of a constant value we only need to change one value.

You may have noticed the pattern we used when naming variables and constants in the above examples. See Appendix A.1 (Naming Identifiers) for more information on naming variables and constants (final variables) in Java.

2.2 Declaring Strings in Java

In Turing, if you wanted to store a person name in your program you would declare a string variable (memory location that can hold one or more characters). However, in Java, there is no equivalent "primitive" string type. Instead, we need to work with `String` objects. As you will see, working with objects is different than working with primitive types. For example, when we declare a `String` variable we are really creating a reference to a `String` object. This reference will keep track of the memory location of the `String` object it refers to. For example, the following code declares a variable called `name` that is a reference to a `String` object and then uses the **new** command to create the `String` object that keeps track of the actual name.

```
String name = new String("Bart Simpson");
```

`String` objects have their own set of methods and properties that we will be looking at in Chapter 5. Since `String` objects are a very common object type, we are allowed to initialize Strings using a form similar to that used to initialize primitive types. For example the following statement is allowed in Java:

```
String message = "Welcome to Java";
```

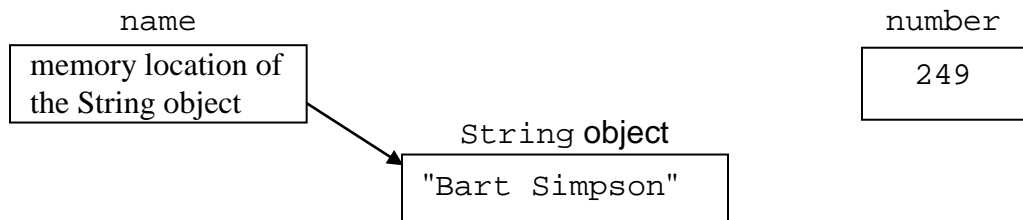
This will declare a reference to a `String` object as well as create and initialize this object without using the **new** command. This method of initializing a `String` object is more efficient than using the **new** command but it is only allowed for `Strings`. When you type the above code into Eclipse, you may notice that the name `String` is in black but not bold. This is because `String` is an object and not a primitive type. There are some subtle differences between objects and primitives.

More Details on the Differences between Objects and Primitives

To help clear up the differences between a variable that refers to an object and a primitive variable, let's look at some memory diagrams. For example, given the following code:

```
String name = "Bart Simpson";  
int number = 249;
```

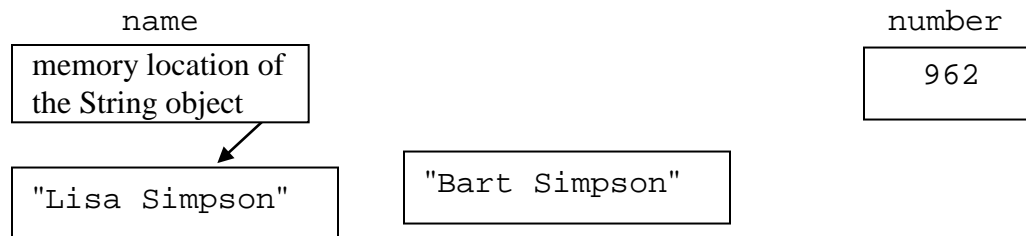
The following diagram represents what is happening in memory:



As you can see a `String` variable just keeps track of the `String` object's location in memory – this is why a `String` variable is sometimes called a `String` reference. On the other hand, the `int` (primitive) variable keeps track of the actual value of the number.

Like a primitive variable, the `String` reference is a fixed size even if the `String` object it refers to is not. In Java, `Strings` are immutable so if we change the value of a `String` variable, we are really just changing the reference to refer to a different `String` object. Here is an example showing what happens in memory when we change the value of both a `String` and an `int`.

```
name = "Lisa Simpson";  
number = 962;
```



As you can see, the `String` containing “Bart Simpson” stays the same, but the `String` variable (reference) now refers to the new `String` object containing “Lisa Simpson”. Because nothing is currently referring to the “Bart Simpson” `String` object, this section of memory will get reclaimed in a process called “Garbage Collection”.

One big advantage of storing `Strings` this way is that the amount of memory allocated for a `String` object matches the actual string you are storing. So if you want to store the name “Mr. T”, you will only need 10 bytes (5 characters). Later, if you want to store the name “Nikki Nikki Tembo No So Rembo Oo Ma Moochi Gamma Gamma Goochi”, the new `String` object will use 122 bytes (61 characters). In both cases, you can use the same `String` variable: `name` to refer to the 2 different `String` objects. In Java, the size of a string is limited by the amount of available memory. In Turing the default and maximum size of a string was 255 characters.

While it is important to know and understand the differences between objects and primitives, in most cases they will behave in a similar way. Later in the course, when we are looking at memory traces of our code, we will treat `Strings` like primitives in our memory diagrams.

Identifying the types of objects (including primitives) that you will need in your Java programs is an important part of writing computer programs.

2.3 Using the `Scanner` Class to Input Information

Now that we know how to create memory spaces to store information, we need to know how to input data from the user into these memory locations. Create a new project in Eclipse and type in the following Java program. You can also cut and paste this code from the PDF file.

```
// Import the Scanner class
import java.util.Scanner;

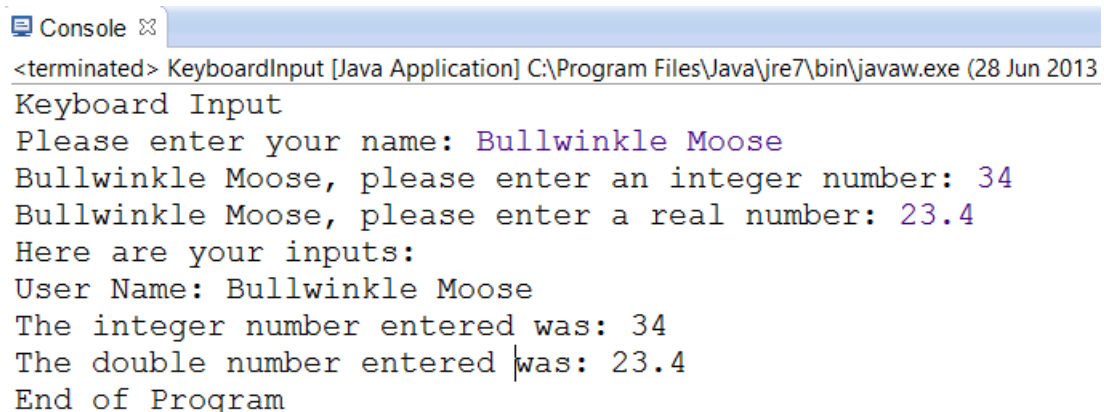
public class KeyboardInput
{
    public static void main (String [] args)
    {
        // Create a new Scanner object for Keyboard input
        Scanner keyboard = new Scanner (System.in);

        // Input the information from the keyboard
        // We also declare each new variable as needed
        System.out.println ("Keyboard Input");
        System.out.print ("Please enter your name: ");
        String name = keyboard.nextLine ();
        System.out.print (name + ", please enter an integer number: ");
        int myInteger = keyboard.nextInt ();
        System.out.print (name + ", please enter a real number: ");
        double myDouble = keyboard.nextDouble ();
        keyboard.close();

        // Output the inputted information
        System.out.println ("Here are your inputs:");
        System.out.println ("User Name: " + name);
        System.out.println ("The integer number entered was: " + myInteger);
        System.out.println ("The double number entered was: " + myDouble);
        System.out.println ("End of Program");

    } // main method
} // KeyboardInput class
```

Now try running this program by pressing the Run As -> Java Application (or Ctrl-F11). If the program reports any error message, check your code carefully and fix any typing mistakes. You should get the following sample output:



```
<terminated> KeyboardInput [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (28 Jun 2013)
Keyboard Input
Please enter your name: Bullwinkle Moose
Bullwinkle Moose, please enter an integer number: 34
Bullwinkle Moose, please enter a real number: 23.4
Here are your inputs:
User Name: Bullwinkle Moose
The integer number entered was: 34
The double number entered was: 23.4
End of Program
```

Now let's look at some of the new statements added since Chapter 2.

```
import java.util.Scanner;
```

The above statement is included because we are using the `Scanner` class in our program. This statement tells the compiler to look for the `Scanner` class code in the `java.util` package. We need to use the `import` statement to tell the compiler where to find the code for any classes we are using in our program.

Since common classes in the `java.lang` package such as the `String`, `Math` and `Character` classes are used in almost every program; the `java.lang` package does not require an `import` statement. However, for all other classes not in the `java.lang` package, you will need to include the appropriate `import` statement. You won't need to worry about other import statements until Chapter 8.

In Eclipse if you use a new class that requires a package to be imported, you can use the "quick fix" auto correct feature. Without the proper import you will get an error. If you put the mouse pointer over the error, you will get a suggested correction. In most cases (but not all), the suggested `import` will be correct.

Instead of using an import statement you can write the full name of a class in your code. For example, without an import statement, instead of using `Scanner` in our code, we would use the full name `java.util.Scanner` instead. Using imports is the preferred method since it makes your code cleaner and easier to follow.

Unlike the "include" statement in Turing, the import statement does not add in any code, it just helps the compiler find classes. Therefore, unnecessary import statements may slow down the compiling process but they will not affect the size of your final bytecode.

The next thing you may have noticed is that we declared the variables `name`, `myInteger` and `myDouble` on the same line that they were first used. We could have also declared all of the variables together in a section of code at the top of the program. Although not necessary in Java, putting variable declarations together at the top of the program helps some students to organize their code.

```
System.out.print ("Please enter your name: ");  
String name = keyboard.nextLine ();
```

To read in a string variable we must first ask for the information. The first line above is called a "prompt" for input. For the prompt we used `print()` instead of a `println()` so that the name is entered beside the prompt (see the sample Console output on page 16). Change the `print()` to a `println()` and you will notice the difference. On the next line of code we declare a `String` variable called `name` and then read in this name using the `nextLine()` method. This method has no parameters and it returns the value read, which is then assigned to the `name` variable.

Since we are reading in from the Scanner object called "keyboard" we use `keyboard.nextLine()`. The `nextLine()` method reads in the whole line so this code will read in both the person's first and last names.

```
System.out.print (name + ", please enter an integer number: ");
int myInteger = keyboard.nextInt ();
System.out.print (name + ", please enter a real number: ");
double myDouble = keyboard.nextDouble ();
```

To read in an integer we use `nextInt()` and to read in a double we use `nextDouble()`. Like `nextLine()`, `nextInt()` and `nextDouble()` return a value which is then assigned to the newly declared `myInteger` and `myDouble` variables respectively. You may have also noticed that in the `print()` method, the parameter is: `name + ", please enter an integer number: "`. Normally a `+` sign is used to add two numbers, however, if you put a `+` sign between two String objects it "adds" (concatenates) them together. So if `name` was "Marcus", the prompt appears as:

Marcus, please enter an integer number:

```
System.out.println ("Here are your inputs:");
System.out.println ("User Name: " + name);
System.out.println ("The integer number entered was: " + myInteger);
System.out.println ("The double number entered was: " + myDouble);
System.out.println ("End of Program");
```

These statements will output the information that was input. We can also add some formatting to our output using `printf` (see the next section 2.4).

Since `nextLine` reads the complete line including the new line (Enter) character and the other next commands (`nextInt`, `next`, `nextDouble`, etc) only read the next token (skipping over any whitespace), you have to be careful when mixing `nextLine` with the other methods. For example, given the sample code and input shown below:

<code>String firstLine = keyboard.nextLine();</code>	<u>Input</u>
<code>int firstInt = keyboard.nextInt();</code>	First Line
<code>int secondInt = keyboard.nextInt();</code>	34 47
<code>int thirdInt = keyboard.nextInt();</code>	68
<code>String secondLine = keyboard.nextLine();</code>	Second Line

In this example, the first `nextLine` will read in "First Line" including the new line character at the end of this line. The first `nextInt` will then read in 34. The second `nextInt` will skip over the space between the 34 and the 47 and read in the 47. The final `nextInt` will skip over the new line character at the end of the "34 47" line and then read in the 68. It will not read in the new line character after the 68. Because we then have a `nextLine` after the `nextInt` it will read in the new line character after the 68, assigning an empty String to the `secondLine` variable. As mentioned this problem occurs when we are mixing `nextLine`'s with `nextInt`'s. To avoid this problem we could put an extra `nextLine` before the last line of code to read in the new line character after the 68 so that the last `nextLine` will read in "Second Line" as expected.

2.4 Formatted Output using `System.out.printf()`

For most simple output, `System.out.println()` and `System.out.print()` are all that we need. However, in some cases we want more structured output so for formatted output we can then use the `System.out.printf()` method. The basic structure of this method includes a format string and a list of variables. For example:

```
System.out.printf("format String", list of variables to print)
```

The format string can contain regular text that will be displayed as is (including any spaces) as well as place holders (indicated with a `%` for each placeholder) that correspond to each variable. Here are some examples:

```
int number = 345;
System.out.printf("The answer is: %d", number);
```

will output: The answer is: 345

In this case the `%d` (`d` is for decimal number) will be replaced with the value in that corresponding variable `number` (345 in this example).

We can also add other modifiers to the placeholder to specify the field size. For example:

```
System.out.printf(" *%5d*", number);
```

will output: * 345* (with 2 extra spaces in front of the number – field size of 5 in total)

When the field size is bigger than the value to be displayed, by default the value will be right justified (extra spaces are added to the left), but we can also left justify the result by adding a `-` between the `%` and the field size. For example:

```
System.out.printf(" *%-5d*", number);
```

will output: *345 * (with 2 extra spaces after the number – field size of 5 in total)

We can also add in extra format flags such as `'0'` or `'.'` to enhance your output. For example:

```
System.out.printf("%05d", number); // pad with leading zeros
```

will output: 00345 and

```
number = 1234567;
System.out.printf("%,d", number); // separate number groups
```

will output: 1,234,567

For doubles (we use `f` for floating point numbers) we can also indicate the number of decimal places (with automatic rounding) by adding in a decimal point followed by the number of decimal places. For example:

```
double number = 123456.789;
System.out.printf("Number: %, .2f", number);
```

will output: Number: 123,456.79

Here is a list of some of the formatting codes you may use. For more formatting codes, see the `printf` method description in the Java API. Note: Since each format code begins with a `%`, if you actually want to display a `%`, you need to put two percent signs (i.e. `%%`) in your format string.

<code>%b</code> boolean	<code>%s</code> String
<code>%c</code> Unicode character	<code>%d</code> decimal integer (<code>int</code>)
<code>%f</code> floating point number (<code>double</code>)	<code>%n</code> new line (similar to <code>\n</code>)

Here are the format flags you can use:

- left align	0 show leading 0's
+ show + for positives	, show decimal separator

Here is an example with doubles and Strings – the `%n` (similar to `\n`) at the end gives us a new line character.

```
String item = "Almonds";
double pricePerKg = 12.34;
double weight = 3.4566;
double cost = weight * pricePerKg;
System.out.printf ("%%-10s: %.3f kg @ $%.2f per kg will cost $%8.2f%n",
                    item, weight, pricePerKg, cost);
```

Will output:

```
Almonds    : 3.457 kg @ $12.34 per kg will cost $    42.65
```

To format data that is being put into a String we can use the `String.format()` method which uses the same parameter structure as `printf`. For example, the following code:

```
int hours = 5;
int minutes = 9;
int seconds = 5;
String timeStr = String.format("%d:%02d:%02d", hours, minutes, seconds);
```

Will set `timeStr` to "5:09:05"

2.5 Arithmetic Expressions in Java

Arithmetic expressions are used to manipulate `int`, `double`, `long` and `char` variables in Java. Java has the basic operators for adding, subtracting, multiplying and dividing plus a few extras. The main operators we will be using are shown below. Each will be explained using examples:

Java Operators:	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	Add, subtract, multiply and divide
	<code>%</code>				modulo (finds the remainder)
	<code>++</code>	<code>--</code>			Increment and decrement
	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	Shorthand assignments - See examples below

In arithmetic expressions, the normal order of operations (BEDMAS) is followed. Therefore you can use parentheses (brackets) to get the results you want.

e.g. `4 + 3 * 7 - 9` will multiply, add, and then subtract
`(4 + 3) * (7 - 9)` will add and subtract, then multiply

To make your code more readable, you can always add in extra parentheses.

When you are mixing types (e.g. integers and doubles) in an arithmetic expression, you must be careful. If your expression contains only integer values the result will also be integer. If your expression contains any double values, the result will be double.

With this in mind, if you use `/` to divide 2 integers, the result will be an integer. Any fractional part is truncated (chopped off). This can produce some unexpected results. For example:

`6 / 4` will be 1 since 6 and 4 are both integers while
`6.0 / 4` will be 1.5 since 6.0 is a double

The modulo operator (`%`) produces the remainder on integer division (like `mod` in Turing). For example:

`27 % 4` will be 3 (since 4 goes into 27 6 times with remainder 3) and
`14 % 2` will be 0 (this is an easy way to check if an integer is even)

When assigning arithmetic expressions to a variable we use an `"=`" (Both Turing and Pascal use a `":="`). With an assignment statement, the right-hand side of the equal sign is evaluated and then assigned to the left-hand side of the equal sign. For example:

```
volume = width * length * height;
```

will multiply the contents of the variables `width`, `length` and `height` and then store the result in the variable called `volume`.

When you are using an assignment in Java, you must make sure that the left-hand side of the equal sign is type compatible with the right-hand side of the equal sign. Therefore, you cannot assign a double value into an integer variable. For example, the following line of code will give you a compilation error because the right-hand expression will produce a double value and `number` is an `integer`.

```
int number = 12 * 3.2;           // Not allowed in Java
```

Section 2.6, gives more details on converting between different types.

Java also has some shorthand assignment operators that you can use if you wish. If you want to use these shorthand's, make sure you understand them. In each of these examples, the final result is stored in the variable `x`.

Shorthand Expression	Code Equivalent	Meaning (result is stored in <code>x</code>)
<code>x += y</code>	<code>x = x + y</code>	Add <code>y</code> to <code>x</code>
<code>x -= y</code>	<code>x = x - y</code>	Subtract <code>y</code> from <code>x</code>
<code>x *= y</code>	<code>x = x * y</code>	Multiply <code>x</code> by <code>y</code>
<code>x /= y</code>	<code>x = x / y</code>	Divide <code>x</code> by <code>y</code>

Since adding or subtracting 1 from a variable is a common operation, Java also has special increment and decrement operators. For example:

```
++count or count++    - adds 1 to the variable count  

--pos or pos--      - subtracts 1 from the variable pos
```

`count++` is similar to `count = count + 1` or `count +=1`. If you are only adding 1 to `count`, `count++` is preferred.

As you see, we have two forms of `++` and `--`. We have the prefix operator (e.g. `++count`) and the postfix operator (e.g. `count++`). If you are just adding or subtracting 1 in your Java statement, there is no difference between using the prefix or postfix operator. For example, the following sections of code do the exact same thing -- add 1 to the variable `counter`.

Using the prefix operator

```
int counter = 0;  
++counter;
```

Using the postfix operator

```
int counter = 0;  
counter++;
```

Even though the above sections of code are changing the value of `counter`, no assignment (`=`) is required.

Since ++ and -- are used to change the value of a variable, if you combine these operators with an =, things can get confusing since you will be changing two variables in one statement. For example, the following sections of code do not produce the same result.

Using the prefix operator

Adds then assigns

```
firstNumber = 10;
secondNumber = ++firstNumber;
```

Using the postfix operator

Assigns then adds

```
firstNumber = 10;
secondNumber = firstNumber++;
```

When using the prefix operator, the second line of code will add 1 to `firstNumber` and then assign the new result to `secondNumber`, setting both variables to 11. On the other hand, using the postfix operator, the second line of code will assign `firstNumber` to `secondNumber` and then add 1 to `firstNumber`, setting `secondNumber` to 10 and `firstNumber` to 11. This may seem logical to some but it could be confusing to others, so to avoid writing confusing code, you should limit your use of ++ or -- in assignment statements.

The following table shows some more examples of arithmetic expressions:

Expression	Result	Type of Result
<code>(3+7)*(4-9)</code>	-50	int
<code>3.0+7*4-9</code>	22	double
<code>56 / 15</code>	3	int
<code>56 % 15</code>	11	int
<code>3 / 4 + 4 / 5</code>	0	int
<code>3.0 / 4 + 4 / 5.0</code>	1.55	double
<code>(2 * 15 + 7) % 5</code>	2	int
<code>2.3 * 7 - 21.4</code>	-5.3	double

Here are more examples of some of the special operators. After each statement, the value of the changing memory location `number` is shown on the right.

	<u>number</u>
<code>int number = 2;</code>	2
<code>number = number + 3;</code>	5
<code>number = number * 2;</code>	10
<code>number ++;</code>	11
<code>number += 7;</code>	18
<code>number /= 2;</code>	9
<code>number --;</code>	8

2.6 Type Conversion (Casting and Promotion)

Sometimes you want to convert between two different types of variables. For example you may want to store an integer variable into a double variable or vice versa. Since Java is a strongly typed language, only certain conversions are allowed. Generally if you can convert from one type of variable to another type without losing any information, this is allowed. The following examples show which conversions are allowed in Java. None of the allowed conversions include booleans or Strings.

Given the following variable declarations:

```
char charVar;  
int intVar;  
double doubleVar;
```

The following conversions are allowed, since no information could be lost. For example, a character variable is stored using 16 bits, so when you put the same 16 bits into an integer variable which can hold 32 bits, all of the bits can be stored. In each of these cases the variable on the right-hand side of the equal sign can be **promoted** to the type of the variable on the left-hand side of the equal sign.

```
intVar = charVar;  
doubleVar = charVar;  
doubleVar = intVar;
```

The following conversions are not allowed, since information could be lost. For example, since an integer variable is stored using 32 bits, if you try to put these 32 bits of data into the character variable which holds only 16 bits, some of the bits may be lost.

```
charVar = intVar;  
charVar = doubleVar;  
intVar = doubleVar;
```

If you want to store a double value in an integer, all is not lost. You can "cast" the double to an integer, the same way you would cast an actor to play a role other than themselves. A cast is a way of converting between types when information could be lost. In the case of converting a double to an integer, the fractional part of the number will be lost. For example the following will cast `doubleVar` temporarily to an integer and then store the result in `intVar`:

```
doubleVar = 3.65;  
intVar = (int) doubleVar;
```

When the `doubleVar` is cast to `int` the result is truncated to 3.

When you cast a value to another type, you are telling the compiler that if any information is lost during casting that you are ok with that. For example, in the following code, we are adding a `double` to a `char` and then casting the `double` result back to a `char`. In this case the information we are losing is not needed.

```
char randomLetter = (char) ('A' + Math.random()*26);
```

Generally if you ever get an error saying the "type of the left-hand side in this assignment is not compatible with type on the right-hand side expression" you may be able to fix it with a cast. However, you must be very careful since when you use a cast you could be losing valuable information.

Finally, sometimes you want to convert a `char`, `int` or `double` to a `String`. If you are concatenating a `char`, `int` or `double` with a `String` they will automatically be converted. For example, in the expression: `"The number is " + 3.45` `3.45` will automatically be converted to a `String` before being concatenated to `"The number is "` to produce the `String`:

```
"The number is 3.45"
```

We can use this feature when creating simple output strings for the `println()` instead of using the `printf` method. We can also use `String.format` to convert number data into strings. More information on converting to and from strings is presented in section 5.7.

2.7 Math and Character Class Methods and Constants

To manipulate both numbers and characters, you can use methods in the `Math` class and the `Character` class (see Appendix D for more details). Note: Since these are all `static` methods, you must precede each method name with `Math` or `Character`. For example, to use `sqrt()` you would use `Math.sqrt()`. Below are some examples of a few of the more useful `Math` methods.

When displaying a number using the `printf` method we can round the number to a given number of decimal places. However, if we want to round a number in memory, we can use the `Math.round` method which rounds a decimal number to the nearest integer. Since the resulting integer can be quite large, `Math.round` returns a `long` type. Therefore, if want to store the result in an `int`, we need to cast the result to `int` first. For example to round `doubleVar` to the nearest `int` we would use:

```
int intVar = (int) Math.round(doubleVar);
```

To round a `double` number to a certain number of decimal places, we can use a simple trick. For example, if we want to round `345.678` to 1 decimal place, first we multiple the number by `10` to get `3456.78`. Then we use `Math.round` to round the number to the nearest whole number getting `3457`. After rounding, we undo the multiplication by dividing by `10.0` to get the correct result `345.7`. Putting this altogether, the code to round the double variable `number` to one decimal would be:

```
double roundedNumber = Math.round(number * 10) / 10.0;
```

Notice we divide by `10.0` to avoid integer division since `Math.round` returns a `long`. How would you change this code if you wanted to round a number to 3 decimal places?

You can use a similar trick to round an integer number to the nearest 10, 100 or 1000 etc. See Question 16 at the end of the chapter.

In addition to the `Math` class methods there are also a few `Math` class constants including `Math.PI` for the value of π . Given the radius of a sphere is stored in a variable called `radius`, the following code can be used to find the volume of the sphere:

```
volumeOfSphere = 4.0/3 * Math.PI * Math.pow(radius,3);
```

Why do you think we needed to use `4.0/3` instead of `4/3` in the above code?

For a lot of games and simulations we want to generate random numbers to simulate random events. The method `Math.random` generates a random double number between 0 (inclusive) and 1 (exclusive). To convert this random double to a random integer, we need to multiply the result by how many random integers we want, cast this result to an `int` since we want an `int` and then add in the value of the lowest value we want. Basically we are stretching the double number from 0 to 1 to a larger range, casting this result to an integer and then shifting this value to set the start of the range. For example, to generate a random integer number between 1 and 10 inclusive (including both 1 and 10) we use the following code:

```
int random1To10 = (int) (Math.random()*10) + 1;
```

In this case `Math.random()` generates a double value between 0.0 and 0.999.... Multiplying by 10 stretches this out to give a double value between 0.0 and 9.999.... Then when we cast this result to `int` we get an integer number between 0 and 9 inclusive since the cast will truncate the decimal portion. This will change all of the values between 0.0 and 0.999 to 0 and all of the values between 1.0 and 1.999 to 1 etc. with numbers in the final range of 9.0 to 9.999 being changed to 9. Finally, adding 1 shifts the result to give a final value between 1 and 10. Since the lower value could be a negative number, it is important that we cast to `int` before adding the lower value.

In general, to generate a random integer number we can use the following template:

```
int randomNo = (int) (Math.random()*noOfNumbers) + firstNumber;
```

The `Math.max` and `Math.min` method can be used to compare the values of 2 numbers returning the one that is larger or smaller. To find the largest or smallest value of 3 numbers we will need to write our own code using decision structures (`if` statements) that will be introduced in the next chapter. In the meantime, there is a simple although not very efficient, work around. Given 3 numbers stored in the variables `first`, `second` and `third`, the largest value can be found with the following code:

```
int largest = Math.max(Math.max(first, second), third);
```

This code first finds the largest number between `first` and `second` and then it compares this result to `third` to find the largest of all 3 numbers.

As was mentioned, although the above code is interesting, writing your own `if` statement to find the largest value of 3 numbers will be a lot more efficient.

2.8 Chapter Summary

Keywords/phrases introduced in this Chapter. You should be able to explain each of the following terms:

arithmetic expressions	import	promotion
assignment compatible	increment operator	remainder
assignment statement	integer division	strings
order of operations	modulo operator	truncated
cast	object reference	type
constant	output fields	type conversion
declare	postfix	Unicode
decrement operator	prefix	variable type
identifier	primitive variables	variables

Java key words/syntax introduced in this chapter:

(and)	--
*	*=
/	/=
%	boolean
+	char
+ (with Strings)	double
++	false
-	final
--	int
-=	String
+=	true

Methods introduced in this chapter:

System.out.printf()	String.format()
---------------------	-----------------

Scanner methods

next ()	nextInt ()
nextDouble ()	nextLine ()

Math methods (includes methods in appendix D)

abs()	pow()
max()	random()
min()	round()
PI - final variable	sqrt()

Character methods (includes methods in appendix D)

isDigit()	isUpperCase()
isLetter()	toLowerCase(char)
isLetterOrDigit()	toUpperCase(char)
isLowerCase()	

2.9 Questions, Exercises and Problems

1) Assume you are writing a Java program that is going to need memory locations for the following information. Write the required declaration statements for each of the following: You have two things to consider:

- i) type of data (`int`, `double`, `boolean`, `char` or `String`)
- ii) an appropriate name (don't forget to use camel notation)

For example, for a company's net sales we would have: `double netSales;`

- a) Age of an employee in years
- b) Customer's street address
- c) Mass of an object in a Science experiment
- d) Employee's rate of pay (e.g. \$7.45/ hour)
- e) 4-digit employee number
- f) Single digit product price code (e.g. A, B, C, D, ..., H)
- g) Customer's phone number
- h) Number of students in a class

2) Java stores characters using Unicode while earlier languages used ASCII. What is the difference between the two coding systems? Why does Java use Unicode? Are there any advantages to using ASCII?

3) Another primitive integer type in Java is a `byte`. A `byte` is an 8-bit integer. What range of values can we store in a `byte` variable?

4) For larger integers you can use a `long` variable. A `long` is a 64-bit integer. What range of values can we store in a `long` variable?

5) Why would you want to use the `final` modifier when declaring a variable? Give a specific example.

6) How are primitive data types different from object references? Explain your answer by comparing what happens when you declare and initialize an integer variable to what happens when you declare and initialize a string variable.

7) List the methods for inputting each of the primitive types.

8) What is the best way to read in a `String`? Explain.

9) Evaluate the following numeric expressions. In each case, also indicate whether the final result will be a `double` or an `int`.

- | | | |
|--|------------------------------------|----------------------------------|
| a) <code>58 / 6</code> | b) <code>31 % 10</code> | c) <code>17 * 2 + 3 * 4.5</code> |
| d) <code>3.0 + 117 % 7</code> | e) <code>14 + 18 / 4</code> | f) <code>4.2 * 3</code> |
| g) <code>(int) Math.round(3.78)</code> | h) <code>Math.max(15, 14.2)</code> | i) <code>Math.sqrt(7)</code> |
| j) <code>Math.min(12, 7)</code> | k) <code>Math.pow(3, 2)</code> | |

14) Enter the following Java program:

```
int number = 2147483647;
System.out.println (number);
number++;
System.out.println (number);
```

- Before you run this program, try to predict the output.
- Run this program and see if your prediction came true.
- Looking over the Summary of Java Primitive Data Types table in 3.1, explain what is happening in this program.
- Change the statement `number++;` to: `number += 10;` and try to predict the outcome. Check your prediction.

For the following problems you should prepare a complete plan including sample input/output, a list of required memory (type and contents) and a complete list of all steps required to solve the problem before writing the Java code. You should also include a test plan (see Appendix B.3 in Java text) with a variety of test cases for each problem.

15) Write a program that finds the surface area and volume of a cylinder given its height and radius. You should round your answers to one decimal place. The formulae for the surface area and volume of a cylinder are:

$$SA = 2\pi rh + 2\pi r^2 \qquad V = \pi r^2 h$$

Note: For π you can use the built in constant `Math.PI`

16) Write a program that rounds a number to the nearest 1, 10, 100 or 1000. The user should enter the number to be rounded as well as how it should be rounded. For example, if you wanted to round 5687 to the nearest 10, the answer would be 5690. See section 2.7 for a hint.

17) A slice of pizza contains 355 calories. Cycling at a certain pace burns around 550 calories per hour. Write a program to figure out how long you would have to cycle to burn off a certain number of pizza slices. Give your answer in hours and minutes (rounded to the nearest minute).

18) Given an amount of change less than one dollar, find the coins required to make up this amount. Your program should find the minimum number of coins. Since we no longer have pennies in Canada, you should round your amounts to the nearest nickel before calculating the number of coins required. For example, if the change was 46¢, you would give 45¢ back made up of 1 quarter and 2 dimes for a total of 3 coins. If the change amount was 79¢, you would give 80¢ back made up of 3 quarter and 1 nickel for a total of 4 coins. **Hint:** Use the `%` and `/` operators.

19) Assume you want to find out how much longer one song is than another. The song times and the difference in song times should be given in minutes:seconds. Here is a sample input/output for this problem:

Calculating Differences in Song Times

```
Please enter the name of the longer song: American Woman
Please enter the length of American Woman (min:sec): 5:10
Please enter the name of the second song: These Eyes
Please enter the length of These Eyes (min:sec): 3:45
```

American Woman is 1:25 longer than These Eyes.

Thank you for using the Song Time Difference Calculator

You can assume that the first song entered is the longer song. You should complete a test plan for this program before writing the code.

Hint 1: You can change the delimiter used to split up your input when entering data using a Scanner with the `useDelimiter` method. For example, to add the ':' character as a delimiter, we would use: `keyboard.useDelimiter("[:\\s]");` The list of delimiter characters (inside the `[]`) includes ':' and `\\s` (for whitespace).

Hint 2: You may want to keep track of the time of each song and the difference in time between the songs in total seconds. Also, you can use the `System.out.printf` method to output the time difference in a min:sec format (e.g. 1:07). Your code for this problem should be both simple and easy to follow.

20) Enbridge Gas needs a program to calculate the monthly bill for each customer. The amount of each bill is based on the amount of gas used by each customer each month and the following rates:

Fixed Customer Charge	\$19.25 per month
Gas Delivery Charge (cost of transporting the gas)	13.3452 ¢ per m ³
Gas Supply Charge (cost of natural gas)	17.7303 ¢ per m ³

Write a complete Java program to produce a simple customer bill. Your bill should include the customer's name, the current month (e.g. February), how much gas the customer used this month (in cubic metres – no decimals) and a detailed break down (show all amounts) of the monthly charges including the total amount of the bill. Note: In addition to the above charges, HST of 13% is added to the total of each bill. Since the given rates are the same for each customer, you should use constants (`final`) for each of these rates including the HST. For example, at the top of your program you would have: `final double CUSTOMER_CHARGE = 19.25;`

To help solve this problem don't forget to create a detailed sample input/output for this program to help identify your inputs, outputs and required memory locations. Looking at this sample input/output you should be able to identify the steps required to complete the program. You should use the `System.out.printf` method to help line up the decimal places for the amounts on your bill. You should also complete a test plan to help test your program (see Appendix B.3)

Chapter 3 - Java Control Structures

So far our programs have followed a simple top to bottom sequence with each statement executing in order. For more complicated programs we can change the order of program flow using selection (ifs) and repetition (loops) structures. In each of these structures we use boolean expressions (also known as conditions) to determine the order of the program flow.

3.1 Boolean Expressions (Conditions)

Boolean expressions are expressions that take on the value of true or false. They are used to check if a certain condition is true or false. The following operators are included in boolean expressions:

Symbol	Example	Meaning
<code>==</code>	<code>a == b</code>	a equals b
<code><</code>	<code>a < b</code>	a less than b
<code>></code>	<code>a > b</code>	a greater than b
<code><=</code>	<code>a <= b</code>	a less than and equal to b
<code>>=</code>	<code>a >= b</code>	a greater than and equal to b
<code>!=</code>	<code>a != b</code>	a not equal to b (Note ! instead of not)

Notes:

1) Note the double equal (`==`) for equals, this is very important. A single equal (`=`) is for assignment statements but a double equal is for comparison.

2) For `<=`, `>=` and `!=` the equal sign is always second.

Compound Boolean expressions can be made by joining one or more simple Boolean expressions with the operators `&&` (and), `||` (or), and `!` (not)

Given a and b are boolean expressions then:

`(!a)` is false if a is true and true if a is false.

`(a || b)` is true if either a or b or both are true; it is false otherwise.

`(a && b)` is true only if both a and b are true; it is false otherwise.

To check if a variable is in a certain range (e.g. `10 <= number <= 15`) you should use the compound expression:

```
(number >= 10 && number <= 15)
```

The following are not valid boolean expression:

```
(10 <= number <= 15)
```

```
(number >= 10 && <= 15)
```

3.2 Selection Structures

Sometimes in our programs we only want to execute a certain statement if a certain condition is first met. For example, if we wanted to count the number of students with honours (a mark of 80 percent or above), we could use the following section of Java code:

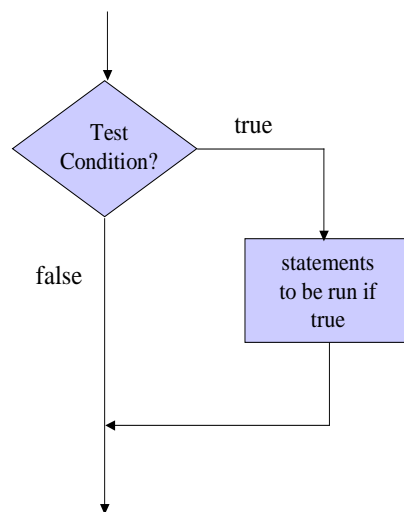
```
if (mark >= 80)
    noOfHonourStudents++;
```

Note that the boolean expression (condition) is between parentheses. In this example, the expression `noOfHonourStudents++` is only executed if the condition is true (mark is greater than or equal to 80). If the condition is false this expression is skipped and the program continues on with the next line of code. Also note, unlike Turing, there is no "then" or "end if".

The code in the `if` statement has been indented to show that the second line isn't part of the normal flow of the program. You should always indent/format your `if` structures to make them easier to follow. Fortunately, the Eclipse IDE will automatically format your code if you press `Ctrl+Shift+F`. If your code does not format properly, look for extra spaces or errors in your code.

The following flowchart shows the program flow in an `if` statement

Selection (if)



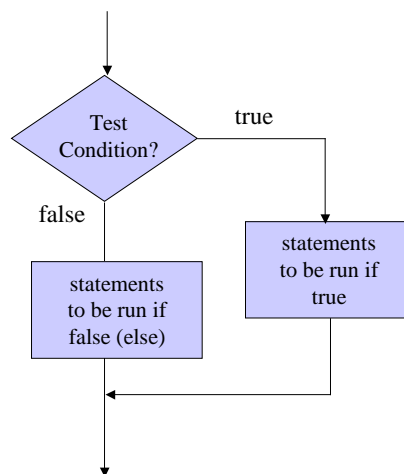
Normally, only the first statement after the `if` condition is executed when the condition is true. However, if we want to include more than one statement inside of our `if`, we can group statements into a compound statement using curly braces. For example:

```
if (mark >= 80)
{
    System.out.println("Congratulations on getting honours");
    noOfHonourStudents++;
}
```

Sometimes you want to do one thing when the condition is true and another thing when the condition is false. We can do this using an `if...else` structure.

The following flowchart shows the program flow in an `if...else` statement:

Selection (if else)



An example Java statement using the `if else` structure is given by:

```
if (mark >= 80)
{
    System.out.println("Congratulations on getting honours");
    noOfHonourStudents++;
}
else
{
    System.out.println("Sorry you didn't get honours");
}
```

In this example, the second set of curly braces is not required since only a single statement is included in the `else`. However, they were included to balance with the curly braces in the first part of the `if`. You can always include the curly braces even if there is only one statement after the `if` condition. In fact, some people recommend always using curly braces, even if you have only one statement to execute.

Here is another example showing when to use an "else" statement:

```
if (temperature >= 30)
    System.out.println("It is a very hot day");
if (temperature < 30 && temperature >= 20 )
    System.out.println("A nice summer day");
if (temperature < 20)
    System.out.println("A little cool for July");
```

In this example, only one condition can be true at a time, so we can make this code more efficient by using else statements. For example:

```
if (temperature >= 30)
    System.out.println("It is a very hot day");
else if (temperature >= 20) // and temperature < 30
    System.out.println("A nice summer day");
else // temperature < 20
    System.out.println("A little cool for July");
```

In this second example if the first condition is met the message "It is a very hot day" is displayed and then control jumps to the end of the if structure. Since there is an else before the second if this condition will only be checked if the temperature is below 30. Therefore the message "A nice summer day" is only displayed if the temperature is greater than or equal to 20 and less than 30. Finally the last message "A little cool for July" will only be displayed if the first two if's are false (temperature below 20). The comments are included to help clarify the function of the code.

Summary of the different forms of the if statement:

<pre>if (condition) statement_{TRUE};</pre>	<pre>if (condition₁) statement₁ _{TRUE}; else if (condition₂) statement₁ _{FALSE}, ₂ _{TRUE}; else statement_{BOTH 1 & 2} _{FALSE};</pre>
<pre>if (condition) statement_{TRUE}; else statement_{FALSE};</pre>	

Note: "statement" in the above examples can either be a single statement or a compound statement (more than 1 statement between curly brackets). Also note that there are no semi-colons after the conditions. Finally, in the third form you can add additional else if statements after the first else if and the final else is optional.

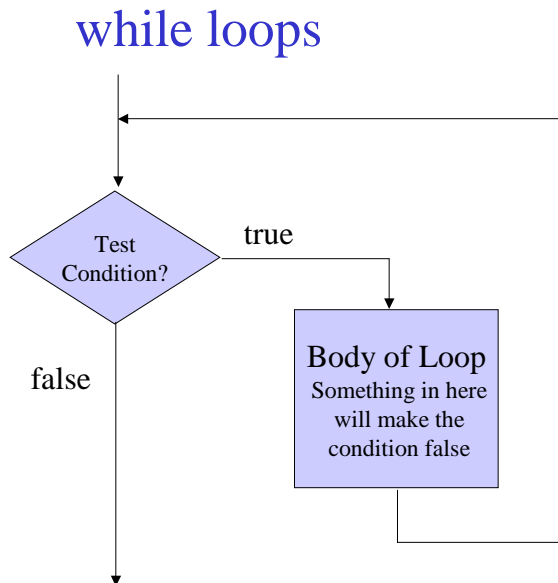
In Java we have 3 possible types of loops. We will group these into two basic types, conditional loops and counter loops.

3.3 Conditional Loops

Conditional loops are used when you want to repeat a single statement or a group of statements while a certain condition is true. There are two forms:

The `while` loop

The following flowchart shows the program flow in a `while` loop



The Java code for the `while` loop is given below:

```
while (condition)
{
    // body of loop
}
```

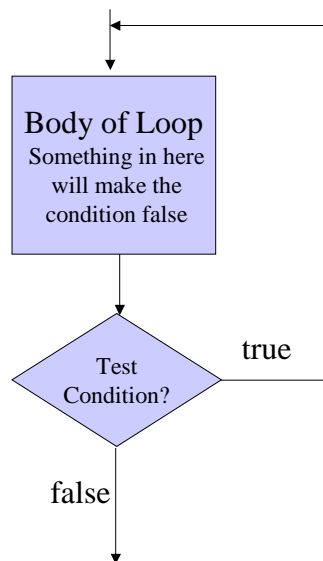
In the `while` loop the body of the loop is executed repeatedly as long as the condition is true. The condition is evaluated before the loop is executed, so if the condition is false at the beginning, the loop is not executed at all.

The other type of conditional loop is called a `do while` loop. In the `do while` loop, the condition is checked at the end of the loop so the body of the loop is always executed at least once.

The `do while` loop

The following flowchart shows the program flow in a `do while` loop:

do while loops



The Java code for the `do while` loop is given below:

```
do
{
    // body of loop
}
while (condition);
```

Notice that since the condition is at the end of the loop there is a semi-colon after the condition. This is only for the `do while` loop. Never put a semi-colon after the condition in a `while` loop.

As you can see these two loops are very similar. Which loop you use will depend on the situation. If you have a situation where you are going to do something at least once then you should probably use a `do while` loop. On the other hand, if you have a situation where you may never enter the loop in the first place, you should use a `while` loop. In general, `while` loops are probably used more than `do while` loops.

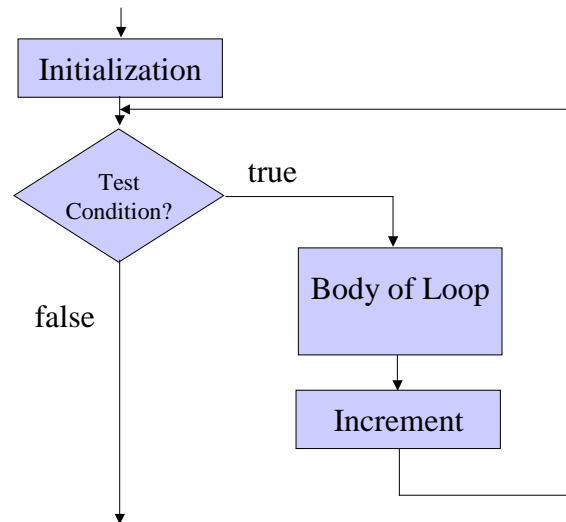
Note: In Turing we used the "exit when" statement to tell the computer when to exit the loop. In Java, we do the opposite. The condition in all Java loops tells the computer when to continue the loop, instead of when to exit the loop.

3.4 Counter Loops

Counter loops are used when you want to repeat a single statement or a group of statements a fixed number of times. (Note: In Java the for loop is very different than the for loop in Turing)

The following flowchart shows the program flow in a `for` loop

for (counter) loops



The Java code for the for loop is given below:

```
for (initialization; condition; increment)
{
    // body of loop
}
```

initialization - executed once at the start of the loop

condition - test condition, tested each time before entering the loop

- only enter or continue if this expression is true

increment - evaluated after each execution of the body

For example, the following loop will output the numbers 1, 2, 3, ..., 10:

```
for (int count = 1; count <= 10; count++)
{
    System.out.println(count);
}
```

In `for` loops the control variable (e.g. `count`) is usually an integer or a character. In most cases, you won't need this control variable outside of the `for` loop so it is a good idea to declare this variable when you initialize it, as we have done in the above example. In this case, the scope (where it can be used) of the variable `count` is only inside the loop.

Just like a `while` loop or a `do while` loop, the middle condition in a `for` loop tells the computer when you want to continue the loop and not when to quit the loop. Although `for` loops in Java are more complicated than `for` loops in Turing, they give you more control over the looping process. For example, in the following loop we check two conditions when deciding whether or not to continue the loop:

```
for (int divisor = 2; divisor <= upperLimit && !found; divisor++)
{
    // Body of loop: divisor counts from 2 to the upperLimit
    // We can also quit the loop early by setting found = true
}
```

3.5 Example Program using Selection and Loop Structures

To help demonstrate the use of selection and loop structures, the following example program is presented that converts a temperature from degrees Fahrenheit to degrees Celsius or visa versa.

Look over this code carefully and make sure you understand every line. The `\u00b0` in the print statements is the Unicode character for degree (°) that is used to display °F and °C. You may also notice that we declare the `celsiusTemp` and `fahrenheitTemp` variables at the top of the program. We could have declared these variables when they were read in but since we have two sections of code that use the same variables we declared them as shown. On the other hand, the `tryAgain` variable needs to be declared at the top of the program instead of inside the loop. If we declare a variable inside of a loop, its scope (where it can be used) is limited to the inside of the loop. This would cause a problem for the `tryAgain` variable, since it is used in the `while` condition which is outside of the loop after the closing curly brace.

```
import java.util.Scanner;

/**
 * The Temperature Conversion program. Purpose: To convert a temperature from
 * Fahrenheit to Celsius or from Celsius to Fahrenheit
 *
 * @author ICS3U
 * @version updated September 2013
 */

public class TemperatureConversion
{
    public static void main(String[] args)
    {
        // Set up the Scanner for keyboard input and
        // declare the main variables
        Scanner keyboard = new Scanner(System.in);
        double celsiusTemp, fahrenheitTemp;
        char tryAgain;

        // Display an intro title
        System.out.println("Welcome to the Temperature Conversion Program");
```

```

// Loop to process each conversion
do
{
    // Inputs the type of conversion you would like to do
    System.out.println("Enter 1 to convert from \u00b0F to \u00b0C");
    System.out.println("Enter 2 to convert from \u00b0C to \u00b0F");
    System.out.print("Enter your Choice: ");
    int typeOfConv = keyboard.nextInt();

    // Check that the input is valid. User will re-enter until valid
    while (typeOfConv < 1 || typeOfConv > 2)
    {
        System.out.print("Choose 1 or 2, please re-enter: ");
        typeOfConv = keyboard.nextInt();
    }

    // Ask for a temperature and convert it and then
    // display the results. There are two different
    // routines depending on the type of conversion
    if (typeOfConv == 1)
    {
        System.out.print("Enter the temperature in degrees Fahrenheit: ");
        fahrenheitTemp = keyboard.nextDouble();

        celsiusTemp = (fahrenheitTemp - 32) * 5 / 9;

        System.out.printf("%.1f\u00b0F is equal to %.1f\u00b0C\n",
                           fahrenheitTemp, celsiusTemp);
    }
    else
    {
        System.out.print("Enter the temperature in degrees Celsius: ");
        celsiusTemp = keyboard.nextDouble();

        fahrenheitTemp = celsiusTemp * 9 / 5 + 32;

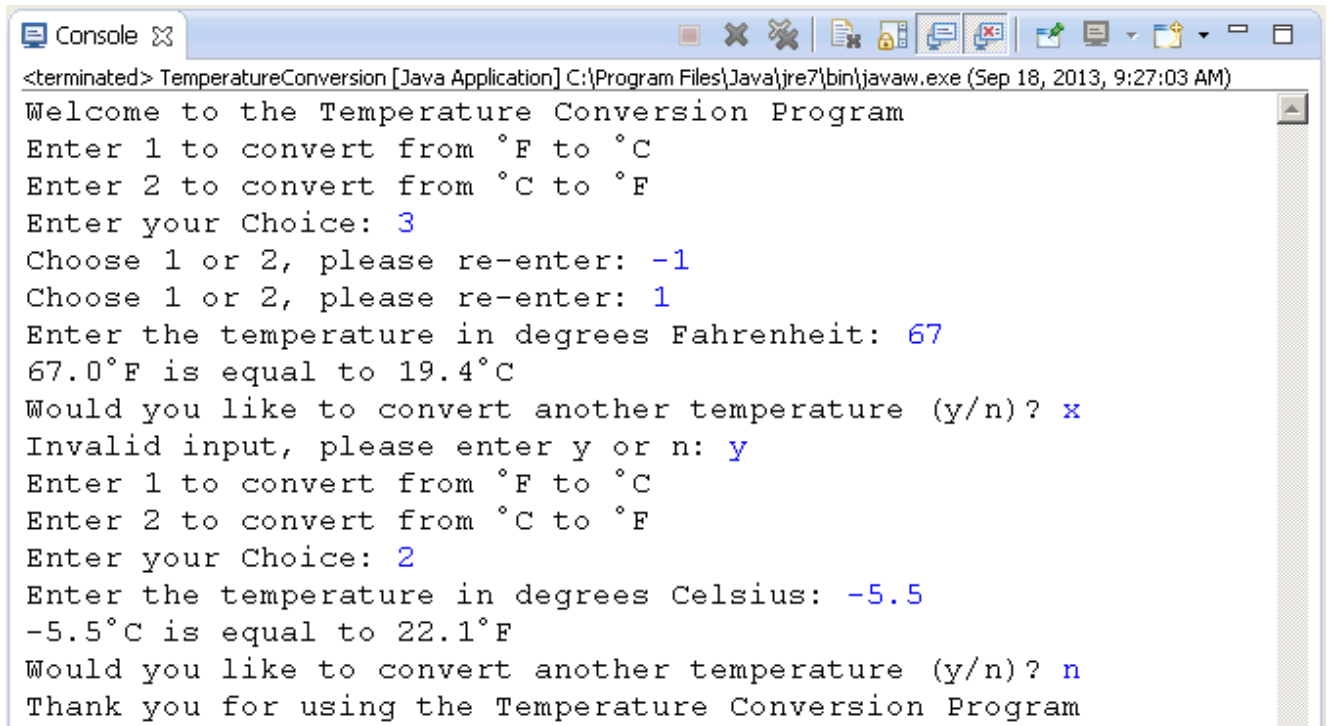
        System.out.printf("%.1f\u00b0C is equal to %.1f\u00b0F\n",
                           celsiusTemp, fahrenheitTemp);
    }

    // Ask if the user would like to try again
    // Include an error check to see that answer is y or n
    System.out.print("Would you like to convert another temperature (y/n)? ");
    tryAgain = Character.toLowerCase(keyboard.next().charAt(0));
    while (tryAgain != 'y' && tryAgain != 'n')
    {
        System.out.print("Invalid input, please enter y or n: ");
        tryAgain = Character.toLowerCase(keyboard.next().charAt(0));
    }
}
while (tryAgain == 'y');

keyboard.close();
System.out.println("Thank you for using the Temperature Conversion Program");
} // main method
} // TemperatureConversion class

```

A sample run of the above program is shown on the next page:



```
<terminated> TemperatureConversion [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Sep 18, 2013, 9:27:03 AM)
Welcome to the Temperature Conversion Program
Enter 1 to convert from °F to °C
Enter 2 to convert from °C to °F
Enter your Choice: 3
Choose 1 or 2, please re-enter: -1
Choose 1 or 2, please re-enter: 1
Enter the temperature in degrees Fahrenheit: 67
67.0°F is equal to 19.4°C
Would you like to convert another temperature (y/n)? x
Invalid input, please enter y or n: y
Enter 1 to convert from °F to °C
Enter 2 to convert from °C to °F
Enter your Choice: 2
Enter the temperature in degrees Celsius: -5.5
-5.5°C is equal to 22.1°F
Would you like to convert another temperature (y/n)? n
Thank you for using the Temperature Conversion Program
```

You may have noticed a couple of input error-checking loops were included in the above program. The first error check is used to check if the `typeOfConv` input is valid (1 or 2). If an invalid conversion type is entered, the program continues to ask the user to re-input until a valid type is entered.

The second error check tests if the `tryAgain` input is valid ('y' or 'n'). Since `next()` reads in a String we use `charAt(0)` to get the first character of the String ('y' or 'n'). This input character is then converted to lower case using the `Character.toLowerCase` method before it is stored – making both 'Y' and 'N' valid inputs as well as 'y' and 'n'.

You should always include input error checks in your programs since invalid input may result in unexpected or incorrect output. The potential problems of input errors can be summed up by the popular computer acronym GIGO, which is short for Garbage In, Garbage Out. Since `typeOfConv` is an integer variable, you may have noticed that if the user inputs non-integer values such as 1.0 or "one", the program will not work. At this time you can assume that all user input will be the correct type. Later in the course we will introduce more advanced error checking techniques to handle these types of errors (see Appendix G for an example).

3.6 Pseudocode and Flowcharts

With more complicated problems, it is important to carefully plan out your solutions before writing any Java code. In addition to preparing a sample input/output which helps identify all of the required outputs and inputs and identifying your memory requirements, you need to describe the algorithm (steps required to solve the problem). When your solution includes decisions and/or loops you can use either pseudocode or flowcharts to help plan out and describe these steps.

Pseudocode describes the required steps to solve a problem using plain English statements. To help identify decision and loop structures, the English statements are indented. Because we are not writing code, it is important to avoid using Java specific code in your pseudocode. Well written pseudocode can help you to write your program comments.

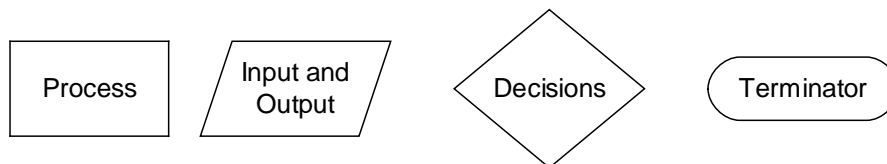
Here is sample pseudocode for problem 11 in the exercises. In this problem you are asked to enter a series of numbers and then find the total, average, smallest and largest number of the numbers entered.

```
Display a Title and Introduction
Set the total to zero
Set the number of numbers to zero
Set the smallest so far to a large value
Set the largest so far to a low value
While there are more numbers to process
    Ask for and enter the next number
    Add this number to the total
    Add 1 to the number of numbers
    If this number is greater than the largest so far
        Set the largest so far to this number
    If this number is less than the smallest so far
        Set the smallest so far to this number

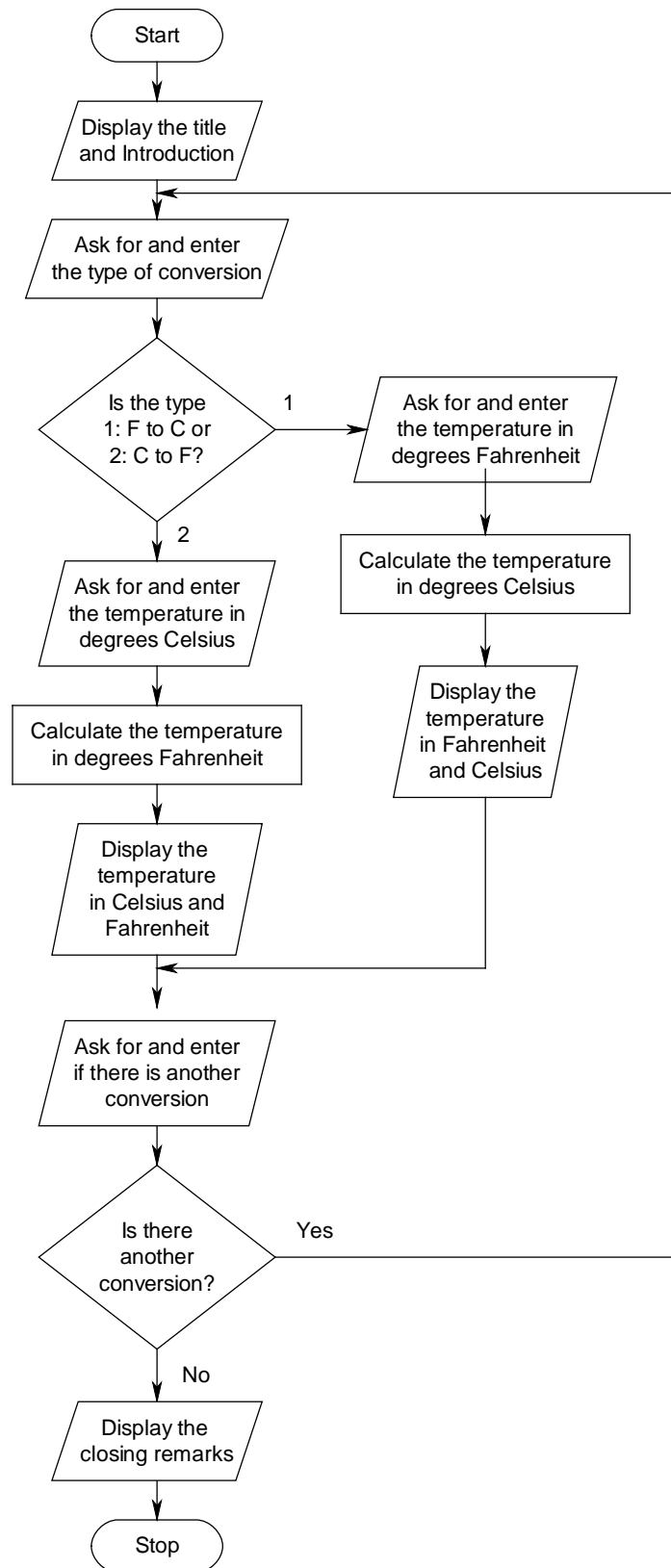
    Ask for and enter if there are any more numbers
    (Includes an error check)
Repeat the above steps while there are more numbers to enter

Calculate the average
Display the total, average, smallest number and largest number
Display a closing remark
```

Flowcharts were used earlier in the chapter to show the program flow for the various decision and loop structures. When using flowcharts, the four most common flowchart symbols (shapes) are a rectangle for process (including calculations), a parallelogram for input/output, a diamond for decisions and an oval for flow terminators (stop and start).



Using these symbols, here is a flowchart for the temperature conversion program presented earlier. To keep things simple, the input error checking loops were not included.



3.7 Chapter Summary

Keywords/phrases introduced in this Chapter. You should be able to explain each of the following terms:

boolean expression	counter loops	program flow
compiler error	exception	pseudocode
compound boolean exp.	flowchart	run time error
compound statement	increment	selection
condition	initialization	syntax error
conditional loops	logic error	test data
control structures	program efficiency	test plan

Java key words/syntax introduced in this chapter:

==	&&	if
<		if...else
>	!	if...else if...else
<=	while	(etc.)
>=	do while	
!=	for	

3.8 Questions, Exercises and Problems

1) Evaluate the following boolean expressions (Answers should be true or false):

a) (3 > 5 && 2 > 1)	b) (31 < 51 && 25 >= 14)
c) (18 == 15 14 <= 17)	d) (32 > 115 12 != 12)
e) (!(27 > 15))	f) !!true
g) (true false) && false	h) (false && true) true

2) Given the following selection structure:

```

if (first == second || second == third)
    System.out.print("1");
else if (first > second && second >= third)
    System.out.print("2");
else
    System.out.print("3");

if (first % 3 == 0)
    System.out.print("4");
  
```

Predict the output and indicate the number of comparisons required for the above section of Java code, given the following values of the first, second and third:

first	second	third	Predicted output	No of Comparisons
7	12	7		
18	13	5		
34	24	24		

3) Given the following sections of code – one with a while loop and one with a do while loop:

```
int number = start;
while (number < 5)
{
    number ++;
}
```

```
int number = start;
do
{
    number ++;
}
while (number < 5);
```

a) If start were 3 what would be the value of number after running:
the while loop? the do loop?

b) If start were 5 what would be the value of number after running:
the while loop? the do loop?

c) Explain the differences and similarities between a while loop and a do while loop.

4) Without typing these programs in to the computer, predict the exact output of the following sections of Java code. For each question also show the value of all variables (memory trace) as the program runs. The "\n" is a new line character.

a) **double** product = 1.5;
 for (**int** count=1; count<=5; count++)
 {
 product = product*count;
 System.out.print(count + " ");
 }
 System.out.println("\n" + product);

Memory Trace

<u>count</u>	<u>product</u>
--------------	----------------

Output

b) **int** first = 43;
 int second = 10;

 System.out.println
 ("Working with While Loops");
 while (first > second)
 {
 System.out.println(first + " " + second);
 first -=5;
 second +=6;
 }
 System.out.println("That's all folks");

Memory Trace

<u>first</u>	<u>second</u>
--------------	---------------

Output

5) Comparing the following two code segments. How are they the same? How are they different? Which one is better? Explain.

```

if (noOfDimes == 1)
    System.out.print("1 dime");
if (noOfDimes > 1)
    System.out.print(noOfDimes + " dimes");

if (noOfDimes == 1)
    System.out.print("1 dime");
else if (noOfDimes > 1)
    System.out.print(noOfDimes + " dimes");

```

6) Convert the following for loop code to a while loop.

```

int total = 0;
for (int count = 1; count < 10; count++)
{
    total +=count;
}
System.out.println("The total is: " + total);

```

7) The following Java program should read in a series of percentage marks and then count the number of passing marks (50 or above) and the number of failing marks. The program keeps entering marks until -1 is entered which signifies that there are no more marks to process. Find the errors in this program. For each error, indicate whether the error is a compiler/syntax error, a logic error or a run-time error. Also, explain how you would correct each error. See Appendix B.1 for an explanation of each type of error.

```

import Scanner;
class ErrorProgram
{
    public static void main ()
    {
        Scanner keyboard = new Scanner (System.out);
        double mark;
        int passCount;
        System.out.println ("Counting the passes and failures");
        do
        {
            System.out.print ("Please enter a mark (Enter -1 to quit): ");
            mark = keyboard.nextInt ();
            if (mark < 50)
                passCount++;
            else
                failCount++;
        } while (mark = -1);
        println ("Number of passing marks: " + passCount);
        System.out.println ("Number of failures: " + failCount);
        System.out.println ("Total marks entered: " + passCount + failCount);
        System.out.println ("Thank you for using this program");
    } // main method
} // ErrorProgram class

```

8) Assume that you are writing a Java program where at one point you are going to ask the user to input his/her month of birth as a number (1 - 12). Write the required Java code (don't write the whole program) to handle the input of this information. If the month entered is an illegal month, you should tell the user that he/she has made a mistake and then let the user re-enter. Each time the user re-enters you should keep checking the input until a legal month number is entered.

For the following questions you should think about how you are going to approach these problems before going to the computer.

In your plan/analysis you should think about the output, input, process (pseudocode or flowchart) and memory for each problem.

9) Letter grades are assigned as follows:

A - mark in the 80's or 90's	B - mark in the 70's	C - mark in the 60's
D - mark in the 50's	F - mark less than 50	

Write a Java program that reads in a student's numeric grade and finds and displays the corresponding letter grade.

10) Write a Java program that uses a for loop to display the whole numbers from 1 to 15 along with their squares and cubes. You should display the numbers in a nice table with column headings. Hint: Use `printf` to display the numbers in nice columns (see 2.4).

Example partial output:

Squares and Cubes		
Number	Square	Cube
1	1	1
2	4	8
3	9	27
.	.	.
.	.	.
15	225	3375

11) Write a Java program that inputs one or more integer numbers. The number of numbers is not known ahead of time so after entering each number, you should ask the user if he or she wants to enter another number. The program should calculate and display both the total and the average of the numbers entered. It should also find and display the value of the largest and smallest number entered.

12) Write a Java program which draws a rectangle of "*" on the screen. Your program should allow the user to input the height and the width of the rectangle. The required code to solve this problem is quite simple if you plan out your code carefully. Extra: Only draw the outline of the rectangle (see example below). The following examples are for a rectangle with a height of 4 and a width of 6.

Normal:	*****	Outline:	*****
	*****		* *
	*****		* *
	*****		*****

13) A triangle can be classified as equilateral (3 equal sides), isosceles (2 equal sides) or scalene (no equal sides). In addition, a triangle may also be a right-angle triangle. Write a program that inputs the 3 side lengths of a triangle and then states what type of triangle it is. If it is not possible to form a triangle with the given side lengths, your program should display an appropriate message. Example results are shown in the table below:

First Side	Second Side	Third Side	Description of the Triangle
3	4	5	Scalene right-angle triangle
7	7	7	Equilateral triangle
9	9	12	Isosceles triangle
12	3	8	Undefined triangle

Hint: Use Pythagoras' Theorem to find out if a triangle is a right-angle triangle.

14) Orion Entertainment sells CD's for \$13.99 and DVD's for \$19.99. If the total number of items purchased by a customer is 10 or more, the customer receives a 10% discount on his or her entire purchase. HST (13%) is then added to the price after the discount to calculate the total price.

Write a Java program that produces a bill for each customer. After asking how many CD's and DVD's are being purchased, the program should display a bill that shows the number and value of the CDs and DVD's purchased, the amount of the discount (if any), the total before tax, the HST and the total amount of the purchase.

Your program should be able to handle more than one customer. After all of the purchases have been made, the program should give a summary of the day's sales showing the total number of customers and the total overall sales.

To make your program easier to update, the prices, rate of discount and the tax rate should be stored as constants (final) defined at the beginning of the program. For example:

```
final double CD_PRICE = 13.99;
```

15) A number palindrome is a 'symmetrical' number that remains the same when its digits are reversed. For example, 123454321 and 7938228397 are both number palindromes. Write a Java program that lets the user input a long integer and then tells the user whether the number is a palindrome. Hint: use % and / to look at a numbers individual digits. We use a long instead of int for your number variable so that your program can handle larger integers.

16) The multiplicative persistence of a number is the number of times you need to multiply the digits of the number together to get a single digit number. For example the multiplicative persistence of 77 is 4 because it takes 4 steps ($7 \times 7 = 49$, $4 \times 9 = 36$, $3 \times 6 = 18$ and $1 \times 8 = 8$) to get a single digit product.

Write a Java program to find the multiplicative persistence of a number. You can assume that the number entered will be a positive integer. To help come up with an algorithm to solve this problem, work out the multiplicative persistence of 679 on paper and make a note of all of the steps required.

17) The following problem is taken from the University of Waterloo's Canadian Computing Competition. Read over this problem very carefully and plan out your solution before starting on the Java code. Don't forget to test your plan and program carefully to make sure that it can handle all possible inputs.

Sumac Sequences

Problem Description

In a sumac sequence, t_1, t_2, \dots, t_m , each term is an integer greater than or equal to 0. Also, each term, starting with the third, is the difference of the preceding two terms (that is, $t_{n+2} = t_n - t_{n+1}$ for $n \geq 1$). The sequence terminates at t_m if $t_{m-1} < t_m$.

For example, if we have 120 and 71, then the sumac sequence generated is as follows:

120, 71, 49, 22, 27

This is a sumac sequence of length 5.

Input Specification

The input will be two positive numbers t_1 and t_2 , with $0 < t_2 < t_1 < 10000$.

Output Specification

The output will be the length of the sumac sequence given by the starting numbers t_1 and t_2 .

Sample Input

120

71

Output for Sample Input

5

Extra: What is the largest possible Sumac Sequence we can generate, with the given range of values for t_1 and t_2 ?

Chapter 4 –Methods

4.1 Introduction to Methods

In Turing, subprograms (sections of code that perform a specific task) are called **procedures** or **functions**. In Java, subprograms are called **methods**. As we saw in Chapter 1, every object has characteristics (data elements) and behaviour (functionality). We use methods to describe an object's behaviour.

Since Java is an object-oriented programming language we usually start by identifying the types of objects we will need to solve a problem and then for each object we look at the characteristics (data elements) and behaviour (methods) we will need. However, before we look at some of these object-oriented design concepts we are going to start with more of a top-down approach to problem solving. This means that, to start, we will be writing mostly static methods.

When faced with writing the code for a large project, the task can be overwhelming. Both objects and methods allow us to break a task down into smaller more manageable sub problems or tasks. This approach is known as top-down design or divide and conquer.

Dividing a program (i.e. Java class) into methods, with each method performing a specific well-defined task, is an effective way to implement a software solution for a given problem.

Advantages of top down design and methods:

- Since each sub-problem is less complicated, it is easier to solve. Therefore using methods makes it is easier to develop software.
- Breaking a problem into parts helps to clarify solution requirements and what is needed. Therefore using methods makes it is more likely that your code will meet requirements and solve the given problem.
- By dividing up a problem into smaller problems, you isolate unique parts that do not overlap. Therefore, methods make your programs smaller by eliminating duplicate code.
- Solutions to smaller, simpler problems are easier for others to understand. Therefore methods make your code easier to understand
- When you have multiple parts, each of which solves a basic problem, it is more likely that you will be able to re-use these parts. Therefore methods make your code re-usable.
- With more than one problem to solve, it is usually easier for more than one person to work on the solution. Therefore using methods makes it easier to write code when working with a team of software developers.

4.2 Instance Methods vs. Static Methods

In Java, every method is contained within a class. Each class can have instance methods and/or static methods (also known as class methods).

Instance methods act upon an instance of a specific class; therefore we need to give the name of a reference to an object instance when calling an instance method. For example, if `name` is a reference to a `String` object, we can use the `length` method to find the length of the `String` and the `charAt` method to find the value of individual characters in the `String`.

```
String name = "Jane Austen";  
int length = name.length();           // Sets length to 11  
char letter = name.charAt(0);         // Sets letter to 'J'
```

Since instance methods look at the specific characteristics of individual objects, we need to put the name of the object reference followed by a dot ('.') before the method name (e.g. `name.length()`). In this case `name` acts like another parameter to indicate which `String` we want to find the length of.

When using `Scanners` we are also using instance methods. For example, given:

```
Scanner keyboard = new Scanner (System.in);  
int number = keyboard.nextInt();
```

the `nextInt()` method looks for the next integer to be input from the `Scanner` object called `keyboard` which was created to look for input from the standard keyboard input (`System.in`).

Static or class methods are general methods that don't operate on a particular object. Therefore you do not need to create an object (instance of the class) to use these methods. However, since these methods are part of a particular class, you do need to indicate which class they belong to by putting the class name followed by a dot ('.') before the method name. We have already seen static methods in the `Math` and `Character` classes. For example to use the `sqrt()` method in the `Math` class we use a statement such as:

```
double answer = Math.sqrt(16);
```

In this case the "`Math.`" indicates which class the `sqrt` method is in. In the next section we will see that, when we are creating methods to be used within the same class, the class name and dot ('.') before the method name are optional.

Since you do not need to create an instance of an object to use static methods, they are similar to the procedures and functions used in procedural (non object-oriented) languages. Therefore, to introduce methods, the methods we will be creating in this chapter will all be static methods. In Chapter 6, when we create our own classes/objects, we will be looking at how to create instance methods.

4.3 Writing Your Own Static Methods

In this section we will write a simple method to calculate and return the area of a triangle. After deciding on the task you want your method to perform, you should consider what inputs (parameters) the method will need from the calling program in order to perform this task. By using method parameters we can change a method's inputs making each method more versatile. The method's output is what it returns back to the calling program. For the area of a triangle method we can summarize the method inputs and outputs as follows:

<u>Method Inputs (Parameters)</u>	<u>Method Output (Return Value)</u>
Base of the triangle (double)	Area of the Triangle (double)
Height of the triangle (double)	

When naming your methods you should pick a descriptive name that describes what the method is returning or, when the method returns nothing, what the method does. In our example since the method is finding and returning the area of a triangle, a good name would be `areaOfTriangle()`. Like variables, you should use camel notation in method names. The code for this method and a main program to test the method are given on the next page.

As you can see, the new method is placed above the `main` method inside the `TriangleClass` code. The comments given above the method are standard `javadoc` comments. The two new tags `@param` and `@return` are used to describe the method's parameters and return value. You should always include comments for each method you write.

The method heading is very similar to the heading for `main`. However, unlike `main`, the first modifier is `private` since we are only using this method inside the `TriangleArea` class. The second modifier tells us that this is a `static` method. All of the methods we will be writing will be static until we create our own classes in later chapters. The `double` indicates that the value returned (area in this case) will be a `double`.

Finally, inside the parenthesis we have the two parameters and their types. You should choose the names of your parameters carefully so that they describe the information you are passing into the method.

Underneath the heading between 2 curly braces we have the body of the method that calculates the area of a triangle using the formula $A = \frac{1}{2}bh$ and then returns the result to the calling program. Every method that doesn't have a `void` return type must return an appropriate value.

Below the method code is the code for the main program that calls the `areaOfTriangle()` method. Even though the `areaOfTriangle()` comes first, the program always begins with the main program.

```
// The "TriangleArea" class.
import java.util.Scanner;

public class TriangleArea
{
    /** Finds the area of a triangle
     * @param base    the length of the base of the triangle
     * @param height  the height of the triangle
     * @return        the area of the triangle
     */
    private static double areaOfTriangle (double base, double height)
    {
        double area = base * height / 2;
        return area;
    }

    public static void main (String [] args)
    {
        Scanner keyboard = new Scanner (System.in);
        System.out.println ("Finding the area of a triangle");

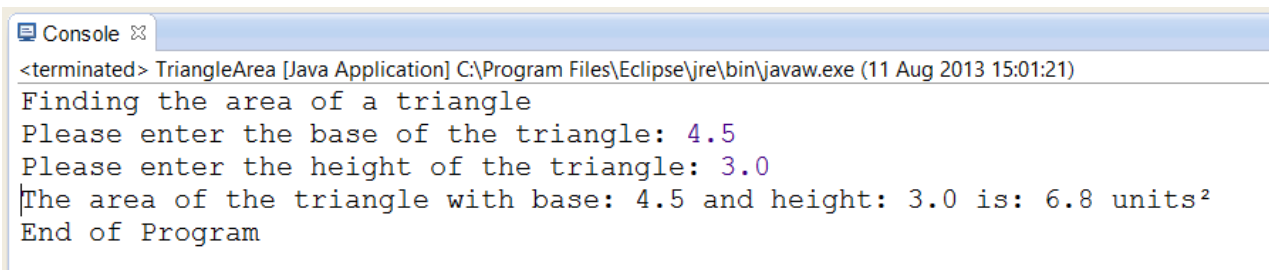
        System.out.print ("Please enter the base of the triangle: ");
        double enteredBase = keyboard.nextDouble ();
        System.out.print ("Please enter the height of the triangle: ");
        double enteredHeight = keyboard.nextDouble ();

        double triangleArea
            = areaOfTriangle (enteredBase, enteredHeight);

        System.out.printf("The area of the triangle with base: %.1f " +
                           "and height: %.1f is: %.1f units\u00b2%n",
                           enteredBase, enteredHeight, triangleArea);

        System.out.println ("End of Program");
    } // main method
} // TriangleArea class
```

Sample output from the above program:



```
Console
<terminated> TriangleArea [Java Application] C:\Program Files\Eclipse\jre\bin\javaw.exe (11 Aug 2013 15:01:21)
Finding the area of a triangle
Please enter the base of the triangle: 4.5
Please enter the height of the triangle: 3.0
The area of the triangle with base: 4.5 and height: 3.0 is: 6.8 units2
End of Program
```

Note: In the above code, the format string in the last `printf` command was quite long so we had to split it over 2 lines using a “+” to join each section. When splitting up a command that is too long we usually try to split it at a “,” but in this case we needed to split up the long string. Also, note the Unicode `\u00b2` was used to get the ‘2’ exponent for `units2` in the final output.

Looking over the code, the main program starts by printing a title and then asking for and entering the base and the height. Since we wanted our method to perform a specific task we left the input and output routines inside the main program. Generally, unless the purpose of a method is to input or output, you should not input or output information in a method. Next, the main program calls the `areaOfTriangle()` method with the statement:

```
double triangleArea = areaOfTriangle (enteredBase, enteredHeight);
```

At this point the program jumps up to the `areaOfTriangle()` method. The values in the main program variables `enteredBase` and `enteredHeight` are passed to their corresponding method variables `base` and `height`. This step is equivalent to the statements:

```
base = enteredBase;
height = enteredHeight;
```

The information is passed based on position and not the name of the parameters. Therefore, the first value in the calling program's list of parameters (actual parameter list or argument list) matches with the first value in the method's list of parameters (formal parameter list). If by mistake you called the method using:

```
double triangleArea = areaOfTriangle (enteredHeight, enteredBase);
```

the variable `enteredHeight` would be passed to `base` and the variable `enteredBase` would be passed to `height`.

The variables `base`, `height` and `area` are all **local** to the `areaOfTriangle` method. This means that they can only be used by code inside the `areaOfTriangle` method. Therefore, we need to use the "return" statement to send back the calculated area to the main program. The returned value is stored in the variable called `triangleArea` since this is the variable on the left-hand side of the equal sign.

Just like the parameters, the name of the variable that stores the returned value doesn't matter. Therefore, you don't need to call the variable in the main program `area` because you called it `area` in the method. You could, if you wanted to, but you don't have to.

In order to use a method we have to know what parameters we need to send in and then make sure we store the returned value in an appropriate variable. It is not necessary to know the details of what happens locally inside the method including the names of any variables. This allows us to easily use methods created by other programmers without worrying about how they implemented their methods.

When the method reaches the return statement it immediately returns to the main (calling) program which then continues with the next statement in sequence. In our example, the main program would then display the results.

Since `areaOfTriangle()` is a static method in the `TriangleArea` class we could have used the statement `TriangleArea.areaOfTriangle()` to call up this method. However, since we are calling this method from the `main` method, which is also in the `TriangleArea` class, the `"TriangleArea."` is not necessary.

In some cases we may want to include more than one `return` statement in our methods. For example:

```
private static double areaOfTriangle (double base, double height)
{
    if (base <= 0 || height <= 0)
        return 0;
    double area = base * height / 2;
    return area;
}
```

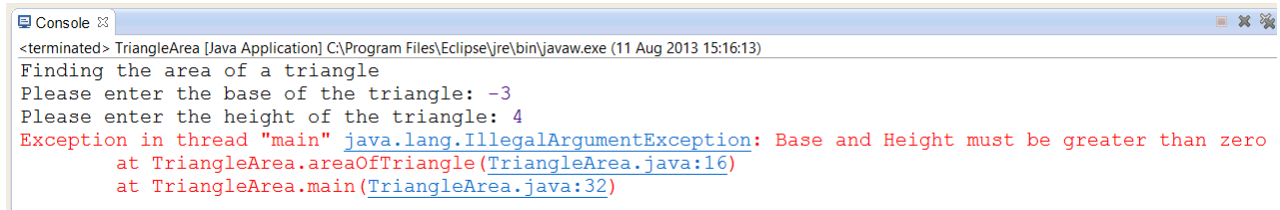
In this version of the `areaOfTriangle` method, if the `base` or `height` are less than or equal to zero, the method immediately returns a value of 0. The last two statements in the method are only run if the `base` and `height` are both greater than zero. Since the `return` statement transfers control back to the calling program immediately, an `else` statement is not required. When writing your own methods, you should take advantage of the fact that you can use a `return` statement to leave a method early.

If a method has a `void` return type we can still use a `return` statement with no value after the word `return` to leave the method early. In this case, the `return` statement is optional.

Since a negative `base` or `height` are not valid parameters (arguments) to the `areaOfTriangle()` method, an alternate method of dealing with this problem is to throw an `IllegalArgumentException` as is shown in the following example. Since our method now throws an `Exception`, we have updated the `javadoc` comments to include an `@throws` tag:

```
/** Finds the area of a triangle
 * @param base    the length of the base of the triangle
 * @param height  the height of the triangle
 * @return        the area of the triangle
 * @throws        IllegalArgumentException if either the
 *                base or height is negative
 */
private static double areaOfTriangle (double base, double height)
{
    if (base < 0 || height < 0)
        throw new IllegalArgumentException
            ("Base and Height must be non-negative");
    double area = base * height / 2;
    return area;
}
```

In this case, the calling program should make sure that both the base and the height are non-negative by including a couple of input error checking loops. In some cases you could also include a `try catch` block (see Appendix G) to handle the thrown exception. Note: If an exception is thrown when base or height is negative, program control is immediately passed back to the main program skipping over the area calculation, just like the early return statement. Using the updated method with the `IllegalArgumentException`, if we don't catch the exception, the program stops working and an error message is displayed in the Console window.

A screenshot of the Eclipse IDE's Console window. The title bar reads "Console". The text inside shows the execution of a Java application named "TriangleArea". It prompts the user to enter the base and height of a triangle. The user enters -3 for the base and 4 for the height. An exception is thrown: "Exception in thread \"main\" java.lang.IllegalArgumentException: Base and Height must be greater than zero". The stack trace shows the exception was thrown at "TriangleArea.areaOfTriangle(TriangleArea.java:16)" and originated from "TriangleArea.main(TriangleArea.java:32)".

```
<terminated> TriangleArea [Java Application] C:\Program Files\Eclipse\jre\bin\javaw.exe (11 Aug 2013 15:16:13)
Finding the area of a triangle
Please enter the base of the triangle: -3
Please enter the height of the triangle: 4
Exception in thread "main" java.lang.IllegalArgumentException: Base and Height must be greater than zero
    at TriangleArea.areaOfTriangle(TriangleArea.java:16)
    at TriangleArea.main(TriangleArea.java:32)
```

The exception shows the message we gave and then traces back the line numbers of the code that caused the problem. In this example, line 32 of the main program called the method with the arguments -3 and 4 and then on line 16 inside the method, the `IllegalArgumentException` was thrown. Again, you should include input error check loops in your code to avoid getting these exceptions in the first place.

You may have also noted that in the above error message the full names: `TriangleArea.areaOfTriangle` and `TriangleArea.main` are given for both the `areaOfTriangle` and `main` methods. This will be helpful in larger programs when you have more than one class.

4.4 Method Overloading

In Java like many other object oriented languages, we can overload a method by using the same name but a different number or type of parameters. The Java compiler then determines which version of the method to call based on the number and type of the parameters. By overloading methods, we have fewer method names to remember. The following example overloads a method called `max` that finds the maximum of two numbers. Since the `max` method is part of our `MaximumOverload` class we don't need to worry about a conflict with the name `max` in the `Math` class. If we want to use the `Math` class version of `max` in the same program we would use `Math.max`. **Note:** Comments have been omitted from this sample program to save space.

```
// The "Maximum" class.

public class MaximumOverload
{
    private static int max (int first, int second)
    {
        if (first > second)
            return first;
        else
            return second;
    }

    private static double max (double first, double second)
    {
        if (first > second)
            return first;
        else
            return second;
    }

    public static void main (String [] args)
    {
        System.out.println (max (5, 12));
        System.out.println (max (4.6, 7.9));
        System.out.println (max (78, 34.7));
        System.out.println (max ("one", "two"));
    } // main method
} // Maximum class
```

Since 5 and 12 are both integers, the first call to `max` will use the first version with two integer parameters. In the second example, since both 4.6 and 7.9 are double, the second version is used. In the third example we are passing an integer and a double to the method. Since we have no exact match for this call, the computer tries to find a usable match. Since an integer can be promoted to a double the second version will be used. However since this version returns a double, the result will be the double 78.0. In the final example, we have no possible match so this will generate a compiler error. We could have also included a version of `max` with 3 parameters since overloading also allows us to change the number of parameters as well as their type.

4.5 More on Method Parameters

Since different programmers and languages use different terms to describe parameters, the following is meant to clarify some of these terms. In the main program when we are calling a method with the following:

```
triangleArea = areaOfTriangle (enteredBase, enteredHeight);
```

the parameters `enteredBase` and `enteredHeight` are sometimes called arguments or actual parameters. This is why the exception we threw when these values were negative was called an `IllegalArgumentException`.

On the other hand, in a method heading such as:

```
private static double areaOfTriangle (double base, double height)
```

the parameters `base` and `height` are sometimes called formal parameters. When you call a Java method the actual parameters are passed to the formal parameters.

Another point that needs to be clarified is how parameters are passed into methods. Java always passes parameters by value. Therefore for primitive variables if you change the value of a parameter in the method it will not affect the corresponding variable (actual parameter) in the calling program.

However, if a parameter is a reference to an object, when you pass the value of the object reference, you are passing the address of the object in the main program. The corresponding variable in the method will be a different reference variable but it will refer to the same object. Therefore the method will be working with the original object and any changes to the object made inside the method will change the object in the main program.

By passing a reference to the original object, Java avoids the cost in both time and memory of creating a new object. Also, in some cases, we may want to change the original object from inside the method. However, there is the possible side effect that you accidentally modify an object in the calling program from inside the method. Fortunately (or unfortunately) since `String` objects are immutable (can't be changed), if you have a `String` parameter in a method, any changes inside the method will not change the corresponding `String` in the calling program. If you want to modify a `String` inside a method, you will need to return the new modified string.

Finally, you should note that since Java always passes by value your actual parameters can be constants. For example, the following statement will find the area of a triangle with a base of 14 and a height of 18.3:

```
triangleArea = areaOfTriangle (14, 18.3);
```


4.6 Chapter Summary

Keywords/phrases introduced in this Chapter. You should be able to explain each of the following terms:

actual parameter	local variables	pass by value
argument	method	passing parameters
class method	method body	postcondition
constructor	method heading	precondition
formal parameter	modifiers	procedure
function	overloading	static methods
instance	parameter	subprograms
instance methods		

Java key words/syntax used in this chapter

@param	public
@return	return
@throws	static
IllegalArgumentException	throw
private	void

4.7 Questions, Exercises and Problems

- 1) In Java we call subprograms methods. What other names are used for subprograms in languages such as Turing or C++?
- 2) What is a constructor? Explain when you would use a constructor.
- 3) What is the difference between a static method and an instance method? Explain your answer, using an example.
- 4) How do we send information into a method from the calling program? How does the method send information back to the calling program?
- 5) We have already discussed the modifiers `public`, `private` and `static`. Use the Internet to find at least three other modifiers. Explain what each of these modifiers means.
- 6) What is the difference between a precondition and a postcondition?
- 7) What is the difference between an actual parameter and a formal parameter? What is an argument?
- 8) What is the main difference between primitive parameters and object parameters? Explain your answer.

9) What is method overloading? What are the advantages of overloading?

10) The headings for 5 different versions of overload are given below.

Version 1: **static void** overload (**double** first, **char** second, **int** third)

Version 2: **static void** overload (**char** first, **int** second, **double** third)

Version 3: **static void** overload (**char** first, **int** second)

Version 4: **static void** overload (**double** first, **char** second)

Version 5: **static void** overload (**int** first)

You are also given the following variable declarations in the main program

```
int aVar;  
double bVar;  
char cVar;
```

In each of the following calls of overload which version will be called? In some cases there may be no match.

- | | |
|--------------------------------|--------------------------------|
| a) overload (aVar) | b) overload () |
| c) overload (bVar, cVar, aVar) | d) overload (aVar, bVar, cVar) |
| e) overload (cVar, aVar) | f) overload (bVar, cVar) |
| g) overload (bVar) | h) overload (aVar, cVar) |

11) List and explain 4 advantages of writing your own Java methods.

12) Do you think there are any disadvantages of using methods in your Java programs? Explain your answer.

13) For each of the following, complete the method heading. The heading includes any modifiers, the return type, the name of the method, and the list of formal parameters (type and name). You should include any parameters you feel are necessary to make your method as flexible as possible. (DO NOT write the code for the body of each method).

a) A method called `dayNumber()` that finds the day number for a given date. The day number will range from 1 to 365 in most years and from 1 to 366 in leap years. For example, the day number for January 5, 2004 would be 5.

b) A method called `drawRectangle()` that draws a rectangle in a window.
Note: To indicate which window you want to draw the rectangle in, you should include a `Graphics` object `g` as one of the parameters. We will be learning more about `Graphics` objects later in the course, when we look at graphical user interfaces.

c) A method called `searchAndReplace()` that creates a new string by searching a string (`originalStr`) and replacing all occurrences of a second string (`searchStr`) with a third string (`replaceStr`).

d) A method called `tempInCelsius()` that converts a temperature in degrees Fahrenheit to degrees Celsius.

14) Predict the output of the following Java program. You should also show a memory trace for all local variables including any parameters. Finally, identify specific examples of an overloaded method, an actual parameter and a formal parameter in this program. Answer in your notebook. Note: Comments have been omitted to save space

```
public class TraceProgram
{
    static double average (double first, double second)
    {
        double result = (first + second)/2;
        return result;
    }

    static int average (int first, int second)
    {
        return (first + second)/2;
    }

    static double abs (double number)
    {
        if (number > 0)
            return number;
        else
            return -number;
    }

    static void printChar (char ch, int times)
    {
        for (int count = 1; count <= times; count++)
            System.out.print(ch);

        System.out.println();
    }

    static void printChar (int times)
    {
        printChar ('*', times);
    }

    public static void main (String [] args)
    {
        System.out.println (average(5, 10));
        System.out.println (average (3.5, 5.5));
        System.out.println (average (7.0, 6));
        System.out.println (abs(-3));
        System.out.println (abs(4.5));
        printChar ('+', 3);
        printChar (5);
    } // main method
} // TraceProgram class
```

Write the Java code for the following methods. To test each method you will also need a test class with a main method. Make sure you test your methods with a variety of test cases. Also, don't forget to include complete comments for each method including any @param and @return comments.

15) A method called `sumOfNumbersBetween()` that finds the sum of the numbers between two integer numbers including the two numbers. For example, `sumOfNumbersBetween(4, 7)` should return 22 (4+5+6+7).

16) A method called `distanceBetween()` that finds the distance between two points. This method should have 4 parameters: the x and y value of the first point and the x and y value of the second point. For example if you wanted to find the distance between A (1, 2) and B (4, 6), you would use `distanceBetween(1, 2, 4, 6)`. In this case the method should return 5. In case you forgot, the formula for the distance (d) between two points (x_1, y_1) and (x_2, y_2) is given below:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

17) For a positive integer n, the notation n! (read as 'n factorial') is used to describe the product $n \times (n-1) \times (n-2) \dots \times 2 \times 1$. For example $4! = 4 \times 3 \times 2 \times 1 = 24$ and $10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3,628,800$. Also, by definition $0! = 1$. Write the code for a method called `factorial()` that finds the factorial of a positive integer. For example, the following would set answer to 120:

```
answer = factorial(5);
```

As you can see, factorials can get very large so be careful when deciding on the return type for your method and when testing your method. If a negative value is sent into `factorial`, you should throw an `IllegalArgumentException`.

18) A method called `highestCommonFactor()` that finds the highest common factor of two integer numbers. The following table shows some example results:

First Number	Second Number	Highest Common Factor
25	10	5
17	23	1
13	312	13
36	-66	6
0	15	15

You should make sure that your method can handle all possible test cases. In particular, make sure you deal with the cases where one or both of the parameters are negative or zero.

19) A prime number is a positive integer that is divisible only by itself and 1. The number 1 is not a prime number. Write a method called `isPrime()` that checks to see if an integer number is a prime number. This method should return either true or false.

Hint: You can use the modulo operator (%) to check if one number is a factor of another number. Try to make your code as efficient as possible by minimizing the number of comparisons and calculations.

To test your method, write a program to find all of the prime numbers less than 100. Once your program is working correctly, try using your program to count the number of prime numbers less than 1 million. If you think your code is very efficient, try finding the number of primes less than 5 million or even 10 million. Since displaying the prime numbers slows down the process, you should comment out the print statement when counting the number of primes for these larger numbers.

20) Write a method called `mySqrt()` that finds the square root of a double number without using any methods in the `Math` class. You should plan this out on paper before writing the Java code. You can use `Math.sqrt()` to test your code. If the value sent to `mySqrt()` does not have a square root you can return `Double.NaN` (`NaN` is short for "Not a Number").

21) The number 31193 is a prime number with an interesting property. If you remove the last digit from this number you get 3119, which is also a prime number. In fact all of the numbers generated by removing the last digit from the previous number are prime numbers. In this case the prime numbers are 311, 31 and 3. Write a Java program (see also question 19) to find the largest prime number with this property. Note: The answer is less than `Integer.MAX_VALUE`, the maximum integer value

Chapter 5 - Working with Strings

Strings are objects that contain one or more characters. We can use strings to keep track of non-numerical data such as names, addresses and phrases. In Java, String objects are immutable which means that the data stored in a String object cannot be changed. However, since a String variable is just a reference to a String object we can change this variable to refer to a new String object. Because we have to create a new object every time we change a string, in many cases using a String object is not very efficient so you may want to consider using a `StringBuffer` (`StringBuilder` in Java 5.0) or a character array if you are frequently changing the contents of a string.

5.1 Creating and Initializing a New String Object

If we want to keep track of a String in our Java programs, our first step is to create a variable to refer to our string object. For example:

```
String name;  
String address;
```

Unlike primitive types, the above statements do not allocate space for the String objects; they just create references to strings. To create the space for the String object we can use the `new` command. For example:

```
name = new String("Jim Nasium");  
address = new String("123 Main Street");
```

Since we use strings a lot in Java, the language allows us to initialize strings without using the `new` command as is shown in the following examples.

```
name = "Jim Nasium";  
address = "123 Main Street";
```

Of course, as we have already seen, we can combine the declaration and initialization of a String object as follows:

```
String name = "Jim Nasium";  
String address = "123 Main Street";
```

Since using the `new` command will actually create 2 String objects, the last method shown above is the preferred way of creating strings in Java.

5.2 Comparing Strings

Since String variables are references to strings, you have to be careful when comparing strings. If you want to compare the contents of two strings, you must use one of the following special methods. **DO NOT** use `==`, `>`, `<`, `>=` or `<=` to compare two strings since these operators will compare the values of the String references instead of the contents of the String objects.

boolean `equals(String anotherString)` - checks to see if two String objects are equal returning `true` or `false`.

boolean `equalsIgnoreCase(String anotherString)` - same as `equals` but ignores case (i.e. "abc" is the same as "ABC" or "AbC")

int `compareTo(String anotherString)` - compares two strings based on the Unicode value of each character in the string. The return value is based on the first mismatched characters. Note: For Strings "Z" is greater than "A".

Returns value = 0 - if the two strings are equal
 value < 0 - if the string is "less than" anotherString
 value > 0 - if the string is "greater than" anotherString

For example if we used the statement: `"swing".compareTo("swallow")` to compare "swing" to "swallow", the return value would be 8 since the first mismatched characters are 'i' and 'a' and 'i' is the 9th letter in the alphabet and 'a' is the 1st letter in the alphabet. The value is greater than zero since "swing" is alphabetically greater than "swallow". If we had switched the order of the above Strings and compared "swallow" to "swing" the result would have been -8.

Code examples:

```
System.out.print("Please enter the first String: ");
String firstStr = System.out.readLine();
System.out.print("Please enter the second String: ");
String secondStr = System.out.readLine();

// To compare firstStr and secondStr we use the following:
if ( firstStr.equals(secondStr))
    System.out.println("The two Strings are equal");
else
    System.out.println("The two Strings are not equal");

if (firstStr.compareTo(secondStr) > 0)
    System.out.println("The first string is greater than the second");

// To compare a string to a constant value
if (firstStr.equalsIgnoreCase("Yes"))
    System.out.println("You entered 'yes', 'YES', 'Yes' etc. ");
```

5.3 Looking at Strings, One Character at a Time

In some cases you may want to treat a `String` like an array of characters. The `length()` method can be used to find out how many characters are in the `String` and the `charAt()` can be used to access individual characters within a `String`.

int `length()` - returns the number of characters in the string. Note: unlike the length of an array, the length of a `String` is a method so don't forget to include the `()` at the end.

Example:

```
String sentence = "This is a sample string";
System.out.println(sentence.length())    // would output 23
```

char `charAt(int indexOfChar)` - returns the specified character in the `String`.

Note: the index of the first character is 0 and the index of the last character is `length()-1`.

Examples:

```
String sentence = "ABCDEFGH";
System.out.println(sentence.charAt(3))    // would output d
System.out.println(sentence.charAt(10))// would throw an exception
```

When the index given to the `charAt()` method is negative or not less than the length of the `String` a `StringIndexOutOfBoundsException` is thrown.

We can combine the above two methods to solve certain problems. For example, if you wanted to count the number of spaces in a `String` you could use the following section of code:

```
String sentence = "Count the number of spaces in this String";
int spaceCount = 0;
for (int index = 0; index < sentence.length(); index++)
{
    if (sentence.charAt(index) == ' ')
        spaceCount++;
}
System.out.printf("There are %d spaces in \"%s\"%n",
                 spaceCount, sentence);
```

In the above example, this section of code would output:

```
There are 7 spaces in "Count the number of spaces in this
String"
```

Notice the use of `\"` to get a quotation mark inside the format string.

5.4 Creating Substrings

The `substring()` method is used to get a substring from a larger string. In each case a new `String` is created and returned. There are two versions, one with two parameters and one with one parameter.

`String substring(int startIndex, int endIndex)` - returns a substring starting with the character at `startIndex` and including all characters up to but not including the character at the `endIndex`. NOTE: Since the character at `endIndex` is not included, the last character in the substring is the character at `endIndex-1`.

`String substring(int startIndex)` - returns a substring starting with the character at `startIndex` and including all characters to the end of the `String`.

Examples:

```
String str = "abcdefgh";
String newStr;
newStr = str.substring (3);      // sets newStr to "defgh"
newStr = str.substring (1,3);    // sets newStr to "bc"
newStr = str.substring (0,3);    // sets newStr to "abc"
newStr = str.substring (1,4);    // sets newStr to "bcd"

// to remove "def" from str to get "abcgh"
newStr = str.substring(0,3) + str.substring(6);
```

5.5 Searching Strings

The methods `indexOf()` and `lastIndexOf()` are used to search a string for a particular character or string. Since string indexes start at 0, each of these methods returns an index within the string between 0 and `length()-1`. In each case, if the item being searched for is not found, `-1` is returned. When searching for strings, the index of the first character that matches the search string is returned.

The main difference between these two methods is that `indexOf()` searches forward through the string looking for the first occurrence of the string or character you are looking for and `lastIndexOf()` searches backward through the string looking for the last occurrence of the string or character you are looking for.

When calling these methods you need to give at least one parameter to indicate what you are looking for. As was mentioned this could be either a string or a character. You can also add a second optional parameter that specifies the index that you want to start searching from. If this second parameter is not given, `indexOf()` starts searching from the start of the string and `lastIndexOf()` starts searching from the end of the string. In summary, the headings for each of the different versions of these methods are given below:

```
int indexOf(char ch)
int indexOf(String str)

int indexOf(char ch, int fromIndex)
int indexOf(String str, int fromIndex)

int lastIndexOf(char ch)
int lastIndexOf(String str)

int lastIndexOf(char ch, int fromIndex)
int lastIndexOf(String str, int fromIndex)
```

Examples:

```
//              1          2
//              012345678901234567890123456
String fruits = "apple orange pineapple kiwi";
int findPos;

findPos = fruits.indexOf('p');           // would return 1
findPos = fruits.indexOf('p', 5);       // would return 13
findPos = fruits.lastIndexOf(' ');     // would return 22
findPos = fruits.lastIndexOf(' ', 21);  // would return 12

findPos = fruits.indexOf("apple");      // would return 0
findPos = fruits.indexOf("apple", 3);   // would return 17
findPos = fruits.lastIndexOf("pp");    // would return 18
findPos = fruits.lastIndexOf("pp", 14); // would return 1

findPos = fruits.indexOf("banana");     // would return -1
findPos = fruits.indexOf("orange", 12); // would return -1
findPos = fruits.lastIndexOf("peach");  // would return -1
findPos = fruits.lastIndexOf("kiwi", 12); // would return -1
```

5.6 Other String Methods

The following methods are used to change a `String` to upper or lower case or to trim extra blanks from the beginning or end of a `String`. Since `Strings` are immutable, these methods do not change the original `String`. Instead they return a modified `String` object. Therefore, if you wanted to change a person's name to upper case you must re-assign your string reference (name) to the new returned uppercase string. For example:

```
name.toUpperCase();           // Will not change name
name = name.toUpperCase();    // Must set name to new String
```

`String toLowerCase()` - returns a new string with all characters converted to lowercase.

`String toUpperCase()` - returns a new string with all characters converted to uppercase.

`String trim()` - returns a new string with all of the leading and trailing white space (spaces, tabs, returns and newline characters) removed.

Examples:

```
String str = "This IS a MiXeD case StRing";

// To output: THIS IS A MIXED CASE STRING
System.out.println(str.toUpperCase());

// To output: this is a mixed case string
System.out.println(str.toLowerCase());

String str = "    Extra spaces    ";
// To remove the extra spaces from the front and the
// end of str and then store the result back in str
str = str.trim()
```

For a list of other `String` methods, check out the Java API help.

5.7 Converting Other Types to and from Strings

Sometimes we need to convert a non-`String` object or primitive variable to a `String`. If we "+" (concatenate) a string with a primitive type (or most objects), the primitive type (or object) is converted to a string before joining the two values together. For example, if we run the following Java code:

```
int number = 1234;
String str = "The number is: " + number;
```

`number` will be converted to a string before joining it with "The number is: " creating the new string "The number is: 1234". For more options and control of the resulting `String`, you can also use `String.format` introduced in Section 2.4. For example, the second line above could be changed to:

```
String str = String.format("The number is: %d", number);
```

If we want to convert a primitive type to a string, without concatenating it to another string or using `String.format()`, we can use the `valueOf()` method. Like methods in the `Math` class such as `Math.sqrt()`, `valueOf()` is a static method in the `String` class. Therefore, to call this method we use `String.valueOf()`. For example:

```
int intNumb = 345;
double realNumb = 4.5;

// to convert 345 to "345"
String intStr = String.valueOf(intNumb);

// To convert 4.5 to "4.5"
String dblStr = String.valueOf(realNumb);
```

To convert non-primitive variables to strings we can use the `toString()` method available for most objects. For example, to convert a `Date` to a `String` we can use the following:

```
Date currentDate = new Date();
String dateAsString = currentDate.toString();
```

Since `System.out.println()` can only print strings, the `toString()` method is automatically called when you use this method to print an object.

Also, when you concatenate (+) any object and a `String` object, the `toString()` method for the object is called before concatenating. Since the `toString()` method is a very useful method, you should always try to include this method as part of any new class.

As you can see all primitive types and most objects can be converted to a string. However, to convert a `String` to one of the primitive types is not always possible. For example, since `char` can only hold 1 character there is no method to convert a `String` to a `char`. To convert a `String` to an `int` we use the `parseInt()` method in the `Integer` class. For example:

```
intVar = Integer.parseInt(stringVar);
```

will convert the variable `stringVar` into an integer and store the result in `intVar`. Similarly, to convert a `String` to a double we can use the `parseDouble()` method in the `Double` class. For example:

```
doubleVar = Double.parseDouble(stringVar);
```

Both of these methods will throw a `NumberFormatException` if the `String` parameter can't be converted to a number (e. g. "12B" or "123.45.67").

For non-primitive types, you should check each object's API help documentation. Some objects include a constructor that allows you to create a new object from a `String`.

5.8 Using Scanners to break up a String

Back in Chapter 2 we learned that we can use `Scanners` to read in data from the keyboard. `Scanners` can also be used to read data from a `String` when it is given as a parameter to the `Scanner` constructor. Below is an example of a `Scanner` with a `String` parameter. Since we are scanning for integers, we use `hasNextInt` in our check for more numbers (tokens).

```
Scanner data = new Scanner("1 2 3 4 5");
int total = 0;
while (data.hasNextInt())
    total += data.nextInt();

System.out.println(total);    // Should output 15
```

The default delimiter (character that separates each token in the `String`) is a whitespace (space, tab or newline character). We can change this delimiter with the `useDelimiter` method.

Just like a keyboard `Scanner`, all of the standard methods such as `next` (reads the next available `String`), `nextInt`, `nextDouble` and `nextLong` are available. Since we need to know when to stop reading the data, we also usually use the `hasNext`, `hasNextInt`, `hasNextDouble` and `hasNextLong` methods to check if the next token to be read matches what you are looking for. Since `hasNext` looks for a `String`, it will also match an `int`, `double` and `long` as well. The methods `nextLine` and `hasNextLine` are also available but are not usually used when reading `Strings`.

Here is another example using the `useDelimiter` option.

```
Scanner data = new Scanner("one, two, three, four");
// Use , and space as delimiters (put between [ ])
// The + means to treat multiple delimiters as one
data.useDelimiter("[, ]+");

while (data.hasNext())
    System.out.println(data.next());
```

Should output:

```
one
two
three
four
```

Another faster way to break up a `String` is a `StringTokenizer` that will be covered in the next section.

5.9 The StringTokenizer Class

The `StringTokenizer` class provides a powerful tool that can be used to break a string into smaller substrings or "tokens". A token is a sequence of characters that go together. For example, the words in a sentence or the numbers in a mathematical equation could be considered tokens. Special characters called "delimiters" separate these tokens. For example, in a sentence, the delimiters may be white space and punctuation. When creating a new `StringTokenizer` object, you can specify the delimiter characters as well as whether or not these delimiters are to be included as tokens.

The following constructors can be used to create a `StringTokenizer` depending on the results you want.

```
StringTokenizer(String str, String delim, boolean includeDelim)
```

Constructs a `StringTokenizer` using the `String` `str`. The characters in `delim` are the delimiters used to separate the tokens. If the `includeDelim` flag is true, then the delimiter characters are also included as `String` tokens. If the flag is false, the delimiter characters are not included as tokens.

```
Example: StringTokenizer phrase = new
         StringTokenizer("One, Two, Three, Four!",
                        "\n\t\r,!?.", true);
```

would break down the string into the following tokens (␣ indicates a space):

One	,	␣Two	,	␣Three	,	␣Four	!
-----	---	------	---	--------	---	-------	---

```
StringTokenizer(String str, String delim)
```

This constructor is equivalent to: `StringTokenizer(str, delim, false)`
The delimiter characters are not included as tokens.

```
Example: StringTokenizer phrase = new
         StringTokenizer("One, Two, Three, Four!", " \n\t\r,!?.");
```

would break down the string into the following tokens (in this case the spaces are not included since we added a space to the list of delimiters):

One	Two	Three	Four
-----	-----	-------	------

```
StringTokenizer(String str)
```

This constructor is equivalent to: `StringTokenizer(str, " \t\n\r")`
It uses the default delimiters shown which include the space, tab, newline and return characters. The delimiter characters are not included as tokens.

Example: `StringTokenizer phrase`
`= new StringTokenizer("One, Two, Three, Four!")`

would break down the string into the following tokens:

One,	Two,	Three,	Four!
------	------	--------	-------

Once you have created a `StringTokenizer` object, you can retrieve the tokens from this object by calling the `nextToken()` method. The first time you call this method it will return the first token. Each subsequent call returns the next token in the list. If there are no tokens available, `nextToken()` will throw a `NoSuchElementException`. To avoid these exceptions we can call the boolean method `hasMoreTokens()` to check if there are any more tokens left. Combining these two methods we can look at all of the tokens in a `StringTokenizer` using a simple loop.

Example: `StringTokenizer phrase`
`= new StringTokenizer("Words in a String");`

```
while (phrase.hasMoreTokens())
{
    System.out.println(phrase.nextToken());
}
```

would output the following:

```
Words
in
a
String
```

To find out the number of tokens available from a `StringTokenizer` without going through all of the tokens we can call the `countTokens()` method.

Example: `StringTokenizer phrase`
`= new StringTokenizer("Count the words in here");`

```
System.out.println("There are " + phrase.countTokens()
+
    " words in the string");
```

would output the following:

```
There are 5 words in the string
```

Another good use for the `StringTokenizer` class is to parse a mathematical equation. For example, the following statements would parse an equation:

```
String equation = "387 + 4.7 * 23.4";
StringTokenizer parsedEquation = new
    StringTokenizer(equation, "+-/*()", true);
```

With each call of `parsedEquation.nextToken()` you would get the next number or operator in the equation. For example, the following section of code would produce the output shown: (Note: the `trim()` is included to trim off the extra spaces included with each number, since space is not one of the delimiters.)

```
while (parsedEquation.hasMoreTokens())
    System.out.println(parsedEquation.nextToken().trim());
```

Output:

```
387
+
4.7
*
23.4
```

As you can see `StringTokenizers` can be a very powerful tool if their use is appropriate to the situation and they are used properly.

5.10 Chapter Summary

Keywords/phrases introduced in this Chapter. You should be able to explain each of the following terms:

concatenate
delimiter
exceptions

immutable
index

substring
token

Methods introduced in this chapter:

String methods

<code>charAt()</code>	<code>String()</code> - constructor
<code>compareTo()</code>	<code>substring()</code>
<code>equals()</code>	<code>toLowerCase()</code>
<code>equalsIgnoreCase()</code>	<code>toUpperCase()</code>
<code>indexOf()</code>	<code>trim()</code>
<code>lastIndexOf()</code>	<code>format()</code> - static
<code>length()</code>	<code>valueOf()</code> - static

Integer and Double methods

<code>parseInt()</code>	<code>parseDouble()</code>
-------------------------	----------------------------

New Scanner methods

<code>hasNext()</code>	<code>hasNextInt()</code>	<code>hasNextDouble()</code>	<code>hasNextLong()</code>
------------------------	---------------------------	------------------------------	----------------------------

StringTokenizer methods

<code>countTokens()</code>	<code>nextToken()</code>
<code>hasMoreTokens()</code>	<code>StringTokenizer()</code> - constructors

5.11 Questions, Exercises and Problems

- 1) Explain what “immutable” means. Why do you think strings are immutable?
- 2) What is wrong with the following section of code:

```
Scanner keyboard = new Scanner(System.in);
String enteredPassword;
do
{
    System.out.print("Please enter a password: ");
    enteredPassword = keyboard.next();
}
while (enteredPassword != "opensesame");
```

- 3) List and explain some examples of when you would want to use a StringTokenizer.
- 4) Predict the output of the following section of Java code. Answer in your notebook.

```
//              1              2              3              4              5
//              01234567890123456789012345678901234567890123456789012
String firstStr  = "There are many methods available to work with Strings";
String secondStr = "You should understand how each of these methods works";
String thirdStr  = "                Good Luck                ";

System.out.println ("Working with Strings");
System.out.println (firstStr.length ());
System.out.println ("Constant".length ());
System.out.println (thirdStr.length ());
thirdStr = thirdStr.trim ();
System.out.println (thirdStr.length ());

System.out.println (firstStr.charAt (4));
System.out.println (secondStr.charAt (35));

System.out.println (firstStr.substring (36));
System.out.println (firstStr.substring (23, 28));
System.out.println (secondStr.substring (11, 21));
System.out.println (secondStr.substring (secondStr.length () - 10));
String newStr = secondStr.substring (0, 26) +
    firstStr.substring (46) + firstStr.substring (35, 40);
System.out.println (newStr);

System.out.println (firstStr.indexOf ('e'));
System.out.println (firstStr.indexOf ('e', 20));
System.out.println (firstStr.lastIndexOf ('a'));
System.out.println (firstStr.lastIndexOf ('a', 5));
System.out.println (secondStr.indexOf ("th"));
System.out.println (secondStr.lastIndexOf ("th"));
```

5) Given the following section of code:

```

if (firstStr.equals (secondStr))
    System.out.println ("Strings are equal");
else if (firstStr.equalsIgnoreCase (secondStr))
    System.out.println ("Strings are equal but different cases");
else
    System.out.println ("Strings are not equal");

if (firstStr.compareTo (secondStr) > 0)
    System.out.println ("firstStr is greater than secondStr");
else if (firstStr.compareTo (secondStr) < 0)
    System.out.println ("firstStr is less than secondStr");
else
    System.out.println ("firstStr is equal to secondStr");

```

Predict the output of this code, given the following values of firstStr and secondStr:

- | | |
|-----------------------|-----------------------|
| a) firstStr = "TEST"; | b) firstStr = "Test"; |
| secondStr = "TEST"; | secondStr = "TEST"; |
| c) firstStr = "cat"; | d) firstStr = "swim"; |
| secondStr = "dog"; | secondStr = "swan"; |

6) Predict the output of the following section of code. Work through each line step by step

```

String sentence = "aaaaaAAaaaaaaa";
String searchWord = "aaa";
int count = 0;
int findIndex = sentence.indexOf (searchWord);
while (findIndex >= 0)
{
    count++;
    int searchFrom = findIndex + searchWord.length ();
    findIndex = sentence.indexOf (searchWord, searchFrom);
}
System.out.println (count);

```

7) Write a Java program that inputs a string and then outputs the string backwards and in capitals. Hint: use a for loop and the charAt() method. For example, if the input was "Welcome to Java" the output would be:

AVAJ OT EMOCLEW

8) Write a Java program that inputs a name with the given names first followed by the last name and then outputs the name in the form: last name, any given names

Examples:	<u>Name Input</u>	<u>Name Output</u>
	Sue Johnson	Johnson, Sue
	Prince	Prince
	Billy Bob Joe Carson	Carson, Billy Bob Joe

Note: There should no `println` or `nextLine` statements in the following methods. All input and output should occur in the main (calling) method.

9) Write a Java method called `subStringCount()` that counts the number of times a sub-string appears inside another string. The method should have two string parameters: `str` and `subStr` and it should return the number of times that `subStr` appears in `str`. For example, the following should set `noOfTimes` to 3:

```
noOfTimes = subStringCount ("The selfish fisherman caught too
                             many fish!", "fish")
```

10) Write a Java method called `searchAndReplace()` that searches a string (`origStr`) for another string (`searchStr`) and then replaces all occurrences of `searchStr` with a third string (`replaceStr`). Your method should return the new string with all of the replacements made. To test your method, you should also write a small main method. Use the following method heading:

```
private static String searchAndReplace(String origStr,
                                       String searchStr,
                                       String replaceStr)
```

For example, the following code should set `newStr` to "He wore a black shirt and black pants."

```
String sentence = "He wore a blue shirt and blue pants.";
String newStr = searchAndReplace(sentence, "blue", "black");
```

11) Write a method called `sizeOfLargestGroup()` that finds the size of the largest group of consecutive matching characters in a String. This method will have a single String parameter. For example, given "CAACCCDCD" you should return 4 since the largest group of matching characters is the 4 C's in a row. In this case, the 2 other C's in the String don't count because they are not part of the group. Other example calls of this method are shown in the table below:

Method Call	Return Value
<code>sizeOfLargestGroup ("xyyXxxxxx")</code>	5
<code>sizeOfLargestGroup ("AAACCCBBBCCBBBAA")</code>	3

12) Write the code for a method called `lettersInCommon()` that finds all of the letters that are common to two strings. This method will have two String parameters and it should return a String made up of all of letters common to both strings. The string of common letters should be in lower case and should not contain any duplicate letters. If the two strings have no letters in common you should return an empty String. Your program should ignore case (i.e. 'A' and 'a' should be considered the same letter). For example:

```
lettersInCommon("Don't forget to plan", "your solution carefully")
should return "aeflnort". Note: The order of the returned letters could be different.
```

13) In Canada we use a 9-digit social insurance number (SIN) to uniquely identify each person. The last digit in this 9-digit number is known as a check digit that is used to check the validity of a SIN. To find or check the check digit we use the Luhn formula, a formula created in the late 1960's by a group of mathematicians. The Luhn formula is also used to validate other numbers such as credit card numbers. The following steps describe how to use the Luhn formula to check if a SIN is valid. In this example we will be checking the SIN: 375148624

Step 1: Double the value of the even numbered digits of the number beginning with the second digit from the left. For example:

$$7 \times 2 = 14 \quad 1 \times 2 = 2 \quad 8 \times 2 = 16 \quad 2 \times 2 = 4$$

Step 2: Add the individual digits (i.e. add 1+4 instead of 14) of the numbers obtained in Step 1 to each of the odd numbered digits in the original number. For example:

$$1 + 4 + 2 + 1 + 6 + 4 + 3 + 5 + 4 + 6 + 4 = 40$$

Step 3: The total obtained in Step 2 must be a multiple of 10 for the number to be valid SIN.

To find the check digit, given the first 8 digits, you complete steps 1 and 2 and then subtract the resulting total from the next highest multiple of 10. In our example, for the first 8 digits the total would be 36 and then $40 - 36 = 4$ which is the check digit for this number.

a) Write a Java method called `isValidSIN()` that checks if a SIN (stored as a String with no spaces) is valid. The heading for `isValidSIN()` should be:

```
static boolean isValidSIN(String socialInsuranceNumber)
```

b) Write a Java method called `addCheckDigit()` that takes an 8-digit number (stored as a String) and returns a 9-digit number (also stored as a String) with the proper check digit added. The heading for `addCheckDigit()` should be:

```
static String addCheckDigit(String partialSIN)
```

Hint: Since the above methods will share some common code you may want to write a third method that will be used by both of these methods.

14) Write a Java program that inputs a string and then replaces all of the four letter words in the string with four asterisks. For example:

Input: See if you can find all of the four letter words in this sentence.

Output: See if you can **** all of the **** letter words in **** sentence.

Hint: Use a `StringTokenizer` or a `Scanner`

15) The following question is from the Dwrite On-line Programming Contest

Word Arithmetic

Tommy always gets really bored during his English class. However, Tommy loves Math! He spends most of his free time adding two numbers together (it's actually really exciting!). One day he came up with a brilliant idea to both cure his boredom in English class, and to not get caught by his teacher: add words together! The way Tommy adds to words together is exactly how he would add two numbers:

- Every letter represents a number: A = 0, B = 1, ..., Z = 25
- Add the words one letter at a time from right to left
- If the sum of two letters overflows (that is, greater than Z = 25), then the letter to write down will be the sum modulo (the remainder) 26, and the carry will be the quotient you get when you divide the sum by 26 (so essentially same as two numbers in base 26 instead of base 10).

For example, Z + Z = BY (since Z + Z = 50, and 50 modulo 26 is 24, which is Y, and the carry is the quotient you get when you divide 50 by 26, i.e. 1, which represents B).

Note that you don't want the answer to have any leading A (as that's equivalent to having leading 0s).

The input text file **words.txt** will contain 5 test cases. Each case is on a line with two words, **W1** and **W2**, separated by a single space. Every word is at least 1 letter and no more than 254, made up of only upper case letters. You can use a Scanner to read in the text file (see Appendix E.1). To read in each word separately, you will use `next()`.

The output file will contain 5 lines of output. Each a single word representing **W1 + W2**, spelled in upper case. Note: For the Dwrite contest results are output to a text file (see Appendix E.3). However, for this problem, you can output your results to a Console.

Sample Input:

```
CAT DOG
MOM DAD
ABA A
BCD BC
ZZZ ZZZ
```

Sample Output:

```
FOZ
POP
BA
BDF
BZZY
```

Note on 3rd case: words can be of different length. A = 0 so ABA + 0 shouldn't change, but leading 0s are removed. Since 010 = 10, the answer is 10 (that is, BA).

Chapter 6 – Creating New Classes in Java

In Java, in addition to using built in classes such as `Strings`, we can also create our own classes/objects. Just like real world objects, objects in Java have characteristics (data) and behaviour (methods). When we create a new Java class we create a template that defines an object's data and methods. Using the class code, we can create "instances" of a particular class and then manipulate these new objects.

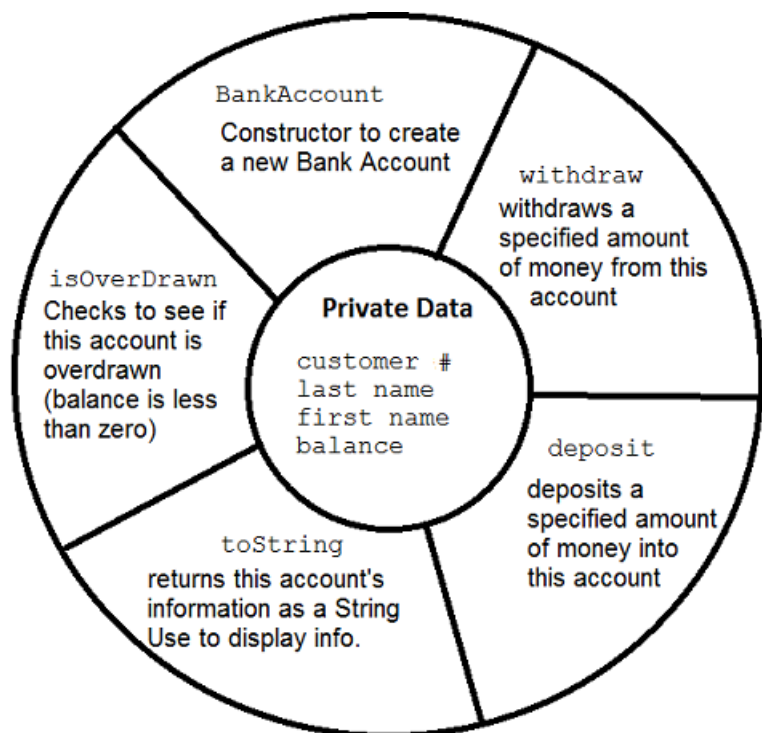
6.1 Encapsulation

As was mentioned, objects have both data and methods. Putting an object's data and methods together in a class is known as "encapsulation". Another important part of encapsulation is that an object should be responsible for its own data. Therefore, in most cases, we make an object's data private and then use public methods to access this private data. This is known as "data-hiding" since the details of how the data is stored in an object are hidden from the user of the object.

When using an object, the user focuses on the behaviour of the object or what they want to do with it. For example, when using a `String` object we can use all of the `String` methods such as `substring`, `charAt` and `length` without worrying about how the individual characters of the `String` are actually being stored. In fact, if the programmer who developed the `String` class code changed how the `String` data was stored, it would not affect how we use this class.

The diagram on the right shows the data and methods for a simple `BankAccount` class. The structure of the diagram highlights the encapsulation concept by showing the data and methods together within the same outer circle or "capsule".

It also shows how the public methods such as `withdraw` and `deposit` on the outer layer protect the private data such as `customer #` and `balance` on the inner layer. If the users of this `BankAccount` object want to look at or change the private data they need to go through the public methods. This helps each object remain responsible for its own data.



6.2 Example: A Car class

In the following example, we will be creating a simple `Car` class. Start up Eclipse and create a new Java project. Call your new project `CarClassDemo`.

Add a new class called `Car` to your project. Do not include a `main` method in this new class. You should get the following code:

```
public class Car
{

}
```

Our first step is to add the data fields (instance variables) to our new class. These fields will keep track of the characteristics of each `Car` object that we create. A `Car` object could have many instance variables but, to keep it simple, we will only be adding in three variables (model, year and speed). Add this code to the `Car` class.

```
private String model;
private int year;
private int currentSpeed;
```

We will keep all of our data fields private so that they can't be directly accessed from outside the class without going through a public method. In simple examples this doesn't always seem necessary but it is an important principle in object oriented programming.

We should also get in the habit of including comments in our code. Each class should have a comment that identifies what the class is used for as well as the author and version. Add the following comments to the top of the `Car` class:

```
/** Keeps track of a Car object's information including
 * the model, year and current speed of the car
 * @author ICS3U
 * @version current date
 */
```

Our next step is to add some methods to our new class. One of the first methods we need for every class is a constructor that will be used to create a new object. A constructor usually defines the initial values of an object's data fields (instance variables). Add the code on the next page to your new the `Car` class. This code should be included after the declarations of the instance variables. Once again, don't forget to include the comments.

```
/** Constructs a new Car object with the given model
 * and year. Current speed is set to zero
 * @param model the model of this Car (e.g. Corvette)
 * @param year the year of this Car (e.g. 2012)
 */
public Car(String model, int year)
{
    this.model = model;
    this.year = year;
    this.currentSpeed = 0;
}
```

You may have noticed that the constructor has the same name as the class. You will also notice that it is public and it has no return value (not even void). The parameters of the constructor are used to pass in the initial values for the data fields.

Note: Inside the constructor code the `this` modifier refers to the instance variables for “this” particular object (the one we just created). For the `model` and the `year`, we need this modifier to distinguish these variables from the parameters of the same name. Since there is no `currentSpeed` parameter, the `this` modifier in front of `currentSpeed` is optional but is included for consistency.

To test this code, we will create another class with a `main` method. Add a new class to your project called `CarTest`. Make sure the new class has a `main` method. In Eclipse you can check off the box to add the main method stub when creating a new class. Add the following code to your main method in the `CarTest` class. Remember, there is no main method in the `Car` class.

```
Car firstCar = new Car("Honda Civic", 2008);
System.out.println(firstCar);
```

Run your `CarTest` program and you should get something like the following: (the number after the `@` sign may be different).

```
Car@30a4effe
```

What about the model, year and current speed. When you display an object using `println`, the object’s `toString` method is called to display the object as a `String`. Since we didn’t define the `toString` method in our `Car` class, the `toString` method for the `Object` class is called instead. By default, all objects extend the `Object` class and therefore we inherited `Object`’s `toString` method. The `toString` method in the `Object` class, returns a string consisting of the name of the class of which the object is an instance and the unsigned hexadecimal representation of the hash code of the object (separated by `@`). Since this doesn’t tell you much about the individual object, you should always override the `toString` method when creating a new class. On the next page is the code required for the `toString` method.

By adding the following `toString` method to our `Car` class we will “override” the `Object` class’ version. The new `toString` method provides more information about the contents of each of our `Car` objects. Here is the method with comments.

```
/** Returns a String representation of this Car
 * @return the Car's year, model and current speed (if moving)
 */
public String toString()
{
    if (currentSpeed == 0)
        return String.format("%d %s stopped", year, model);
    else
        return String.format("%d %s travelling at %d kph",
                               year, model, currentSpeed);
}
```

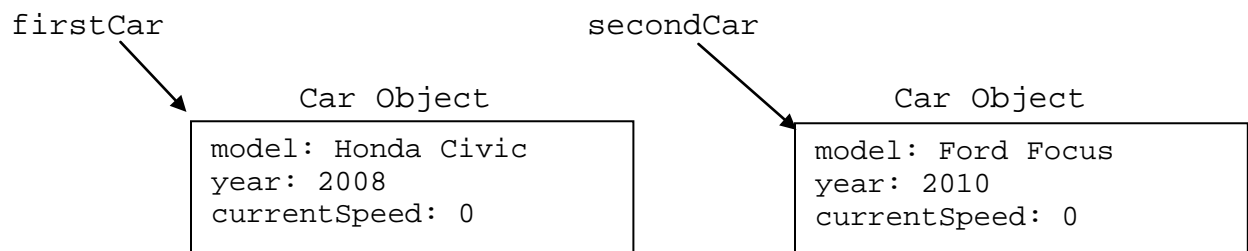
In this method, the `String.format` command creates a new `String` using the same format codes used in `printf`. Before testing your code, let’s add another `Car` to the main program of the `CarTest` class. Add in this code below the `firstCar` code:

```
Car secondCar = new Car("Ford Focus", 2010);
System.out.println(secondCar);
```

When you run your program, the output should be:

```
2008 Honda Civic stopped
2010 Ford Focus stopped
```

In the above examples, we created two instances of the `Car` class (two `Car` objects). Each object has its own model, year and current speed. A memory representation of our two objects is shown below. `firstCar` and `secondCar` are `Car` references that refer to the two `Car` objects shown.



Each `Car` object has 3 instance variables (`model`, `year` and `currentSpeed`). Our next step is to write some methods that can be used to change the instance variables of these `Car` objects. The `accelerate` and `brake` methods given on the next page will be used to change a `Car`’s current speed. Add these methods to your `Car` class code. Put them between the Constructor and the `toString` methods.

```
/** Accelerate the Car (increase its speed by 5 kph)
 */
public void accelerate()
{
    currentSpeed += 5;
}

/** Brake the Car (decrease its speed by 10 kph)
 * Makes sure the speed doesn't fall below 0 kph
 */
public void brake()
{
    if (currentSpeed >= 10)
        currentSpeed -= 10;
    else
        currentSpeed = 0;
}
```

Since both of these methods are instance methods the `static` modifier does not appear in the heading. Now that we are creating our own classes, most of our methods will be instance methods (non-static).

To test these methods, add the following code to the main program in the `CarTest` class. You may have noticed that when we call `Car` methods in the main program, we use the name of the object references (`firstCar` or `secondCar`) to call each instance method. These objects references act like parameters to the methods telling the computer which `Car` we want to accelerate or which `Car` we want to brake.

```
// Accelerate both cars
for (int repeat = 1; repeat <= 7; repeat++)
    firstCar.accelerate();
secondCar.accelerate();

// Print out their status
System.out.println(firstCar);
System.out.println(secondCar);

// Brake both cars
firstCar.brake();
firstCar.brake();
secondCar.brake();
secondCar.brake();

// Print out their new status
System.out.println(firstCar);
System.out.println(secondCar);
```

Before running this code, try to predict the output. The output you should get is shown on the next page.

Running `CarTest` you should get the following output:

```
2008 Honda Civic travelling at 35 kph
2010 Ford Focus travelling at 5 kph
2008 Honda Civic travelling at 15 kph
2010 Ford Focus stopped
```

You may have noticed that even though we called `brake` twice for `secondCar`, the speed didn't fall below zero. This is because the `brake` method has a check to make sure that we don't reduce the speed of the `Car` below 0 when braking.

If we made the `currentSpeed` instance variable `public` instead of `private` we could have changed the speed more directly in the `main` program using the command:

```
firstCar.currentSpeed = 50;
```

At first glance, this may seem like an easier way to change the car's speed. However, since we are trying to create Java objects that model real life objects, instantly setting the speed of a `Car` to 50 kph doesn't make sense.

By using public methods to update an object's private data we can model more realistic behaviour. We can also make sure that an object is responsible for its own data to avoid unrealistic results. For example, most cars have a maximum speed. Since we are using the `accelerate` method to change the speed, by adding a few lines of code to this method we could prevent the `currentSpeed` from exceeding this maximum. If the `currentSpeed` was `public` and could be changed directly, a user of our `Car` object could set the speed to a very large value which could cause problems.

Update the `accelerate` method so that the `currentSpeed` of a `Car` can not exceed a maximum of 150 kph. Look at the `brake` method code for ideas on how to do this. Test your code by modifying the `main` program in `CarTest`. Hint: You can change the upper limit of the `repeat` variable in the `for` loop to a higher value to try to accelerate the car beyond 150 kph.

Can you think of some other instance variables and methods that you may want to add to the `Car` class?

6.3 Another Example: Designing a `Time` object

Assume that we want to create a simple `Time` object that keeps track of the time of things such as Songs in an MP3 player or a Movie on a DVD. You can assume that each `Time` object would keep track of these times to the nearest second.

Our first step in designing a new object is to think about how we would use this object and what behaviour we would need. Try to think of some methods we will need for our new `Time` object.

Like most objects we will need one or more constructors that will be used to create each new `Time` object. One way to construct a `Time` object would be to give the time in hours, minutes and seconds. For example:

```
Time firstTime = new Time (2, 34, 45);
Time secondTime = new Time (6, 23);
```

In the second case we are assuming the `Time` has 0 hours. We could also construct these same `Time` objects using `Strings`. For example:

```
Time firstTime = new Time ("2:34:45");
Time secondTime = new Time ("6:23");
```

By overloading our constructor method (in the above examples we have identified 3 different constructors) we are offering the users of our class more options.

In addition to constructors, most classes should include a `toString` method that will automatically be called when we try to display our objects. For example, the following code:

```
System.out.println(firstTime);
System.out.println(secondTime);
```

should display `2:34:45` and `6:23` respectively. Since we may want to find the total time for all of the Songs we have in our collection or on a play list, we should also create an `add` method to add two time objects. For example:

```
Time totalTime = firstTime.add(secondTime);
```

should add the value of `firstTime` and `secondTime` setting the value of `totalTime` to `2:41:07`. Finally we will include a `compareTo` method that compares two `Time` objects to see which one is bigger. For example:

```
firstTime.compareTo(secondTime);
```

should return a positive value since `firstTime` is greater than `secondTime`.

Now that we have identified some of the methods we need to use with our `Time` object, our next step is to determine how we can store each `Time` objects data. If we only wanted to create and display our `Time` objects we could probably use a `String` to store each `Time` object. However, since we need to add and compare `Time` objects, we will have to come up with another option. Even though keeping track of the hours, minutes and seconds for each `Time` object may simplify some methods such as `toString`, it will make other methods such as `add` and `compareTo` more complicated. Therefore, since the base time unit for all `Time` objects is seconds, we will be using a single integer instance variable to keep track of the total number of seconds in each `Time` object.

Even though we haven't written the code for our `Time` object yet sometimes it is a good idea to write the test code first. Here is some sample test code for the `Time` object. It includes test code for 4 constructors (one is tested 3 times with different data formats), the `toString` method, the `add` method and the `compareTo` method. For each constructor we will assume the given times are valid. This code uses an `ArrayList` to keep track of a list of all of the `Time` objects. Although we haven't covered `ArrayList` or `Collections.sort` yet, hopefully this code will make sense.

```
// Keep track of a list of Time objects
ArrayList<Time> timeList = new ArrayList<Time>();

// Add six different Time objects to the list
// This code will check our constructors
timeList.add(new Time(1, 32, 49));
timeList.add(new Time(4, 27));
timeList.add(new Time(58));
timeList.add(new Time("3:09:03"));
timeList.add(new Time("39:17"));
timeList.add(new Time("35"));

// Display each element in the list (Checks toString)
for (Time next : timeList)
    System.out.println(next);

// Add up all of the Times (checks add)
// Display the running total for a more thorough check
Time totalTime = new Time(0);
for (Time next : timeList)
{
    totalTime = totalTime.add(next); // totalTime = totalTime + next
    System.out.println("Total so far: " + totalTime);
}

// Sort the list of Times (Checks compareTo)
Collections.sort(timeList);

// Display the sorted list
System.out.println("Times in ascending order: ");
for (Time next : timeList)
    System.out.println(next);
```

6.4 Java code for a Time object

Below and on the next page is the Java code (including full comments) for our simple Time object. Look over this code carefully and make sure you understand all of it. You may want to test this code on the computer by entering it and the main program test code from 6.3 into Eclipse. (Note: You can cut and paste the code from the pdf)

```
import java.util.Scanner;

/** Keeps track of a time to the nearest second
 * @author ICS3U
 * @version August 2013
 */
public class Time implements Comparable<Time>
{
    private int seconds;

    /** Constructs a new Time object given the Time in seconds only
     * @param seconds the Time in seconds
     */
    public Time(int seconds)
    {
        this.seconds = seconds;
    }

    /** Constructs a new Time object given the Time in minutes and seconds
     * @param minutes the number of minutes in the new Time
     * @param seconds the number of seconds in the new Time
     */
    public Time(int minutes, int seconds)
    {
        this.seconds = 60 * minutes + seconds;
    }

    /** Constructs a new Time object given the Time in hours, minutes and seconds
     * @param hours the number of hours in the new Time
     * @param minutes the number of minutes in the new Time
     * @param seconds the number of seconds in the new Time
     */
    public Time(int hours, int minutes, int seconds)
    {
        this.seconds = 3600 * hours + 60 * minutes + seconds;
    }

    /** Constructs a new Time object, given the Time as a String
     * Can handle the following formats hh:mm:ss, mm:ss or ss
     * @param timeStr the Time as a String
     */
    public Time(String timeStr)
    {
        Scanner split = new Scanner(timeStr);
        split.useDelimiter(":");
        seconds = 0;
        while (split.hasNextInt())
            seconds = 60 * seconds + split.nextInt();
        split.close();
    }
}
```

```

/** Adds this Time to other Time, returning the total
 * @param other the Time to add to this Time
 * @return the total Time of this Time plus the given other Time
 */
public Time add(Time other)
{
    return new Time(this.seconds + other.seconds);
}

/** Compares this Time to another Time object
 * @param other the Time object to compare to this Time
 * @return a value > 0 if this Time is greater than other,
 *         a value < 0 if this Time is less other and
 *         0 if the Times are the same
 */
public int compareTo(Time other)
{
    return this.seconds - other.seconds;
}

/** Returns this Time as a String
 * @return the Time as a String (h:mm:ss or just m:ss if no hours)
 */
public String toString()
{
    // Find the hours, minutes and seconds equivalent
    int hours = seconds / 3600;
    int minutes = (seconds % 3600) / 60;
    int sec = seconds % 60;

    // Don't show the hours if they are zero
    if (hours > 0)
        return String.format("%d:%02d:%02d", hours, minutes, sec);
    else
        return String.format("%d:%02d", minutes, sec);
}
}

```

You may have noticed that **implements Comparable<Time>** has been added to the heading of the Time class. This identifies the Time class as a Comparable object that can be compared to other Time objects. This code is required since the Collections.sort method used in the main program can only sort Comparable objects. By implementing the Comparable interface our Time class is required to override the abstract compareTo method that will be needed for sorting.

The constructor with the String parameter uses a Scanner to break apart the time string. It should work for Strings with hours:minutes:seconds, minutes:seconds or just seconds. Try it with some examples to help understand how it works.

The **this** modifier used in some of the code is used to refer to the object that was used to call the method. For example, if in the main program we have: `firstTime.add(secondTime)` then in the add method **this.seconds** refers to the seconds instance variable for firstTime and **other.seconds** refers to seconds instance variable for secondTime (the parameter).

6.5 Inheritance

Instead of creating each new class from scratch, object oriented languages such as Java allow us to create subclasses that inherit both characteristics (variables) and behaviour (methods) from another class (superclass). Subclasses can also add new variables and methods. New methods with the same name and parameters will “override” methods in the superclass giving the new subclass specialized behaviour. Inheritance is an effective way of re-using or adding to both existing code and new code.

For example, in Chapter 8 you will be using a `Robot` class that will moves a small mobile robot in a 2D grid. The `Robot` class has a limited number of variables and methods so in order to make a better performing robot we will be creating a `RobotPlus` class. When we create our `RobotPlus` class we use the following heading:

```
public class RobotPlus extends Robot
```

In this case the “extends” means that the new `RobotPlus` class is a subclass of the `Robot` class and therefore inherits all of `Robot`’s data and methods. This means that a `RobotPlus` object can do everything that a `Robot` does. We can also add new data (characteristics) and methods (behaviour) to our `RobotPlus` class to make it do things a `Robot` can’t do. For example, there is no `turnAround` method in the `Robot` class but we can easily add a `turnAround` method to `RobotPlus` class. We can also “override” methods in the `Robot` class to give the `RobotPlus` new behaviour for methods that already exist in the `Robot` class. Finally, we can add new variables to `RobotPlus` so that it can keep track of new things that `Robots` didn’t keep track of (e.g. number of moves made).

Another example of when you will be using inheritance is the final project. To develop nice graphical applications we will be extending the built-in `JFrame` and `JPanel` classes. Since your new classes will inherit both data and methods from the `JFrame` and `JPanel` classes, you can easily create graphical windows with minimal code. You can then add new data and methods to create specific behaviour for your project. Depending on what you are creating for your final project, you may be extending other Java classes as well.

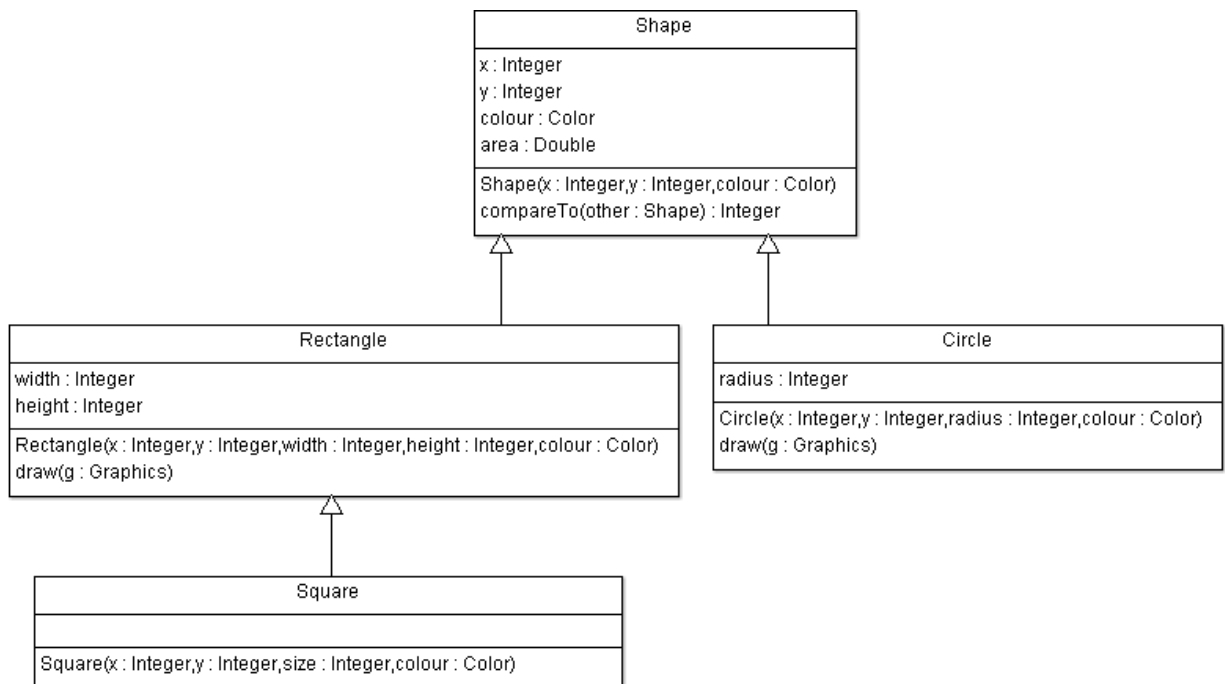
The big advantage of inheritance is that we don’t need to duplicate code that already exists. The other advantage is that by creating a new subclass we leave the original superclass code alone. All of the new features are part of the subclass so if the new code has any bugs it will not affect the original code of the superclass.

In the above examples we will be extending classes that already exist, but in some cases you may want to take advantage of inheritance when designing your own code. By using inheritance you can minimize the amount of duplicate code. You can create superclasses with more general data and behaviour and subclasses with more specific data and behaviour. For example, in a chess game, you could create a `Piece` class that has the data and behaviour common to all chess pieces and then you could create subclasses such as `Pawn`, `Queen`, `Rook` etc. that have more specific behaviour.

To identify if we can use inheritance we look for “is-a” relationships between classes. For example, a `Dog` class could be a subclass of a `Mammal` class because a “Dog is a Mammal”. On the other hand a `Room` class would not be a subclass of `House` class since a `Room` is not a `House`. In this second case, the `House` and the `Room` classes have a “has-a” relationship which means the `House` class could contain a `Room` object as one of its instance variables. This “has-a” relationship is not inheritance.

To help test your understanding of these relationships, try questions 4 to 6 at the end of the Chapter (Section 6.7). Use the “is a” relationship to help identify inheritance.

We can use UML (Universal Modeling Language) class diagrams to show the inheritance relationships between classes. Here is a simplified diagram showing the relationships between the `Shape`, `Square`, `Rectangle` and `Circle` classes. In this case both `Rectangle` and `Circle` extend `Shape` and `Square` extends `Rectangle`.



In this diagram (created by ArgoUML), all shapes have the instance variables `x` and `y` position, `area` and `colour` so these are defined in the superclass `Shape`. `Rectangle` also has `width` and `height` variables and `Circle` has a `radius` variable. `Square` inherits its `width` and `height` variables from `Rectangle`. The behaviour of `compareTo` is defined at the `Shape` level since `Shape` keeps track of the `area` that will be used for comparing. This will allow us to compare different types of `Shapes` by their areas. The `draw` methods are defined in the `Rectangle` and `Circle` classes because you can't draw a shape until you know what kind of shape it is. Since `Squares` are drawn like `Rectangles`, they will inherit their `draw` method from `Rectangle`.

The inheritance tree, or class hierarchy, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the farther down in the hierarchy a class appears, the more specialized will be its behaviour.

When you call a method for a certain type of object it first looks for this method inside its own class. If it can't find the method in its own class it looks for the inherited method from its superclass. So if you ask a `Square` to draw itself it will use the `draw` method in the `Rectangle` class.

6.6 Chapter Summary

Keywords/phrases introduced in this Chapter. You should be able to explain each of the following terms:

class	has-a relationships	object
data hiding	is-a relationships	super modifier
encapsulation	inheritance	this modifier
extends	instance variable	

6.7 Questions, Exercises and Problems

1) What does Encapsulation mean? What are some advantages of Encapsulation?

2) For each of the following classes, suggest at least 3 data fields (instance variables) that you may want to include in each class:

- | | |
|---------------------------------|-------------------------------|
| a) a <code>Student</code> class | b) a <code>Book</code> class |
| c) a <code>Song</code> class | d) a <code>Drink</code> class |

3) When designing a new class, why is it important to think about the methods first?

4) For each of the following indicate whether the “extends” relationship is valid (V) or not valid (NV). **Hint:** Use the “is a” relationship to help identify inheritance relationships.

Giraffe extends Mammal	_____	Baseball extends Sport	_____
Tool extends Saw	_____	Spaghetti extends Pasta	_____
Garage extends House	_____	Time extends Date	_____

5) For each of the following lists of classes, draw a simple inheritance hierarchy diagram (no data or methods required) to show the relationships between the classes. In some cases there will be no connection between the classes.

- | | |
|--|---|
| a) <code>Student</code> , <code>Teacher</code> , <code>Person</code> ,
<code>Educator</code> , <code>Principal</code> ,
<code>PartTimeStudent</code> ,
<code>FullTimeStudent</code> | b) <code>House</code> , <code>Kitchen</code> , <code>Building</code> ,
<code>Room</code> , <code>HighSchool</code> , <code>Classroom</code> ,
<code>School</code> , <code>Desk</code> , <code>Bathroom</code> ,
<code>Neighbourhood</code> , <code>ElementarySchool</code> |
|--|---|

6) Given the classes `Animal`, `Bird` and `Amphibian`, the instance variables: `age`, `weight`, `noOfLegs` and `wingspan`, and the methods: `Animal`, `Bird`, `Amphibian`, `fly`, `grow`, and `metamorphosize`, draw a UML class diagram with both data and methods shown. Parameters for methods are not required.

7) Given the following code for a simple `Television` class. **Note:** To save space comments have not been included.

```
public class Television
{
    private String brand;
    private int size;
    private int volume;
    private int channel;
    private boolean on;

    public Television(String brand, int size)
    {
        this.brand = brand;
        this.size = size;
        this.volume = 10;
        this.channel = 2;
        this.on = false;
    }

    public void turnOn()
    {
        this.on = true;
    }

    public void turnOff()
    {
        this.on = false;
    }

    public void changeChannel(int newChannel)
    {
        if (on)
            channel = newChannel;
    }

    public void turnUpVol()
    {
        if (on && volume < 50)
            volume++;
    }

    public void turnDownVol()
    {
        if (on && volume > 0)
            volume--;
    }

    public String toString()
    {
        if (on)
            return String.format("%s %d\" on channel %d volume: %d",
                                   brand, size, channel, volume);
        else
            return String.format("%s %d\" turned off", brand, size);
    }
}
```

a) Predict the output of the following main program code:

```
Television first = new Television("Sony", 43);
Television second = new Television("Samsung", 54);
System.out.println(first);
System.out.println(second);

first.turnOn();
first.turnUpVol();
second.turnDownVol();
first.changeChannel(34);
second.turnOn();
second.changeChannel(17);

System.out.println(first);
System.out.println(second);

for (int press = 1; press <= 60; press++)
    first.turnUpVol();
for (int press = 1; press <= 20; press++)
    second.turnDownVol();

System.out.println(first);
System.out.println(second);

first.turnOff();
first.changeChannel(21);
second.changeChannel(13);
first.turnOn();
first.turnDownVol();
second.turnUpVol();
System.out.println(first);
System.out.println(second);
```

b) Looking over the `Television` class, what are some other instance variables (characteristics) that we could add to a `Television` class?

c) Looking over the `Television` class, what are some other methods (behaviour) that we could add to a `Television` class?

8) Some object oriented languages such as C++ allow multiple inheritance where a class can extend more than one superclass. In Java, only single inheritance is allowed and a class can only extend one other class. What are some possible problems with multiple inheritance?

9) Given the following code for the `First` and `Second` classes:

```
public class First
{
    private int dataOne;

    public First (int data)
    {
        dataOne = data;
    }

    public void one ()
    {
        System.out.println("A"+dataOne);
    }

    public void two ()
    {
        System.out.println("X");
        this.one();
    }
}

public class Second extends First
{
    private int dataTwo;

    public Second (int data)
    {
        super (data);
        dataTwo = 2 * data;
    }

    public void one ()
    {
        System.out.println("B"+dataTwo);
    }

    public void three ()
    {
        System.out.println ("Y");
        super.one();
    }
}
```

Draw a memory diagram and predict the output of the following section of code. Your memory diagram for the two objects created should look like the diagrams at the bottom of page 82. If any code would cause an error, explain the error.

```
First myFirst = new First (3);
myFirst.two ();
myFirst.three ();

Second mySecond = new Second (5);
mySecond.one ();
mySecond.two ();
mySecond.three ();
```

10) Type in the main program and `Time` class code given in Sections 6.4 and 6.5. Add a `subtract`, `max` and `equals` method to the `Time` class. Here are the headings for each method. Use these headings to help figure out what each method does.

```
public Time subtract(Time other)
public Time max(Time other)
public Boolean equals(Time other)
```

Update the main program to add code to test these new methods.

11) A rational number (fraction) is a number with both a numerator and a denominator such as $1/3$, $2/5$ and 4 (a denominator of 1 is not shown). Design and write the code for a `RationalNumber` class. In addition to a few constructors and a `toString` method, you should include methods to add, subtract, multiple and divide `RationalNumbers`. You should also include code to compare two `RationalNumbers` to see which one is bigger. Hint: You should plan on always storing your `RationalNumbers` in lowest terms. You should also write a main program to test all of the methods of this new `RationalNumber` class.

Chapter 7 - Arrays

7.1 Introduction to Arrays

An array is a collection of one or more adjacent memory cells. An array allows us to store one or more pieces of data of the same type as a group using a single variable name. For example, if we had an array of 10 integers called `numbers`, the array would be represented in memory as follows:

97	56	11	23	34	23	89	34	78	34
----	----	----	----	----	----	----	----	----	----

In this example, `numbers` would refer to the whole array and `number[i]` would refer to an individual element in the array, where `i` is the index or subscript of the element. In Java, array indexes start at zero, so in the above example, `numbers[0]` would be 97 and `numbers[9]` would be 34. If you try to use an index outside of the valid range (0 to 9 in this case), Java throws an `ArrayIndexOutOfBoundsException`.

In Java, all arrays are objects, even arrays of primitive types such as `int` and `double`. Therefore, when using arrays, you need to declare a reference variable for the array and then create the memory spaces for the array using the `new` command. For example to declare and create an array of 50 integers called `listOfNumbers` we would use the following statement:

`int` `[]` `listOfNumbers` `= new int[50];`
↑ ↑ ↑ ↑
type of each 1D name of creates an array
element array array of 50 integers

The first `int` tells the computer that we will be referring to an array of integers. The `[]` tells the computer that the name `listOfNumbers` will be referring to a one-dimensional array. Together, `int [] listOfNumbers` declares the array reference. Then, to actually create the array that the variable `listOfNumbers` will be referring to we use the `new` command followed by the type of variables in the array (in this case `int`) and the number of elements in the array (in square brackets). When first created, each of the elements in an array of `int`, `double` or `char` are all set to zero. So the 50 numbers in the above array would all be zero to start. In a `boolean` array, each element is initially set to `false`.

Arrays in Java are created dynamically so the number of elements in the array could be a variable. For example to create an array to keep track of the names of the students in a class we could use the following:

```
String [] names = new String[noOfStudents];
```

In this case, the size of the array is determined by the variable `noOfStudents`.

Since Strings are objects, the previous statement will create an array of String references. We will still need to create the String objects that this array refers to. When we create an array of object references the initial value of these references is `null`, which means that each element initially refers to nothing. Therefore when working with an array of objects we need to create both the array object and the objects for each element. For example, if we wanted to create an array to keep track of the 3 vertices of a triangle using a Point object for each vertex we would need the following statements:

```
// Create the array object
Point [] triangleVertices = new Point[3];

// Create each Point object of the array
triangleVertices [0] = new Point(2,4);
triangleVertices [1] = new Point(3,5);
triangleVertices [2] = new Point(4,4);
```

In the next section we will see how we can create and initialize an array all in one step.

To find the size of an array, we can use the `length` variable associated with each array. For example `triangleVertices.length` would be 3. You may have noticed that to find the length of an array "`length`" is a variable not a method so the `()` are not required. However, when working with strings "`length`" is a method so the `()` are required.

Since arrays can have many elements, we usually use `for` loops to access all of the elements of an array. For example to print the `names` array created above we could use the following:

```
for (int index=0; index < names.length; index++)
    System.out.println(names [index]);
```

Notice that the value of `index` starts at 0 and that we only continue while the value of `index` is less than the length of the array. Since the array indexes start at 0, the maximum index is always one less than the length of the array (i.e. `0 <= array index <= length-1`).

Arrays can be a very powerful tool in your programming toolbox, but you should be careful to use arrays only when necessary. For example if you were asked to write a program to read in 10 marks and find the average, an array is probably not necessary. However, if you were asked to read in a list of 10 marks and then display a list of all of the marks greater than the average, it would be best to use an array. Hopefully you can see why the array is not necessary in the first case but necessary in the second case. In the exercises at the end of this chapter you will get a chance to practice using arrays in different problems.

7.2 Example Program Using Arrays

To help you understand how to use arrays, the following example program will solve the problem presented at the end of the last section. It will read in a series of marks and then display the marks above the average. To save space only the main program has been presented. Sample output for this program is given on the next page.

```
public static void main (String [] args)
{
    Scanner keyboard = new Scanner (System.in);

    // Find out how many marks are to be entered and set up the array
    System.out.println ("Welcome to the Marks program");
    System.out.print ("How many marks will you be entering?: ");
    int noOfMarks = keyboard.nextInt ();
    int [] marks = new int [noOfMarks];

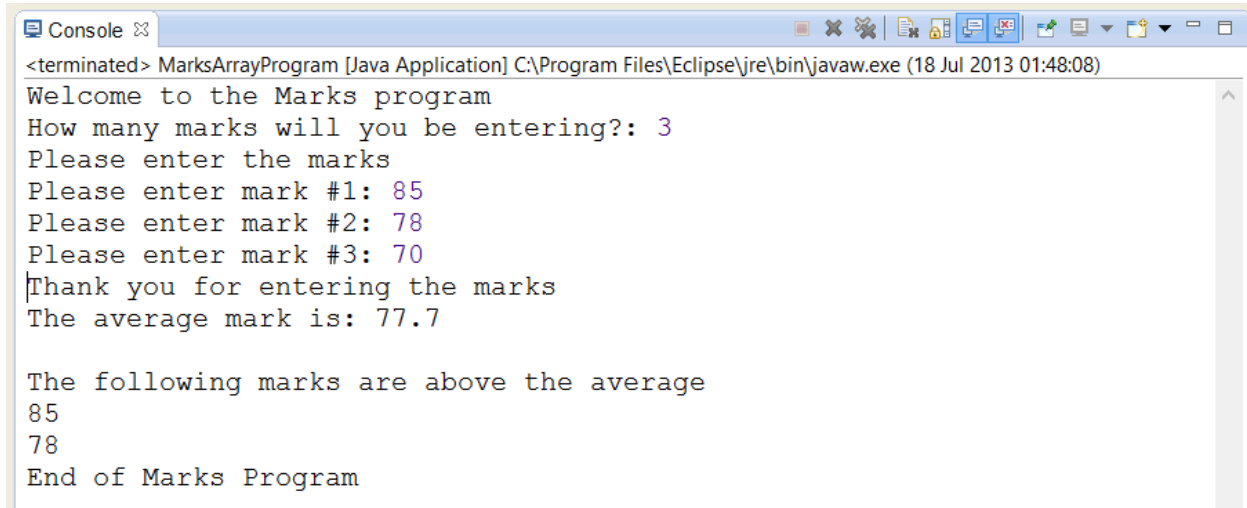
    // Enter the marks, totalling the marks as they are entered
    int totalMark = 0;
    System.out.println ("Please enter the marks");
    for (int markNo = 0 ; markNo < marks.length ; markNo++)
    {
        System.out.printf ("Please enter mark #%d: ", markNo + 1);
        marks [markNo] = keyboard.nextInt ();
        totalMark += marks [markNo];
    }
    System.out.println ("Thank you for entering the marks");

    // Find the average mark and then display a list of all
    // of the marks that are above the average
    double average = (double) totalMark / marks.length;
    System.out.printf ("The average mark is: %.1f\n", average);

    System.out.println ("\nThe following marks are above the average");
    for (int markNo = 0 ; markNo < marks.length ; markNo++)
    {
        if (marks [markNo] > average)
        {
            System.out.println (marks [markNo]);
        }
    }

    System.out.println ("End of Marks Program");
} // main method
```

Sample output for the above program is given on the next page:



```
<terminated> MarksArrayProgram [Java Application] C:\Program Files\Eclipse\jre\bin\javaw.exe (18 Jul 2013 01:48:08)
Welcome to the Marks program
How many marks will you be entering?: 3
Please enter the marks
Please enter mark #1: 85
Please enter mark #2: 78
Please enter mark #3: 70
Thank you for entering the marks
The average mark is: 77.7

The following marks are above the average
85
78
End of Marks Program
```

7.3 Initializing Arrays

If you want to initialize the values in a newly created array, you can save a few lines of code by using an initializer list. For example:

```
int [] listOfNumbers = {16, 23, 7, 15};
```

will create a new integer array with the 4 elements 16, 23, 7 and 15. When using an initializer list we do not need the `new` command to create the array. In the above example, the number of elements listed between the curly braces will determine the size of the new array.

Initializer lists can also be used to create arrays of objects such as Strings. For example:

```
String [] dayNames = {"Monday", "Tuesday", "Wednesday",  
                      "Thursday", "Friday", "Saturday", "Sunday"};
```

For other objects (other than Strings) you will need to use the `new` command to create these objects in the initializer list. For example to initialize the `triangleVertices` array shown in section 7.1 using an initializer list we would use the following:

```
// Create the array object and set the initial values
Point [] triangleVertices = {new Point(2,4),  
                             new Point(3,5),  
                             new Point(4,4)};
```

In some cases you may want to re-initialize an array later in the program. In these cases we can create a new anonymous array object and then change our array reference so that it refers to this new array object. For example:

```
dayNames = new String [] {"Lundi", "Mardi", "Mercredi",  
                           "Jeudi", "Vendredi", "Samedi", "Dimanche"};
```

In this case, if no reference is referring to the original array, the memory where the original array was stored will be cleaned up during garbage collection.

7.4 Related (Parallel) Arrays

In the example presented in section 7.2 we entered a list of marks and then displayed the marks above the average mark. If we wanted to display the names of the students with marks above the average, we would need two arrays, one to keep track of the students' names and one to keep track of their marks. For example:

```
String [] names = new String [noOfStudents];
int [] marks = new int [noOfStudents];
```

When setting up these two arrays, it would make sense to have the marks array and the names array related so that the name and mark for each student would have the same index in each array. For example, if `names[3]` was "Wolf, Jason" then `marks[3]` would be the mark for Jason Wolf.

When we set up more than one array with related corresponding elements these are called related or parallel arrays. Using related arrays can be a very useful tool for many problems. When using related arrays we need to be careful to make sure that we are using the correct array in each case. For example, to display both the name and the mark of the students who got a mark above the class average, we would use the following code:

```
System.out.println ("List of Students Above the Class Average");
System.out.println ("      Name                Mark");
for (int student = 0 ; student < noOfStudents; student++)
    if (marks [student] > classAverage)
    {
        System.out.printf ("%s%3d", names [student], marks [student]);
    }
```

In this example we compare the contents of the `marks` array to the class average to see which students qualify and then we display the contents of both the `names` array and the `marks` array. In all cases we use the same index (`student`). On the next page is an updated complete program from Section 7.2 that contains two related arrays. Again, to save space, only the main program is shown.

```
public static void main (String[] args)
{
    Scanner keyboard = new Scanner (System.in);

    // Find out the number of students and set up the arrays
    System.out.println ("Welcome to the Marks program");
    System.out.print ("Please enter the number of students: ");
    int noOfStudents = keyboard.nextInt();
    String [] names = new String [noOfStudents];
    int [] marks = new int [noOfStudents];

    // Enter the marks and names, totaling the marks as they are entered
    int totalMark = 0;
```

```

System.out.println ("Please enter the names and marks");
for (int student = 0 ; student < noOfStudents; student++)
{
    System.out.print ("Please enter the next student's name: ");
    keyboard.nextLine(); // Needed since last input was nextInt()
    names [student] = keyboard.nextLine();
    System.out.printf ("Please enter %s's mark: ", names [student]);
    marks [student] = keyboard.nextInt();
    totalMark += marks [student];
}

System.out.println ("Thank you for entering the names and marks");

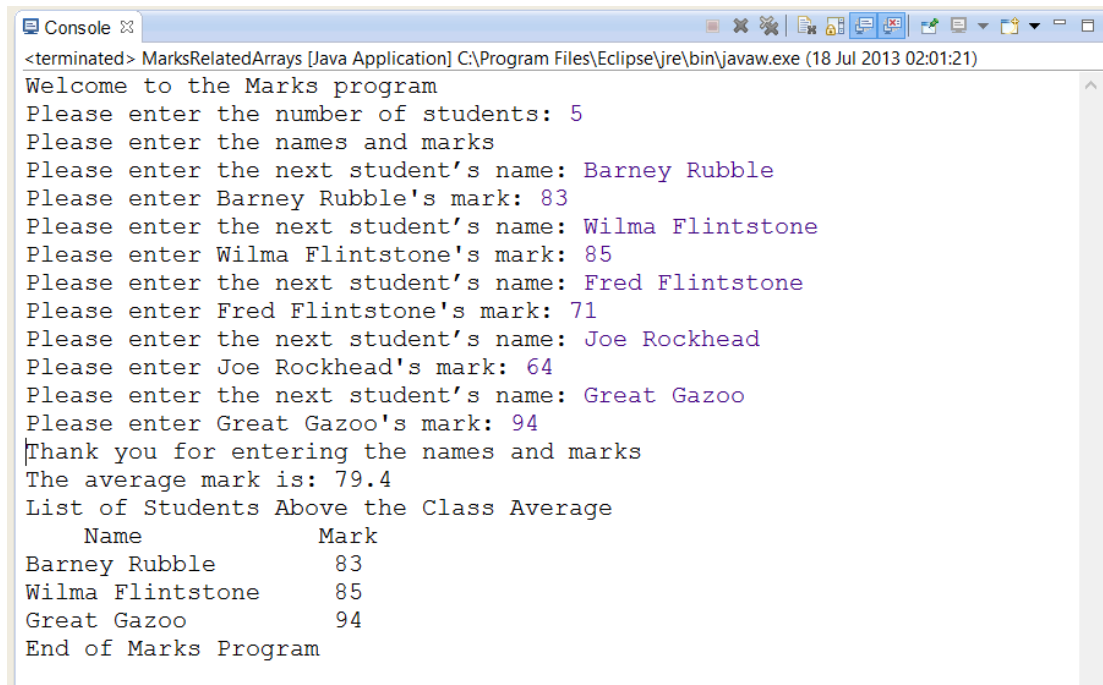
// Find the average mark and then display a list of all
// of the students and their marks who are above the average
double classAverage = (double) totalMark / noOfStudents;
System.out.printf ("The average mark is: %.1f%n", classAverage);

System.out.println ("List of Students Above the Class Average");
System.out.println ("    Name                Mark");
for (int student = 0 ; student < noOfStudents; student++)
    if (marks [student] > classAverage)
    {
        System.out.printf ("%s-20s%3d", names [student], marks [student]);
    }

System.out.println ("End of Marks Program");
} // main method

```

Here is a sample output:



```

<terminated> MarksRelatedArrays [Java Application] C:\Program Files\Eclipse\jre\bin\javaw.exe (18 Jul 2013 02:01:21)
Welcome to the Marks program
Please enter the number of students: 5
Please enter the names and marks
Please enter the next student's name: Barney Rubble
Please enter Barney Rubble's mark: 83
Please enter the next student's name: Wilma Flintstone
Please enter Wilma Flintstone's mark: 85
Please enter the next student's name: Fred Flintstone
Please enter Fred Flintstone's mark: 71
Please enter the next student's name: Joe Rockhead
Please enter Joe Rockhead's mark: 64
Please enter the next student's name: Great Gazoo
Please enter Great Gazoo's mark: 94
Thank you for entering the names and marks
The average mark is: 79.4
List of Students Above the Class Average
    Name                Mark
Barney Rubble           83
Wilma Flintstone        85
Great Gazoo             94
End of Marks Program

```

7.5 Arrays of Objects (An Alternative to Related Arrays)

In the last section we used 2 related arrays to keep track of a student's name and mark. We needed 2 arrays because we needed to keep track of 2 pieces of information for each student. Instead of using 2 arrays we can bundle the student's name and mark into a single object and then have an array of this object. To bundle the student's name and mark into a single object we need to create a new class which defines this new type of object.

Since the data inside this `Student` class is private, we need public methods to use this data. The `isMarkAbove` method lets us check to see if the mark is above the class average and the `toString` method will be used to display the `Student` data in a nice format. Remember the `toString` method will be called when we try to print the `Student` object in the `println` method. Here is the `Student` class code:

```
/** A class to keep track of a Student's name and mark
 * Includes methods to check if the mark is above a given target mark
 */
public class Student
{
    private String name;
    private int mark;

    /** Constructs a new Student object with the given name and mark
     * @param name the name of the new Student
     * @param mark the Student's mark
     */
    public Student(String name, int mark)
    {
        this.name = name;
        this.mark = mark;
    }

    /** Checks to see if this Student's mark is above a given target
     * @param targetMark the mark to check against this mark
     * @return true if this Student's mark is above the target
     *         false if it is not
     */
    public boolean isMarkAbove(double targetMark)
    {
        return mark > targetMark;
    }

    /** Returns a String representation of this Student's info
     * @return the Student's name and mark
     */
    public String toString()
    {
        return String.format("%-20s%3d", name, mark);
    }
}
```

To use this new class we first create a single array of Student objects with the following statement:

```
Student [] listOfStudents = new Student [noOfStudents];
```

The complete code for the main program that uses this class to display the names of students above the average is given below:

```
public static void main(String[] args)
{
    Scanner keyboard = new Scanner(System.in);

    // Find out the number of students and set up the arrays
    System.out.println("Welcome to the Marks program");
    System.out.print("Please enter the number of students: ");
    int noOfStudents = keyboard.nextInt();
    Student[] listOfStudents = new Student[noOfStudents];

    // Enter the marks and names, totalling the marks as they are entered
    int totalMark = 0;
    System.out.println("Please enter the names and marks");
    for (int student = 0; student < noOfStudents; student++)
    {
        System.out.print("Please enter the next student's name: ");
        keyboard.nextLine(); // Needed since last input was nextInt()
        String name = keyboard.nextLine();
        System.out.printf("Please enter %s's mark: ", name);
        int mark = keyboard.nextInt();
        totalMark += mark;

        // Create a new Student object and put them in the list
        listOfStudents[student] = new Student(name, mark);
    }

    System.out.println("Thank you for entering the names and marks");

    // Find the average mark and then display a list of all
    // of the students and their marks who are above the average
    double classAverage = (double) totalMark / noOfStudents;
    System.out.printf("The average mark is: %.1f%n", classAverage);

    System.out.println("List of Students Above the Class Average");
    System.out.println("    Name                Mark");
    for (int student = 0; student < noOfStudents; student++)
        if (listOfStudents[student].isMarkAbove(classAverage))
        {
            System.out.println(listOfStudents[student]); // Calls toString()
        }

    System.out.println("End of Marks Program");
} // main method
```

This program will produce the exact same output as the code in Section 7.4. In this example, whether you use related arrays or a simple class is a personal choice. In more complicated examples where we want to sort the results, using the class can make your code easier to follow.

7.6 Two Dimensional Arrays (Arrays of Arrays)

When solving some problems we sometimes need a two dimensional or multi-dimensional array. For example, in many board games such as Chess, Checkers or Connect Four we could use a two-dimensional array to keep track of the board. To declare and create a two-dimensional array we use the following:

```
int [][] board = new int[6][7];
```

This will create a 2D array with 6 rows and 7 columns (traditionally we refer to the first dimension as the number of rows and the second dimension as the number of columns). Like one-dimensional arrays, the indices start at 0 so in the above example `board[0][0]` refers to the top left element and `board[5][6]` refers to the bottom right element.

If we want to declare, create and initialize a 2D array at the same time we can use the following:

```
int [][] grid = {{3, 6, 7, 8},  
                 {4, 12, 52, 17},  
                 {25, 45, 67, 34}};
```

In this case, `grid` will be a 3 by 4 array with `grid[1][2]` referring to 52.

In Java, two-dimensional arrays are really arrays of arrays so the size of each row doesn't have to be the same. For example, the following array is valid in Java:

```
int [][] arrayOfArrays = {{32, 62, 71, 87, 14, 36},  
                          {49, 12, 52},  
                          {25, 45, 14, 35, 17}};
```

In this case the three rows would have 6, 3 and 5 elements respectively. To find the number of rows in a 2D array or the number of elements in each row we can use the `length` variable. For example, `arrayOfArrays.length` would be 3 since there are 3 rows in the above array. `arrayOfArrays[2].length` would be 5 since there are 5 elements in the third row (index of 2). In a rectangular array all of the rows have the same number of elements. Therefore to find out the number of columns we could look at the length of any row. For example, `grid[0].length` would tell us the number of columns in the `grid` array.

The following example code will initialize all of the elements in a 2D array called `array2D` to zero. It will work for both rectangular arrays and jagged arrays (arrays that have rows with different lengths):

```
for (int row = 0; row < array2D.length; row++)  
    for (int column = 0; column < array2D[row].length; column++)  
    {  
        array2D[row][column] = 0;  
    }
```

Here is another example showing how to find the value and position (row and column) of the largest element in a 2D array called `heights`.

```
int highRow = 0;
int highCol = 0;
for (int row = 0 ; row < heights.length ; row++)
    for (int column = 0 ; column < heights [row].length ; column++)
    {
        if (heights [row] [column] > heights [highRow] [highCol])
        {
            highRow = row;
            highCol = column;
        }
    }

System.out.println ("The highest value is: " + heights [highRow] [highCol]);
System.out.println ("in row " + highRow + " and column " + highCol);
```

Since each row of a 2D array is a 1D array, we can use a 1D array reference to refer to each row of the 2D array. For example, given the following array declaration:

```
String [][] words = {{"cat", "dog", "mouse", "badger"},
                    {"red", "green", "blue", "pink"},
                    {"up", "down", "left", "right"}};
```

Then we can create a 1D array reference to refer to one of the rows in `words`. For example:

```
String [] list = words[1];
```

In this case, since `list` refers to the 2nd row in `words`, `list[2]` would be "blue".

Extending this idea and using the `String` method `toCharArray` we can create a 2D character grid from a 1D array of `Strings`. For example:

```
String [] list = {"ERSYT", "ELBTO", "PHENA", "CBANI", "WZUST"};

char [][] letterGrid = new char[5][5];
for (int row = 0; row < letterGrid.length; row++)
{
    letterGrid [row] = list[row].toCharArray();
}
```

would create a 5 by 5 letter grid like the Boggle grid example in Question 23 at the end of the chapter. We can also use this idea to read in a letter grid file one line at a time.

7.7 Arrays and Methods

When we are using arrays with methods it is important to remember that arrays in Java are objects. Therefore when we pass an array as a parameter to a method we are passing a reference to the array in the calling program. So any changes made to the parameter inside the method will change the original array inside the calling program. Here is an example method:

```
static void doubleTheArray(int [] numbers)
{
    for (int i=0; i < numbers.length; i++)
        numbers [i] = 2* numbers [i];
}
```

with the main program code:

```
int [] listOfNumbers = {3, 6, 88, 12, 16};
doubleTheArray (listOfNumbers);
```

In this example, when we call `doubleTheArray ()`, the local variable: `numbers` is assigned the value of the main program variable: `listOfNumbers`. Since these variables are array references, `numbers` will now refer to the same array as `listOfNumbers`. Therefore the line: `numbers [i] = 2* numbers [i]` will modify the elements in the `listOfNumbers` array in the main program. We are still passing the parameter by value but, since the value being passed is a reference, the result is very similar to what happens when a parameter is passed by reference in languages such as C and Turing.

By passing an array reference into the method, a local copy of the original array is not needed, saving both time and memory. Also since the method knows where the original array is stored in memory it can communicate information back to the calling program by changing the contents of the array. For example, in the above example, the method doubles all of the numbers in the original array.

The main disadvantage of passing "by reference" is that we have to be careful that we don't accidentally change the values in an array parameter. In the above example we may have wanted a method that doubles the values in an array but leaves the original array intact. To leave the original array unchanged we would need to create a new array in the method and then return a reference to this new array back to the calling program.

For example the new method would be:

```
static int [] doubleOfArray(int [] numbers)
{
    int [] doubledArray = new int[numbers.length];
    for (int i=0; i < numbers.length; i++)
        doubledArray [i] = 2* numbers [i];

    return doubledArray;
}
```

with main program code:

```
int [] listOfNumbers = {3, 6, 88, 12, 16};
int [] newListOfNumbers = doubleOfArray (listOfNumbers);
```

In the above example, we created a new array the same size as the original and then stored the double of each element in this new array leaving the original unchanged. Finally we returned a reference to this new array back to the calling program. In this example, we have 4 array references but we only have two arrays since `listOfNumbers` and `numbers` refer to the same original array and `doubledArray` and `newListOfNumbers` refer to the modified array.

It is important to note that when the method is finished the array reference `doubledArray` will no longer be available since it is local to the method. However the array that it refers to will be available in the calling program using the array reference `newListOfNumbers`. This is because before the method ended we returned the `doubledArray` reference back to the calling program to be stored in the `newListOfNumbers` reference. Some of above concepts can seem tricky at first but it is important that you understand array references when using arrays as parameters or when returning array references.

7.8 Chapter Summary

Keywords/phrases introduced in this Chapter. You should be able to explain each of the following terms:

<code>ArrayIndexOutOfBoundsException</code>	initializer list
element	pass "by reference"
index	subscript

Methods and variables introduced in this chapter:

	Array methods and variables
<code>length</code>	
	String methods
<code>toCharArray()</code>	

7.9 Questions, Exercises and Problems

- 1) Give 3 specific examples of why we would want to use arrays in a Java program.
- 2) Write the Java statements required to declare and create each of the following arrays:
 - a) An array of the names of the students on the basketball team.
 - b) An array of the points scored by each of the above players in the first game.
- 3) Write the Java statement to declare and create an array to keep track of the board in a game of Tic Tac Toe.
- 4) Given the following Java code, predict the output of the given statements:

```
boolean [] used = new boolean [8];
```

a) `System.out.println(used [3]);` b) `System.out.println(used [8]);`

- 5) Declare, create and initialize an array called `daysInMonth` that keeps track of the number of days in each of the 12 months of the year.
- 6) Given the following declaration and initialization:

```
int [] numbers = {13, 24, 19, 26, 37, 5, 70, 25}
```

Predict the output for the following sections of Java code. Each section is separate.

- a) `System.out.println(numbers.length)` b) `System.out.println(numbers[4])`
- c)

```
for (int index = 0; index < numbers.length; index ++)  
    if (numbers[index] % 2 == 0)  
        System.out.print (index);
```

- 7) Explain what the following section of Java code does.

```
int [] numbers = {34, 23, 67, 89, 12, 25, 45, 99, 22, 11, 34, 5};  
  
int newSize = 0;  
for (int index = 0; index < numbers.length; index++)  
    if (numbers[index] < 25)  
        newSize++;  
  
int [] newNumbers = new int[newSize];  
  
int newIndex = 0;  
for (int index = 0; index < numbers.length; index++)  
    if (numbers[index] < 25)  
    {  
        newNumbers [newIndex] = numbers[index];  
        newIndex++;  
    }
```

8) Given the following statement to declare and create a two dimensional array of double:

```
double [][] matrix = new double[3][5];
```

and assuming the values shown have been assigned to the array, predict the output of the following sections of code:

2.5	5.0	3.0	7.5	9.5
1.0	2.5	5.0	3.5	6.0
9.0	4.0	8.0	2.3	4.5

a) `System.out.println(matrix[2][3]);`

b) `double total = 0;`
`for (int i = 0; i < 3; i++)`
`total += matrix[1][i];`

`System.out.println(total);`

c) `total = 0;`
`for (int row = 0; row < matrix.length; row++)`
`for (int column = 0; column < matrix[row].length; column++)`
`if ((row + column) % 2 == 0)`
`total += matrix[row][column];`

`System.out.println(total);`

9) Assume you have declared and created an array of integers called `numbers`. You can also assume that the value of each element in this array have already been set. Write the section of Java code required to perform each of the following tasks. Each section of code is separate.

a) Count the number of negative numbers in the `numbers` array.

b) Display all of the elements in `numbers` that are greater than 10.

c) Reverse the order of the elements in the `numbers` array. For example:

Original Array:	34	12	17	67	50	99	18
Modified Array:	18	99	50	67	17	12	34

10) Write a method called `isValidDate()` that checks to see if a particular date (year, month, day) is valid. To simplify your code you should use an array (see question 5). This method should return `true` if the date is valid and `false` otherwise. For example:

`isValidDate(2004,2,29)` would return `true` since 2004 is a leap year
`isValidDate(2006,9,31)` would return `false` since September has only 30 days
`isValidDate(2007,13,6)` would return `false` since there is no 13th month

Note: You may want to write an `isLeapYear()` method to use in this method.

11) Write a Java program that responds to questions inputted by the user. This is a very simple program to practice using an array of Strings. Your answers don't have to be related to the questions asked. For each question, just give a random response that indicates a yes, no or maybe answer. Your program should include at least 10 different and creative answers. The program should quit when the question entered is "Bye". Below is a sample run of this program. User input is shown in bold.

Welcome to the Computer Advisor

Question: **Should I complete this assignment as soon as possible?**

Answer: Most Definitely!

Question: **Should I buy a new computer?**

Answer: It depends on what you think.

Question: **Can I use these answers in my program?**

Answer: No way!!!

Question: **Bye**

Thank you for using the Computer Advisor Program

Call your program ComputerAdvisor.java

12) Write a Java program to keep track of the frequencies for a series of rolls of two fair six-sided dice. The user should choose the number of times the dice should be rolled. As the dice are rolled, you should keep track of the number of 2's, 3's, 4's ... 12's rolled using an array. You may want to write a method called `rollOfTwoDice()` that simulates the total count on the roll of two dice. Be careful that your method matches the results of actually rolling two dice. The results should be summarized in a table showing both the frequency and the percentage for each possible roll. For example:

Dice Roll Simulation											
Number of Rolls - 500											
Roll	2	3	4	5	6	7	8	9	10	11	12
Frequency	x	x	x	x	x	x	x	x	x	x	x
Percentage	x	x	x	x	x	x	x	x	x	x	x

Test your program with different values for the number of rolls (e.g. 500 or 5000). You should check your results carefully. Don't forget to plan out this program before going to the computer. This is actually a very simple program; so don't try to make it more complicated than it is. Call your program: `RollingDice.java`.

13) Shown below is a sample main program that calls a collection of methods that use arrays. Complete this program by writing the code for the following methods:

- | | | |
|---------------------------------|-------------------------------|--------------------------------|
| a) <code>generateArray</code> | b) <code>displayArray</code> | c) <code>averageOfArray</code> |
| d) <code>indexOfSmallest</code> | e) <code>selectionSort</code> | f) <code>mergeArrays</code> |

By looking at the comments, parameters and how each method is used you should be able to determine the requirements for each method. You should plan out the code for each method before going to the computer.

```

public class ArrayMethods
{
    // The code for all of the methods (including comments) goes here
    // Look at the code below to figure out the heading for each method

    public static void main (String [] args)
    {
        // Title and Introduction
        System.out.println ("                          Using Methods with Arrays");

        // Generate an array of 30 doubles between 1 and 100
        double [] firstList = generateArray (30, 1, 100);

        // Display the array in nice columns in the given Console
        System.out.println ("Here are the numbers: ");
        displayArray (firstList);

        // Find and display the average of the numbers in the array
        System.out.printf ("The average of the array numbers is %.2f\n",
                           averageOfArray (firstList));

        // Find and display the index and value of the smallest number
        int index = indexOfSmallest (firstList);
        System.out.print ("\nThe index of the smallest number is: ");
        System.out.println (index);
        System.out.printf ("The smallest number is: %.2f\n",
                           firstList [index]);

        // Sort using a selection sort and then display the array
        selectionSort (firstList);
        System.out.println ("\nHere is the sorted array: ");
        displayArray (firstList);

        // Generate a second array of 25 elements between -100 and 100
        // and sort this second list
        double [] secondList = generateArray (25, -100, 100);
        sortArray (secondList);

        // Merge the two sorted arrays into a single sorted array
        double [] mergedList = mergeArrays(firstList, secondList);

        // Display the merged array
        System.out.println ("Here is the merged array: ");
        displayArray (mergedList);

        // Closing Remarks
        System.out.println ("The Arrays Methods Program is Complete");
    } // main method
} // ArrayMethods class

```

14) United Way 50/50 Draw

A program is required to keep track of a United Way 50/50 Draw. Students are selling tickets in a draw where the winner gets half the money collected. The other half of the money will go to United Way. Your program should enter each student's name in the chance draw (array) and then randomly select one of the lucky entries. You do not know how many tickets you will sell but you have decided that there will be a maximum number of tickets available at the start of each draw. You should use constants to keep track of the maximum number of tickets (MAX_TICKETS) and the price of each ticket (TICKET_PRICE). Initially, MAX_TICKETS should be 200 and TICKET_PRICE should be \$0.50.

Students should be able to buy more than one ticket, however, your program should only need to enter each student's name once. Your program should be simple but user-friendly (make sure you can handle a variety of input errors). When no one wants to buy a ticket or when there are no more tickets left, you should draw the winning ticket and then display the winning number, the name of the winner, and how much money was won.

15 a) Write a method called `findIndexOf()`, that finds and returns the index of the first element of an array of integers that matches a certain value. This method has two parameters: the integer array that you are searching in and the number you are looking for (target). If the target is not found in the array, you should return -1. To help get you started the method heading for `findIndexOf()` is given below:

```
static int findIndexOf(int [] numbers, int target)
```

b) If you were told the array of numbers was sorted, how could you make your code more efficient? Explain how you would search the sorted array. Write out the code for this new method.

16) Write a Java method called `removeFromArray()` that removes the first occurrence of an integer number (target) from an array of integers. The method will have two parameters, the original array and the number you wish to remove. If the target is found, the method should return the new array (with the number removed). If the target is not found, the method should just return the original array. For example:

```
int [] numbers = {6, 7, 3, 9, 5};  
int [] newNumbers = removeFromArray (numbers, 9);
```

```
would set newNumbers to {6, 7, 3, 5};
```

17) Write a complete Java method including all comments called `indexOfSubArray` that returns the index of first occurrence of a sub array inside another array. To get a match the entire contents of the sub array has to appear in the first array in the exact same order with no breaks in the sequence. If the sub array does not occur in the original array as specified, the method should return -1. For example, given the following arrays:

```
int [] firstList = {4, 5, 5, 6, 7, 8, 9, 8, 4, 5};
```

```
int [] firstSubList = {5, 6, 7};
```

```
int [] secondSubList = {8};
```

```
int [] thirdSubList = {4, 6, 8};
```

```
int [] fourthSubList = {4, 5, 6};
```

`indexOfSubArray (firstList, firstSubList)` should return 2

`indexOfSubArray (firstList, secondSubList)` should return 5

`indexOfSubArray (firstList, thirdSubList)` should return -1

`indexOfSubArray (firstList, fourthSubList)` should return -1

To help get you started, the method heading is given below:

```
static int indexOfSubArray (int [] list, int [] subList)
```

18) A company keeps track of its customers and how much money they owe using two related arrays. The first array keeps track of the customer's name and the second array keeps track of their outstanding balance (how much money they owe). Assuming there are 50 customers, the following could be used to declare and create these arrays:

```
String [] names = new String[50];
```

```
double [] balances = new double[50];
```

You can assume that the values in the both arrays have already been assigned. Write the sections of Java code required to perform each of the following. Each question is separate. Methods are not required.

- Every month each customer is charged a \$2.00 account fee. Add this \$2.00 fee to each of the customer's balances.
- Display a list (with proper column headings) of the names and balances of all customers who have a balance greater than \$200.00
- Display the name of the customer with the highest balance. If more than one customer has the highest balance, you only have to display one of the names.

19) Letter Frequency Assignment

Write a complete Java program that counts the number of occurrences of each letter of the alphabet in a text file. You should ignore case (i.e. count 'A' and 'a' as the same letter). You should output your results in a nicely formatted table (see below). In your results, you should show the frequency of each letter as well as the percentage (rounded to 2 decimal places) of the total number of alphabetic characters (letters) in the file. The list should be sorted in order from the most frequent letter to the least frequent letter. Your program should ask for the name of the text file to analyze so that you can easily work with more than one file.

Sample Output:

```

Letter Frequency Analysis

File Name: sample.txt

Letter      Frequency      Percentage
E           104          11.30%
T           90           9.78%
A           85           9.24%
.           .            .
.           .            .
.           .            .
Z           1           0.11%
X           0           0.00%

Totals      920          100.00%

End of File Analysis

```

Make sure you plan out your code carefully before beginning. I would suggest you break the problem into two parts:

- 1) Count the number of occurrences of each letter in the file
- 2) Produce a sorted list of your results.

You should completely test your program using a small text file for input. Once you have your program working, analyze the "alice.txt" file in the class folder. This file contains the complete text of Alice in Wonderland.

Hint: For this problem you may want to use related arrays or a single array of a simple `Letter` class that keeps track of each letter and its frequency. If you use a `Letter` class, look at sections 6.3 and 6.4 to get ideas on how to sort your list by frequency. If set up properly, you can use the `Arrays.sort` method to sort your array.

20) In the game of Connect Four you have an upright board with 6 rows and 7 columns. To make a move, the player chooses a column to drop his/her piece in. The piece then drops to the next available position in that column. Each piece is either Black (B) or Red (R). For example, given the board position on the left, a move by B to column 3 will result in the position shown on the right.

				B		
		B		R		
	R	B	R	R	B	

B drops a piece in
Column 3

		B		B		
		B		R		
	R	B	R	R	B	

The object of the game is to get 4 in a row horizontally, vertically or diagonally. The following questions relate to the game of Connect Four. Note: To make part d) easier you may want to think about creating an 8 by 9 board with an extra border of EMPTY squares all around.

- a) Write the declaration to create a two-dimensional array to keep track of the board. Since integers are usually easier to work with you may want to make your array an array of integers and use the following constants to define the three possible contents of each square on the board:

```
final static int BLACK = 1;  
final static int RED = -1;  
final static int EMPTY = 0;
```

- b) Write a method called `clearBoard()` that resets the board to EMPTY. To help get you started, use the following method heading:

```
static void clearBoard(int [][] board)
```

- c) Write a method called `findRow()` that finds and returns the row to place the piece in. This method has 2 parameters: the board and the column selected. If the column is full, the method should return -1. To help get you started, use the following method heading:

```
static int findRow (int [][] board, int column)
```

- d) Write a method called `checkForWinner()` that checks to see if there is a winning combination on the board. This method has 3 parameters: the board to check and the row and column of the last piece put on the board. If there is a winner, the last piece placed must be part of the winning combination so these last two parameters can be used to make your checking easier. If there is a winner, this method should return who won (RED or BLACK). If there is no winner, it should return EMPTY. To help get you started, use the following method heading:

```
static int checkForWinner (int [][] board, int lastRow,  
                           int lastColumn)
```

21) The Music Department uses a computer program to keep track of the seats sold for their spring concert. One arrangement of the chairs for the concert has 24 rows with 35 seats in each row. To keep track of the seats sold they use a two dimensional array which is declared as follows:

```
boolean [][] seatsSold = new boolean[24][35];
```

Originally all entries in the array would be `false` (seat not sold). As seats are sold, the corresponding array element is changed to `true`. For example, if you sold seat number 7 in the third row, you would mark this seat as sold with the statement:

```
seatsSold[2][6] = true;
```

In this case we used 2 and 6 because the array is zero based. If you want, you can use a 25 by 36 array and ignore the zero row and column.

a) Write a method called `clearSeats()` that resets all of seats in the `seatsSold` array to not sold (`false`). To help get you started, use the following method heading:

```
static void clearSeats(boolean [][] seatsSold)
```

b) Write a complete Java method called `bestRowAvailable()` that finds and returns the lowest row number (row closest to the front) that has the number of seats that you are requesting beside each other in a row. For example if you were looking for 5 tickets in the three rows shown below where T is a sold seat and F is an unsold seat, row 2 is the lowest row with 5 seats beside each other. If there are no rows with the number of seats you are requesting together, the method should return `-1`.

Front of Theatre

```
Row 1  TTTFFTTFFTTFTTFFTTTTTFFTTFFFTTFFTTFF
Row 2  TTTTTFFTTTTFFFFTTTFTTTTTTTTTTTTTTTT
Row 3  TTTTTFFFFFFFFTTTTFFFFFFTTFTTTTTTTTTT
```

This method should have 2 parameters: the `seatsSold` array and the number of tickets you are looking for. To help get you started, the method heading is given below:

```
static int bestRowAvailable (boolean [][] seatsSold,
                             int noOfTickets)
```

22) Write a Java program that finds all possible ways to arrange 8 Queens on an 8 by 8 chessboard such that no Queen is attacking another Queen. Queens can attack pieces horizontally, vertically and diagonally.

23) In the game of Boggle you are given a 2D grid of letters and you are asked to find words on this grid. A word can be made by moving from letter to letter horizontally, vertically and diagonally. However, each letter on the board can only be used once. You will be given a letter grid and a list of words to find. If the word can be found, display the row and column of the first letter in the word. For example, given the letter grid below and the word "ELEPHANT", you should find "ELEPHANT" starting with the 'E' in row 1 and column 1. If the word occurs on the grid more than once, you should give the starting letter with the lowest row and then the lowest column. If a word can't be found, you should display an appropriate message.

```

ERSYT
ELBTO
PHENA
CBANI
WZUST

```

Input (from a file e.g. boggle.txt)

The first line of the input file will contain the size of the grid (no of rows followed by the number of columns). This will be followed by the letter grid with each row on a separate line. Next will be the number of words you are searching for, following by the words. You can assume all letters will be uppercase. Here is a sample input file:

```

5 5
ERSYT
ELBTO
PHENA
CBANI
WZUST
3
ELEPHANT
TONE
BANANA

```

Output (to the screen)

You should output the grid and then for each word found on the grid you should output the word followed by the row and the column of the first letter of the word. Note: The rows and columns start at 1. If the word is not found you should give an appropriate message. Here is the output for the input file given above. You may have noticed that TONE can also be found starting at (2, 4) but the starting position (1, 1) has a lower row number.

```

Boggle Word Search
ERSYT
ELBTO
PHENA
CBANI
WZUST
ELEPHANT found starting at (1, 1)
TONE found starting at (1, 5)
BANANA was not found
Done

```

Chapter 8 - Robot World

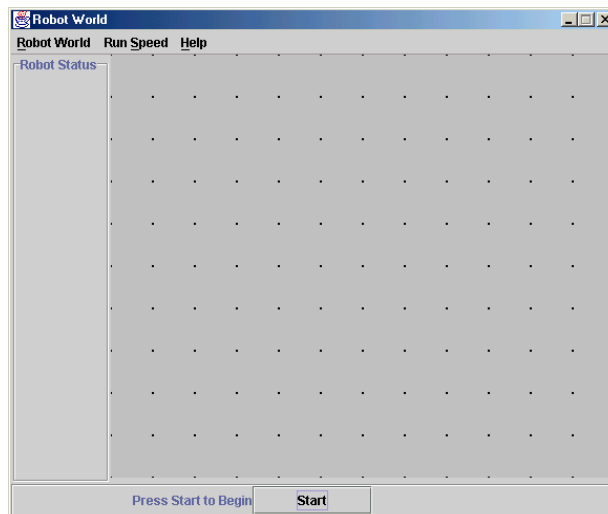
The Robot World package is a collection of classes developed by G. Ridout in the fall of 2001. Robot World was based on/inspired by the University of Waterloo's JKarel robot used in the first year Computer Science courses. The required code and support files will be available on the class Moodle. To use with Eclipse you will need to configure your project's build path to add the `robot.jar` file to the Libraries section as an external jar. You will also need to copy the supporting `.gif` files for the Robot and World images to your project folder. Finally, you will need to include the statements `import robot.*` and `import java.awt.Color` in your Java program to tell the compiler where to find the Robot World code and to use the `Color` class to change the colour of your robots.

8.1 Creating a New Robot world

Our first step when using the Robot World package is to create a new `World` for our robot to move about in. For example, the following statement will create a world with 10 rows and 12 columns:

```
World myWorld = new World (10, 12);
```

When you create a `World` object you create the following Swing application.



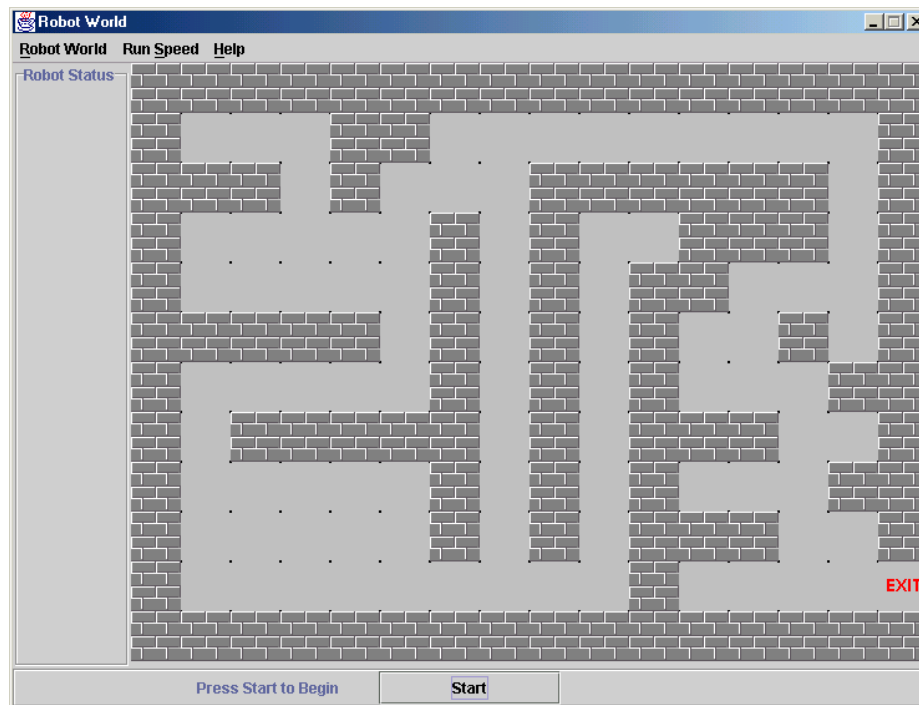
The main part of the Robot World is the grid area in which the robot moves. In this example, this area is the 10 rows high (numbered from 0 to 9) and 12 columns wide (numbered from 0 to 11). The rows and columns are numbered starting with 0 to match the Java array indices. The smallest possible Robot World is 7 by 7 and the largest possible world is 32 rows by 44 columns. The large grid size shown above is used for Robot Worlds up to 14 rows and 20 columns. For worlds with more columns or rows the smaller grid is used. The grid size is automatically selected when you create the world.

Next to the robot area is the Robot Status panel that shows the status of all of the robots currently in this world. We will discuss this panel in the next section when we add robots to the world. The bottom panel includes a Start button that is used to start the robot simulation. The top menus include options to exit the Robot World and to change the running speed of your robot.

A popular past time for robots is to run through a maze. Therefore, in addition to creating a Robot World with a blank robot area we can create a Robot World that contains walls for maze running.

For example: `World mazeWorld = new World ("maze.txt");`

will use the data in the file "maze.txt" to create the world shown below:



In this example, "maze.txt" contains the following text:

```
1111111111111111
1s00110000000001
1110100011111101
1000001010011101
1000001010110001
1111101010100101
1000001010100011
1011111010111001
1000001010100011
1000001010111001
100000000010000e
1111111111111111
```

As you may have guessed, '1' is used to represent a wall, '0' is used to represent a path, 'e' is used to represent the exit and 's' is used to represent the starting point. The starting point is not shown in the maze but it is used to give the initial position of the robot. It is important that this text file does not include any extra blank spaces at the end of each line or at the end of the file. It is also important that your file size is within the limits of the maximum and minimum number of rows and columns given earlier.

We will see in later sections that you can include other characters in this text file to add items other than walls when creating new Robot Worlds.

A Robot World without any robots is not very interesting, so in the next section we will see how to create new robots and then add these robots to our Robot World.

8.2 Creating a New Robot

To declare and create a new `Robot` object we use the following statement:

```
Robot myRobot = new Robot ("My First Robot", Color.RED);
```

This will create a new `Robot` object that will be referred to as "myRobot" in your program. This robot's screen name will be "My First Robot" and its main colour will be red. To add this robot to the Robot World created in the last section we would use the following statement:

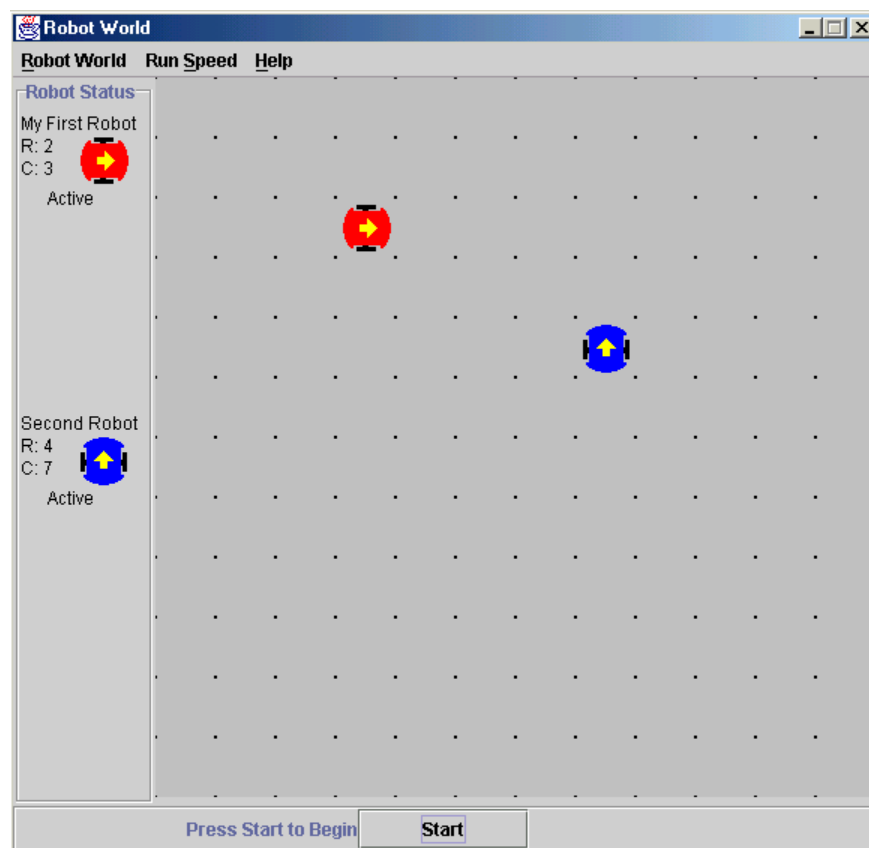
```
myWorld.addRobot (myRobot, 2, 3, Direction.EAST);
```

This will add the `Robot` referred to as `myRobot` to the `World` referred to as `myWorld`. The robot will be added in row 2 and column 3 (remember the upper left corner of the robot world is row 0 and column 0). The robot will initially be facing east. `Direction.EAST` is a static constant in the `Direction` class. Other directions include: `Direction.WEST`, `Direction.NORTH` and `Direction.SOUTH`.

Although most of our problems only require one robot, you can create and add more than one robot to a robot world. For example if we wanted a second robot in our world we can include the following statements:

```
Robot anotherRobot = new Robot ("Second Robot", Color.BLUE);  
myWorld.addRobot (anotherRobot, 4, 7, Direction.NORTH);
```

The following screen shot shows these two robots in a 12 by 12 robot world. Notice the Robot Status panel includes information on the robots including their current row and column, the direction they are facing and their status ("Active" or "Crashed"). Later we will see that this status area also includes a list of all items currently held by the robot.



In addition to adding a robot to a world in a particular row and column, facing in a particular direction, when working with maze files we will want to add our robot in a certain starting position. In this case we can use the following statement:

```
myWorld.addRobotAtStart (myRobot)
```

The above statement will add the Robot referred to as `myRobot` to the world `myWorld` in the row and column of the 's' character in the maze text file. The default direction for the robot is south. This method should only be used if the world has been created using a text file and this text file contains an 's' character.

Now that we have added a robot to a robot world we need explore some of the methods we can use to manipulate our robot.

8.3 Overview of Robot Methods

Appendix F has a complete list of all of the `Robot` methods available. To demonstrate some of these methods, we will present a series of demos. These demos can be found in the `Robot Demos` folder (available in the class folder or on the Moodle). It is hard to show the output of these programs on paper so you should run these demos on your own to see what they do. The first demo is called "RobotWorldDemoOne"

```
import robot.*;
import java.awt.Color;

/** The "RobotWorldDemoOne" class.
 * Purpose: To demonstrate the move(), turnLeft(), turnRight() methods
 * @author Ridout
 * @version July 2002
 */
public class RobotWorldDemoOne
{
    public static void main (String [] args)
    {
        // Create a 10 by 10 world
        World myWorld = new World (10, 10);

        // Create a red robot called "Demo One"
        Robot myRobot = new Robot ("Demo One", Color.RED);

        // Add Demo to the world at position (0,0) facing South
        myWorld.addRobot (myRobot, 0, 0, Direction.SOUTH);

        // Move and turn the robot
        myRobot.move ();
        myRobot.move ();
        myRobot.move ();
        myRobot.turnLeft ();
        myRobot.move ();
        myRobot.move ();
        myRobot.turnRight ();
        myRobot.move ();
        myRobot.move ();
        myRobot.move ();

    } // main method
} // RobotWorldDemoOne class
```

After you run this demo, you will need to press the start button in the bottom panel to start your robot moving. In this demo the robot will move ahead 3 spaces, turn left, and move 2 spaces, turn right and then finally move 3 spaces. You should note that when you are moving and turning a robot you need to specify which robot you are working with by using the "dot" notation. (e.g. `myRobot.move ()`)

You may want to try changing this code to see how it affects the robot's movements. If the robot moves too slowly you can change the speed of the robot by selecting the "Run Speed" menu option before you start your robot. If you are working on an older machine, the robot may still be too slow, so you can add the following statement after you create the robot:

```
myRobot.takeBigSteps();
```

This statement will increase the step size of the robot to speed things up. In some cases, when your robot is working on a large task, you may want even higher speeds. In these cases you can add the following statement:

```
myRobot.takeReallyBigSteps();
```

If you want to re-run the demo you need to re-run your program. The second demo uses a robot to draw out a 7 by 7 square box. This program is called "RobotWorldDemoTwo". To save space only the main program code is shown.

```
// Set up the World and the Robot
World myWorld = new World (9, 9);
Robot myRobot = new Robot ("Demo Two", Color.BLUE);
myRobot.takeBigSteps ();
myWorld.addRobot (myRobot, 1, 1, Direction.EAST);

// Trace out a 7 by 7 4-sided box
for (int side = 1 ; side <= 4 ; side++)
{
    // Take 6 steps, dropping markers as you go
    for (int step = 1 ; step <= 6 ; step++)
    {
        myRobot.move ();
        myRobot.dropMarker ();
    }
    // Make a right turn before tracing out the next side
    myRobot.turnRight ();
}

// Move to the centre of the box
for (int step = 1 ; step <= 3 ; step++)
    myRobot.move ();
myRobot.turnRight ();
for (int step = 1 ; step <= 3 ; step++)
    myRobot.move ();
```

This program introduces a new method called `dropMarker()` that allows you to drop a small circular marker in the robot world. This marker matches the colour of the robot. Modify this code to draw a smaller or larger box. Notice that only 6 steps are required to create a box that is 7 by 7.

The final demo in this section introduces two new very important methods: `isWallAhead()` and `isFacing()`. The first method is used to check if there is a wall directly ahead of your robot. It is a boolean method so it returns true if there is a wall ahead and false otherwise. The second method is used to check the current direction your robot is facing. It has a single parameter that is one of the four directions discussed earlier. It is also a boolean method, returning true if the robot is facing in the given direction and false otherwise. This demo is called "RobotWorldDemoThree". To save space only the main program code is shown.

```
World myWorld = new World (10, 10);
Robot myRobot = new Robot ("Demo Three", Color.GREEN);
myWorld.addRobot (myRobot, 0, 0, Direction.EAST);

// Move the robot back and forth in the Robot World
// Keep going until you reach the bottom (crash)

while (true)
{
    // Keep moving until you find a wall
    while (!myRobot.isWallAhead ())
        myRobot.move ();

    // If facing east, turn around to the right
    if (myRobot.isFacing(Direction.EAST))
    {
        myRobot.turnRight ();
        myRobot.move ();
        myRobot.turnRight ();
    }
    else // If facing west, turn around to the left
    {
        myRobot.turnLeft ();
        myRobot.move ();
        myRobot.turnLeft ();
    }
}
```

This demo has the robot traverse the world going back and forth from left to right and then from right to left. It uses the `isWallAhead()` to decide when to turn around. If it is facing east it makes a wide right turn and if it is facing west it makes a wide left turn. Even if you change the size of the world, this code will still work. Generally we want to write code that will work in any size world and that will be able to sense the robot's surroundings and adjust according. Finally, you may have noticed that this demo ended by crashing into the wall on the last turn. Normally you do not want your robot to crash into a wall. Try modifying the code so that your robot stops when it reaches the bottom corner instead of crashing.

8.4 Working with Items

To make more interesting robot tasks we can add various items to the Robot World and then we can program our robot to pick up and drop these items throughout the world. The following demo will add various items to the Robot World and then pick up and drop some of these items. This demo is called "RobotWorldDemoFour". To save space only the main program code is shown.

```
// Set up the World and the Robot
World myWorld = new World (10, 10);
Robot myRobot = new Robot ("Scavenger", Color.CYAN);
myWorld.addRobot (myRobot, 0, 0, Direction.EAST);

// We have a choice of 3 different shape items
// Before adding these items to the world we need to create them
// When creating these items, you need to specify both
// the type of shape and the colour
Item blueTriangle = new Item (Item.TRIANGLE, Color.BLUE);
Item greenCircle = new Item (Item.CIRCLE, Color.GREEN);
Item redSquare = new Item (Item.SQUARE, Color.RED);

// Once created we can add these items to our world by
// specifying the row and column where they should be placed
// Just like adding a Robot except Items don't have a direction
myWorld.addItem (blueTriangle, 0, 1);
myWorld.addItem (greenCircle, 0, 2);
myWorld.addItem (redSquare, 0, 3);

// We can also create and add some letter Items
// Each letter item is defined by a character and a colour
// If we want we can put more than one item in the same spot
// In this case the A is on the bottom and the C is on top
for (char letter = 'A' ; letter <= 'C' ; letter++)
{
    Item newLetter = new Item (letter, Color.YELLOW);
    myWorld.addItem (newLetter, 0, 4);
}
```

```
// Now we will move our robot across the top,
// picking up the items we just added
while (!myRobot.isWallAhead ())
{
    // Since we could have more than one item in the same spot
    // we use a while loop to keep picking up all items
    while (myRobot.isItemHere ())
        myRobot.pickUpItem ();

    // Move on to the next spot
    myRobot.move ();
}

// Now let's turn around and drop all of the items
// We will drop the first item picked up first so that
// the items will be dropped in reverse order
// A dropLastItem() method is also available
myRobot.turnRight ();
myRobot.turnRight ();

while (myRobot.getNoOfItems () > 0)
{
    myRobot.dropFirstItem ();
    myRobot.move ();
}
```

Look over this code carefully and make sure you understand each of the new methods presented. If necessary, refer to Appendix F for a description of each method.

Running this code you should have noticed that when a robot picks up an item it gets stacked into the robots storage area. If a robot picks up A, B and then C, C is the last item in the stack, and A is the first item in the stack. In this example, calling `dropLastItem()` will drop C and calling `dropFirstItem()` will drop A. There is no way of dropping the B unless you drop the A or the C first. If the robot dropped the last item (C) then B becomes the new last item. In the same way, if the robot dropped the first item (A) then B becomes the new first item. If the robot has only one item, it is both the first and last item.

Other useful methods that we will use in the exercises include: `isLastItem()` that compares the last item held by the robot to a particular item given as a parameter and `compareLastItemToItemHere()` that compares the last item to the item underneath the robot. This second method can be used to help sort items. A complete list of all methods is given in Appendix F.

8.5 RobotPlus: Extending the Robot Class

You may have noticed that in some of the examples certain sections of code are repeated more than once. In these situations we usually want to put this code into a method. For example, to turn a robot 180°, we could write the following method:

```
static void turnAround (Robot robot)
{
    robot.turnLeft();
    robot.turnLeft();
}
```

To use this static method we need to pass the `Robot` reference to the method as a parameter such as:

```
turnAround (myRobot)
```

Unfortunately this doesn't follow the same pattern as other methods such as `move()` and `turnLeft()` that we called using "dot notation" such as:

```
myRobot.turnLeft()
```

In these situations it would be nice if we could add the `turnAround()` method to other methods in the `Robot` class. Unfortunately this is not possible because we don't have access to the `Robot` class source code. Also, you don't normally want to modify a working class in case we add some errors or accidentally alter some of the existing behaviour.

Fortunately, there is an alternative. In Java, we can create a new class that is an extension of another class. This new class will inherit both the data and behaviour of the original class. We can then add new data and behaviour to this new class without altering the original code. In the following example, we will be creating a new class called `RobotPlus` that inherits from the `Robot` class. The initial code for the `RobotPlus` class can be found in the demo folder in a file called `RobotPlus.java`. This code is shown below:

```
import robot.*;
import java.awt.Color;

/** RobotPlus Class
 * A new class which is an extension of the Robot class
 * @author Ridout
 * @version July 2002
 */
```

```

public class RobotPlus extends Robot
{
    /** RobotPlus Constructor
     * Creates a new RobotPlus robot with the given name and colour
     * @param name the name of the robot
     * @param colour the colour of the robot's body
     */

    public RobotPlus (String name, Color colour)
    {
        // Calls the original Robot class constructor
        super (name, colour);
    }

    // New methods for the RobotPlus class

    /** Turns the robot around (180 degrees)
     */
    public void turnAround ()
    {
        this.turnLeft ();
        this.turnLeft ();
    }

    /** Moves the Robot the given number of steps
     * @param noOfSteps the number of steps to move the Robot
     */
    public void move (int noOfSteps)
    {
        for (int steps = 1 ; steps <= noOfSteps ; steps++)
            this.move ();
    }
}

```

In the class heading you will notice the key word `extends` that indicates that the `RobotPlus` class is an extension of the `Robot` class. Notice also that all of the methods are made public so that they are available outside of the class. Since we will be creating instances of the `RobotPlus` class and these methods will be acting upon these instances, these methods are also non-static methods. When we declare and create a `RobotPlus` object using a statement such as:

```
RobotPlus myNewRobot = new RobotPlus("Better Robot", Color.ORANGE)
```

we will call the constructor for the `RobotPlus` class. Since `RobotPlus` is an extension of the `Robot` class, to create a `RobotPlus` object we first need to create a `Robot` object. Therefore, in the `RobotPlus` constructor we see the statement:

```
super(name, colour);
```

that will call the `Robot` class constructor with the given name and colour. No other statements are needed at this time but we will see later that we can add additional initialization code in this constructor to set up a `RobotPlus` object.

Since `myNewRobot` is a `RobotPlus` object that is an extension of a `Robot` object it will be able to do everything a `Robot` can do plus it will have two new methods at its disposal. For example, we could use the following statements:

```
myNewRobot.turnAround(); // turns the robot 180 degrees
myNewRobot.move(3);      // moves the robot 3 steps
```

Since we defined these new methods within our new class we can use the dot notation. Let us take a few minutes to look over how these methods work.

The statement `myNewRobot.turnAround()` tells the computer to look for a method called `turnAround()` that is associated with the object `myNewRobot`. Since `myNewRobot` is a `RobotPlus` object, it looks for the code in the `RobotPlus` class definition. There it finds the `turnAround()` method so it goes to this section of code. Within this method there are two statements saying `this.turnLeft()`. Each of these statements will turn "this" Robot to the left. Since we called `turnAround` using `myNewRobot.turnAround()`, "this" Robot is `myNewRobot` and so `myNewRobot` turns left 2 times to complete the turn.

In this case, the "this" qualifier is optional, but may be needed in other examples. The `move()` method works in a similar fashion. In this case the `move()` method is "overloading" the original `move()` method in the `Robot` class. Our `move()` is distinguished from the original `move()` since it has an integer parameter, where the original `move` had no parameters. In the next section we will see how we can also "override" a method in the `Robot` class.

Since the `RobotPlus` class inherits both data and methods from `Robot` class all of the standard `Robot` properties (data) including the name, colour, position, direction, and items held by the `Robot` will be kept track of for your new `RobotPlus` object `myNewRobot`. Also, `myNewRobot` will inherit and can use all of the standard `Robot` methods such as `move`, `pickUpItem`, `isWallAhead` in addition to any new methods that you define in `RobotPlus`.

In the problems in this chapter, you will be adding many new methods to the `RobotPlus` class so that your robots can solve more complex problems.

8.6 RobotTrack: Example Code for an extended Robot

The following example shows the modified code for the `RobotTrack` class, a special `RobotPlus` like `Robot` that keeps track of its current movements. Using this code, you can mark a position and then return directly to that position at any time.

In addition to some new methods, this code also adds two new data fields to keep track of the movement of the robot. The variable `changeNS` keeps track of any changes in position in the north/south direction. For changes in the west/east direction we use `changeWE`. Since the northwest corner is (0, 0), moves in the south and east direction are considered positive and moves in the north and west direction are negative. You should also notice that the new data fields are private and all of the methods are public. This is normally how we would set up a new class. Look over this code carefully.

```
public class RobotTrack extends Robot
{
    // New private data fields to keep track of the Robot's movements
    private int changeNS;
    private int changeWE;

    /** RobotTrack Constructor
     * Creates a new RobotTrack robot with the given name and colour
     * @param name the name of the robot
     * @param colour the colour of the robot's body
     */
    public RobotTrack (String name, Color colour)
    {
        super (name, colour);
        // Set initial value of change variables to zero
        // by marking the initial spot
        markThisSpot ();
    }

    /** Marks the current location of robot by setting the two
     * change variables to zero
     */
    public void markThisSpot ()
    {
        changeNS = 0;
        changeWE = 0;
    }

    /** Turns the robot to face a given direction
     * @param direction the direction to face the robot
     */
    public void turnToFace (Direction direction)
    {
        while (!isFacing (direction))
            turnRight ();
    }
}
```



```

/** Moves the robot and keeps track of its change in position
 * @overrides the move() method in the Robot class
 */
public void move ()
{
    // Calls the original "Robot" move method
    super.move ();

    // Based on the direction the Robot is facing
    // update the change variables
    if (isFacing (Direction.NORTH))
        changeNS--;
    else if (isFacing (Direction.SOUTH))
        changeNS++;
    else if (isFacing (Direction.WEST))
        changeWE--;
    else if (isFacing (Direction.EAST))
        changeWE++;
}

/** Returns the robot to the marked spot by moving in such
 * a way as to return the two change variables back to zero
 */
public void goToMarkedSpot ()
{
    // Adjust the north/south position
    if (changeNS > 0)
        turnToFace (Direction.NORTH);
    else
        turnToFace (Direction.SOUTH);

    // Moves the Robot in either a North or South direction. When
    // move is called, the overridden version of move (given above)
    // is used and the value of changeNS is updated accordingly.
    // Robot will keep moving until it returns to the original
    // marked row (changeNS = 0)
    while (changeNS != 0)
        move ();

    // Adjust the west/east position
    if (changeWE > 0)
        turnToFace (Direction.WEST);
    else
        turnToFace (Direction.EAST);

    // Moves the Robot in either a West or East direction. The
    // overridden move method is called and changeWE is updated.
    // Robot will keep moving until it returns to the original
    // marked column (changeWE = 0)
    while (changeWE != 0)
        move ();
}
}

```

In this code we "override" the `move()` method with a more advanced `move()` method. This method calls the original `move()` method using `super.move()` and then adds some additional code to keep track of the robot's movements.

Write your own code to test the above `RobotPlus` class or try question 13 at the end of the chapter. Make sure that your code calls all of the new methods.

8.7 Chapter Summary

Keywords/phrases introduced in this Chapter. You should be able to explain each of the following terms:

overload

override

this

Methods introduced in this chapter:

World methods

`addItem()`
`addRobot()`

`addRobotAtStart()`
`World()` - 2 constructors

Robot methods

<code>compareLastItemToItemHere()</code>	<code>isLastItem()</code>
<code>dropFirstItem()</code>	<code>isWallAhead()</code>
<code>dropLastItem()</code>	<code>move()</code>
<code>dropMarker()</code>	<code>pickUpItem()</code>
<code>getNoOfItems()</code>	<code>Robot()</code> - 2 constructors
<code>isExitHere()</code>	<code>turnLeft()</code>
<code>isFacing()</code>	<code>turnRight()</code>
<code>isItemHere()</code>	

Direction methods and constants

`EAST`
`NORTH`
`SOUTH`

`WEST`
`random()`

Item methods and constants

`CIRCLE`
`compareTo()`
`compareLastItemToItemHere()`

`Item()` - 3 constructors
`SQUARE`
`TRIANGLE`

8.8 Questions, Exercises and Problems

1) Without using a computer, predict the output of the following Java program. Use graph paper to show what the world will look like after the program has finished including the last position and direction of the robot.

```
import robot.*;
import java.awt.Color;

public class RobotWorldPredict
{
    public static void main (String [] args)
    {
        World myWorld = new World (9, 9);
        Robot myRobot = new Robot ("Practice Robot", Color.BLUE);
        myWorld.addRobot (myRobot, 2, 2, Direction.NORTH);

        for (char letter = 'A' ; letter <= 'K' ; letter++)
        {
            Item newLetter = new Item (letter, Color.RED);
            myWorld.addItem (newLetter, 2, 2);
        }

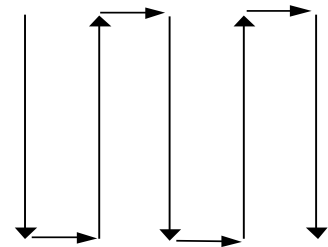
        while (myRobot.isItemHere ())
            myRobot.pickUpItem ();

        for (int trips = 1 ; trips <= 4 ; ++trips)
        {
            myRobot.turnRight ();
            for (int move = 1 ; move <= 4 ; ++move)
            {
                if (move % 2 == 0)
                    myRobot.dropLastItem ();
                else if (move == 3)
                    myRobot.dropMarker ();
                myRobot.move ();
            }
            myRobot.move ();
            myRobot.turnRight ();
        } // main method
    } // RobotWorldPredict class
}
```

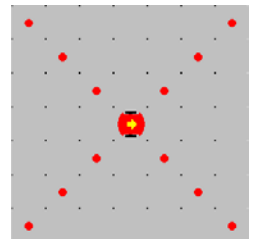
2) Explain the difference between overloading a method and overriding a method.

- 3) Write a Java program to do the following:
- Create a new 12 by 12 world
 - Create a new robot (pick your own colour and name). Place your robot in the world in position (3, 10) facing West
 - Move your robot 7 squares, turn around and then return back to the original starting position. Hint: Use a `for` loop to move your robot.
- 4) Look over the code for `RobotWorldDemoTwo`. Modify this code to satisfy the following:
- Create a new 17 by 17 world
 - Create and add a new orange robot called "Pumpkin" to this world starting in position (15, 15)
 - Have your robot trace out a 15 by 15 square. To mark each square you can use the command `dropMarker()`, which drops a coloured marker in the world.
 - After tracing out the square, have your robot move to the centre of the square and stop.

5) Look over the code for `RobotWorldDemoThree`. In this code, the robot traversed the entire world moving back and forth in a West-East direction slowly moving from North to South. Modify this code to make the robot traverse the entire world moving up and down in a North-South direction, slowly moving from West to East (see the diagram). Your code should work for any size world. Change the size of the world to 10 by 15 and see if it still works. It is OK if your robot stops by crashing into the final wall.



6) Write a Java program that has a robot trace out an 'X' shape using markers (see diagram). Once again, you should design your program so that it can work in any size world. To keep the shape of your 'X' symmetrical you can assume the world will be square and that the dimensions will be odd numbers (e.g. 11 by 11).

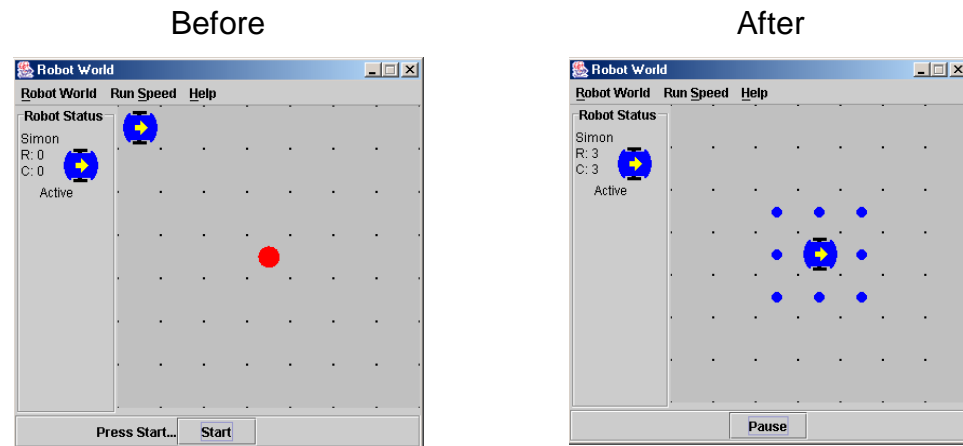


7) Write a Java program to get a robot to run through a maze. The maze world will be loaded from a file. A sample file called "maze1.txt" can be found in the Robot World Demo folder. You should use the `addRobotAtStart()` method to place your robot in the maze. For example:

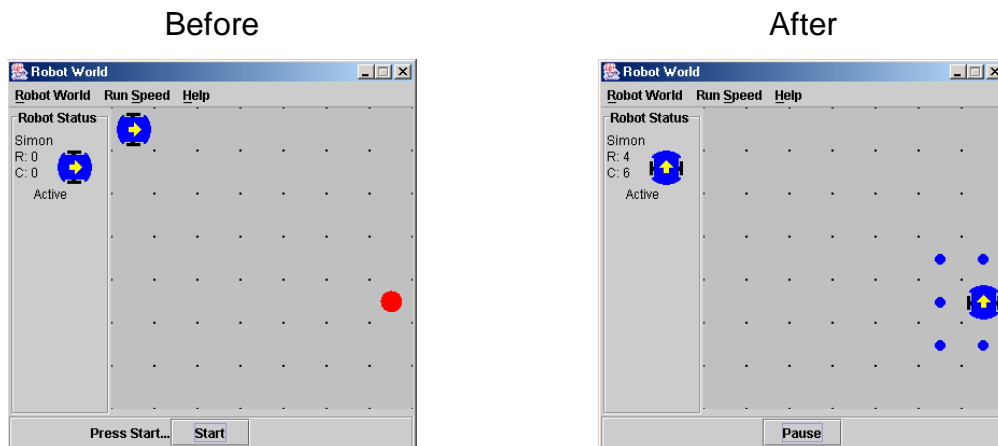
```
myWorld.addRobotAtStart(myRobot);
```

Your objective is to find the exit (use `isExitHere()` to check for the exit). Your program should work for any maze file. Hint: To run the maze, have your robot hug the left or right wall. To test your program, try making a maze of your own.

8 a) Write a Java program that creates a 7 by 7 World and then randomly places a coloured circle (you choose the colour) somewhere in this world. For the initial problem, you can assume the circle will not be placed on an edge (i.e. the randomly generated row and column will be between 1 and 5). Your robot's task is to find this circle and put markers around the outside of the circle. Your robot should then move on top of the circle. To make your program more realistic, your robot should not know any information about the World ahead of time including the size of the World or the position of the circle. Hint: Modify question 5 to search for the circle using the method `isItemHere()`. Here are some before and after pictures showing how this task should be completed.



b) Modify your program so that the circle may be placed anywhere in the World. In this case, your row and column will be randomly generated between 0 and 6. For your modified program make sure your Robot can handle the situations when the circle is placed in a corner or along an edge. For example:



9) Add the following methods to the RobotPlus class. Read each description carefully. Don't forget to include complete comments for each method:

```
turnToFace(Direction direction)
    -turns the robot to face in a given direction

goToCorner(Direction directionNS, Direction directionWE)
    -sends the robot to one of the four corners. For example:
    myRobot.goToCorner(Direction.NORTH, Direction.EAST)
    would send the robot to the Northeast corner of the robot world. You can
    use the turnToFace() method inside this method.

pickUpAndMove()
    - picks up all items in the robot's current location and then moves one
    square ahead (if possible) in the current direction. To find and pick up the
    items use isItemHere() and pickUpItem().
```

**To test your methods, try completing the following tasks
using the RobotPlus class.**

10) Write a program that starts your robot in a random position facing in a random direction. You can use `Direction.random()` to randomly pick the starting direction of your robot. Then, using `goToCorner()`, drop a marker in each of the 4 corners.

11) Create a 12 by 12 world with the letters A, B, C and D in random locations throughout the world. Your task is to program a robot to find the four letters and place one letter in each of the four corners. You can start your robot in any location facing in any direction. To randomly place the letters in your world, use the following code. You can change the colour of the letters if you want.

```
for (char letter = 'A' ; letter <= 'D' ; letter++)
{
    Item newLetter = new Item (letter, Color.RED);
    int row = (int) (Math.random () * 12);           // 0 - 11
    int column = (int) (Math.random () * 12);        // 0 - 11
    myWorld.addItem (newLetter, row, column);
}
```

Hints: In addition to the new methods created above for the RobotPlus class, you can use the code from question 5 that made the robot traverse the entire world moving up and down in a North-South direction, slowly moving from West to East.

To help complete the following problems you may want to add some additional code to your RobotPlus class.

12) Create a 10 by 10 world with ten randomly placed triangles and ten randomly placed squares. (You pick the colours but make all of your triangles the same colour and all of your squares the same colour). Your task is to program a robot to put all of the triangles on the West wall (down the wall from North to South) and all of the squares on the East wall (also down the wall from North to South). Try to accomplish this task as quickly as possible. To add a shape to your world, use the following commands:

```
Item blueTriangle = new Item(Item.TRIANGLE, Color.BLUE);
myWorld.addItem(blueTriangle, row, column);
```

To see if an item is a triangle or a square use the command `isLastItem()`. For example:

```
myRobot.isLastItem(blueTriangle)
```

will return true if the last item is a blue triangle.

13) Create a 12 by 12 world with two randomly placed shapes (a blue triangle and a green square). Your task is to program a robot to find the two items and exchange their positions, putting the blue triangle where the green square was and vice versa. There are many different ways of approaching this problem so you should plan out your code to complete this task in the most efficient way. Read over section 8.6 before trying this problem

14) Create a 20 by 30 world with 30 random letters (ranging from 'A' to 'Z') placed randomly throughout this world. Note: Both the value and position of the letters are random. Your task is to create a robot that picks up all of these letters and places them on the North wall in alphabetical order. To help sort the letters you can use the `compareLastItemToItemHere()` method. Since the letters will be generated randomly and you will be working with 30 letters, you will have to deal with sorting duplicate letters. To generate a random letter Item you can use the following statements:

```
char randomLetter = (char) ('A' + Math.random () * 26);
Item newLetter = new Item (randomLetter, Color.RED);
```

15) The Robot class has a method called `getNoOfItems()` that returns the number of items currently held by a robot. Assuming this method was not available in the Robot class, what additional code would you have to add to the RobotPlus class if you wanted to add a `getNoOfItems()` method to the RobotPlus class. **Hint:** You will need to override all of the methods that add or remove items from the robot's stack.

Appendix A -- Java Style Guide

A.1 Naming Identifiers

Identifiers are used to name classes, variables, constants and methods in your Java programs. When picking identifiers, you must follow the following guidelines. Identifiers may not be keywords such as **import**, **int**, **double**, **public** or **class**. They must start with a letter, underscore character (`_`) or a dollar sign (`$`). After the first character, you can include any combination of letters or numbers. Java is case sensitive, so `Sum`, `sum` and `SUM` are distinct (different) identifiers.

It is good programming practice to use descriptive and meaningful identifiers in your programs. By doing so it will make your programs easier to follow and understand. You should resist the temptation to use a short identifier simply to save a few keystrokes. For example if you wanted to keep track of the rate of pay for each of your employees, you could use an identifier such as `rateOfPay`. This is more meaningful than using shortened identifiers such as `r`, `rate`, or `rop`.

Since we will be using meaningful identifier names, you may find your variable names include several words joined together. To make it easier to separate the words, use the following guidelines:

When naming variables or methods:

- The first letter of the variable name is lowercase
- Each subsequent word in the variable name begins with a capital letter.
- All other letters are lowercase.

This form is sometimes known as camel notation. For example:

```
int noOfStudents;           String lastName;  
double totalMonthlySales;  char tryAgain;
```

Even though we use the same convention for both variables and methods, we can easily distinguish between the two since method names are always followed by parentheses even if they have no parameters. For example: `nextInt()` or `nextDouble()`.

When naming classes:

Follow the same convention used for variables and methods, except that the first letter of the variable name is uppercase. For example:

```
public class BouncingBall  
public class RationalNumber
```


When naming constants (final):

Since constants do not change their value we want to distinguish them from variables. Therefore names of constants are in uppercase. If a name consists of several words, these are usually separated by an underscore(_). For example:

```
final int MAX_SIZE = 1000;  
final double HST_RATE = 0.13;
```

A.2 Program Comments

Internal comments should be used in your Java programs to explain the code and make it easier for other programmers to understand. Also, when studying for tests and exams, comments will help you understand your own programs.

In Java you have three types of comments:

```
// The first style of comment begins with two //'s and then  
// continues to the end of the line. This type of comment  
// is preferred for single line comments because you don't  
// have to worry about closing off the comment
```

```
/*  
The second style of comment uses the / and the * to start the  
comment and the * and / to finish the comment. It is good for  
multiple line comments and for commenting out a section of code  
when you are testing your program. If you use this type of  
comment don't forget the * and / at the end.  
*/
```

```
/** The third style of comment is very similar to the second  
 * style. the extra * on the first line is added to show that  
 * this is a javadoc comment. We will be using these comments  
 * for our program's introductory comments and when commenting  
 * method descriptions. By using this style and the @ options  
 * you can use javadoc to automatically generate HTML  
 * documentation. Some of the special @ options we will be using  
 * are shown below (see online help for a complete list):  
 * @author  
 * @version  
 * @param  
 * @return  
 * @throws  
 */
```

The following comments are a minimum requirement in your Java programs in this course:

1) Introductory comments at the start of the program. Includes the title, purpose of the program, programmer's name (not your student number) (@author), and date last modified (@version). For example:

```
/** Introductory Program
 * Displays a Welcome Message
 * @author G. Ridout
 * @version Date: September 15, 2001
 */
```

2) Comments should be used throughout the program to describe each section of code and to help make the code easier to follow. You do not need to comment each line of the program. With high-level languages such as Java, your code should be written so that it is easy to follow. Comments on sections of code help you and other programmers scan the code quickly. For example:

```
// Find the highest number in a list of integers
int highestNumber = listOfNumbers[0];
for (int index = 1; index < listOfNumbers.length; index++)
{
    if (listOfNumbers[index] > highestNumber)
        highestNumber = listOfNumbers[index];
}
```

In this case you do not need to comment the code within the loop since it is easy to understand once you know the purpose of the section of code. Never comment the obvious. For example

```
int number = keyBrd.nextInt();    // Reads in an integer number
```

In this case the comment just restates what the line does which is pretty obvious from the original code. These types of comments are not necessary.

3) For each method include an introductory comment section to explain its function and purpose. A description of its parameters, preconditions and post conditions and any return values should also be included. For example, the following comments would be for a `pow()` method:

```
/** Returns the value of the base raised to the
 * given exponent
 * @param base    the base of the power
 * @param exponent the exponent to raise the base to
 * @return        the power (base raised to exponent)
 */
```

Some parts of this comment may seem redundant but it is important that all parts are included. If in doubt about the number of comments you should put into your program, check with your teacher.

A.3 Code Paragraphing and Format

Spaces, indenting and blank lines can be used to make your Java code more readable from a programmer's point of view.

Indenting/Formatting Your Code

To help identify different structures (loops and selections) within your program code you should use indents and spaces to highlight these structures. By doing so, you will make your code easier to follow and debug. When using the Eclipse IDE we can press `Ctrl+Shift+F` to automatically format your code. If pressing `Ctrl+Shift+F` doesn't indent your code properly, your code may contain errors or extra blank lines.

There are a variety of styles used to indent code in Java but I recommend the following since it quickly shows the matching curly braces.

```
if (age >= 18)
{
    System.out.println("You can vote");
    noOfVoters++;
}
```

Blank Lines

You should add blank lines to separate sections of code to make it easier to follow and debug your code. For example:

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

// Read in the hours worked and the rate of pay
System.out.print("Please enter your hours worked: ");
double hoursWorked = keyboard.nextDouble();
System.out.print("Please enter your rate of pay: ");
double rateOfPay = keyboard.nextDouble();

// Calculate and display the Gross Pay
double grossPay = hoursWorked * rateOfPay;
System.out.print("Gross pay: %.2f\n", grossPay);
```

Appendix B -- Finding and Correcting Program Errors

This appendix introduces the various types of errors found in computer programs. It then provides some strategies on how to find and eliminate these errors.

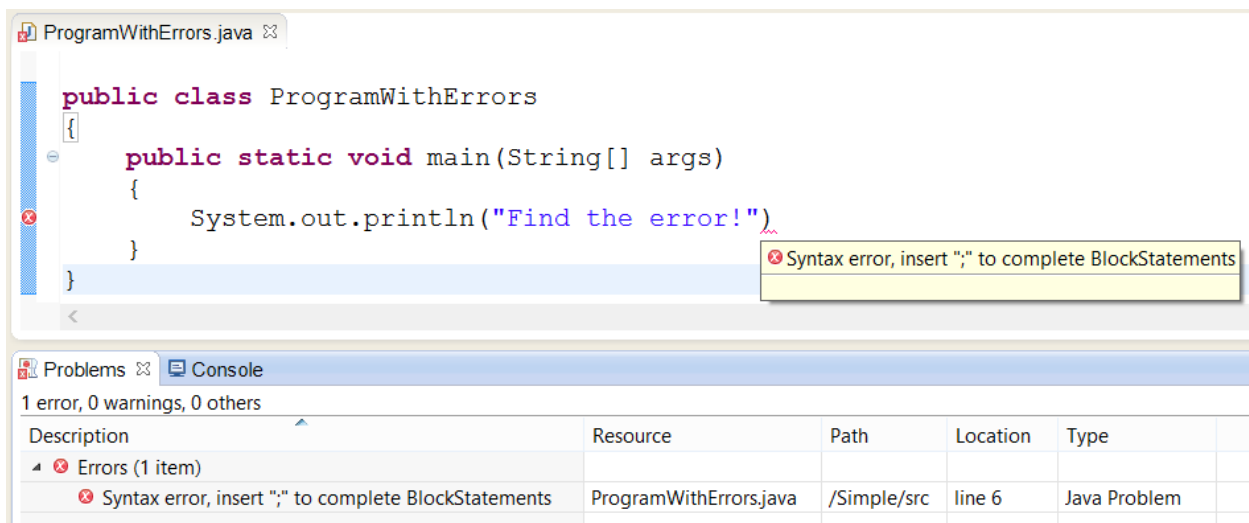
B.1 Types of Errors in Java Programs

The types of errors found in programs can usually be classified into compiler errors (including syntax errors), logic errors and run-time errors. As a programmer, you are responsible for writing error free code that can handle all possible situations including errors in user input.

Compiler Errors including Syntax Errors

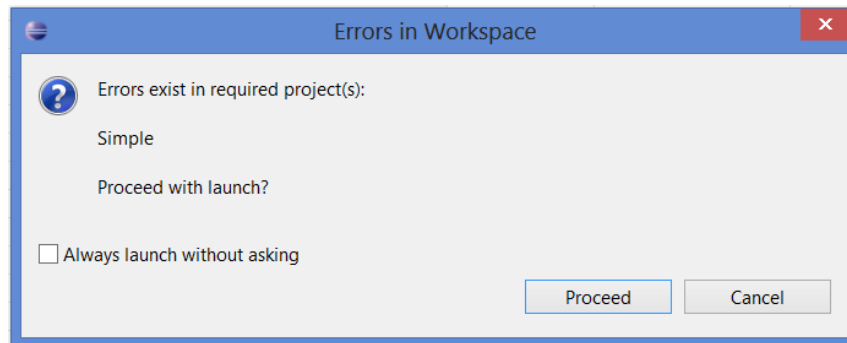
If you misspell a keyword, forget some punctuation such as a `;` or a `"`, or forget to close off a pair of curly braces, you will get a syntax error when you try to compile your program. Since the compiler can't understand the syntax (structure or the form) of the language used, it will be unable to properly translate your code. Other compiler errors, that are not specifically syntax errors, include errors such as failing to declare or initialize a variable before its first use.

The Eclipse IDE usually highlights compiler errors with a red X in the left margin of the line where the error occurred and a red squiggly line where the error is. The error will also appear in the "Problems" window. This window will include all errors and warnings for any open projects. If you hover over the error you will also get a brief description of the error (pop up window on the right below). Here is an example:

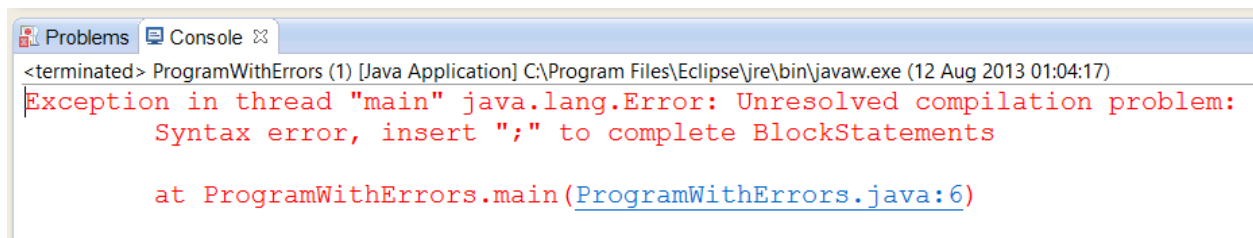


In this case the error message indicates that a `;"` is missing.

If you miss these warning signs and you try to run your program, you will get a pop-up error message (see next page). This error box appears if any of your open projects contain errors so you should normally close unrelated projects.

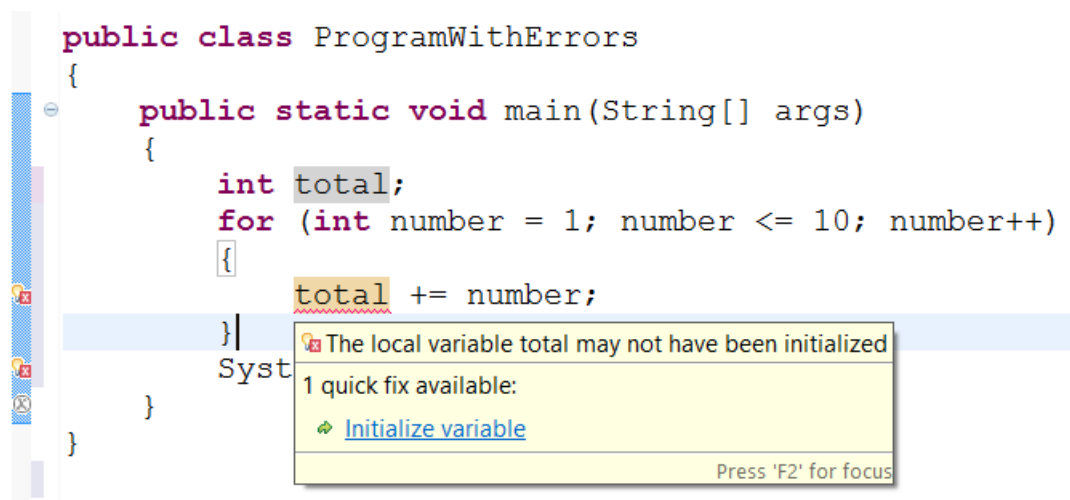


If you choose to proceed you may get an error message in the Console window. This error message will indicate the line number (e.g. 6) where the error occurred.



When your program has compiler errors, the compiler does its best job to describe the error but it is not perfect so don't forget to look for errors in the lines above or below the highlighted line or somewhere else in your code.

Eclipse has a great feature called “quick fix”, that will offer suggestions on how to fix simple compiler errors when you hover over the error. For example in the following section of code, we forgot to initialize the total to zero. This caused 2 errors (see X's on left margin) since we can't add to or display an uninitialized total. To accept the quick fix, we just click on it. In this case the quick fix will automatically add the code to set the total to 0, fixing both errors.



In most cases the first suggested quick fix will fix the problem but sometimes you have to look at a choice down the list and in some cases none of the suggested quick fixes will fix the problem. So, think before you select.

When you first start writing Java code, you may find that you are making a lot of syntax errors until you become more familiar with the language. You should get in the habit of checking your code before you run it to avoid syntax errors. Even though the compiler will find these errors for you, by finding your own errors you will learn the language more quickly.

Logic Errors

Unlike syntax errors, logic errors are harder to find. When you have a logic error your program will compile and run but it will do the wrong thing. For example:

```
if (age < 18)
    System.out.println("You can vote");
```

In this example, the syntax of the statement is correct so the program will run, but it will give the wrong results. Logic errors are usually found where you make logical choices, for example, in if or loop structures. Incorrect formulas in a calculation could also be considered logic errors.

Run time Errors

As the name suggests, run time errors occur when your program is running. Sometimes run time errors are totally the programmer's fault such as indexing an array outside of its bounds or a null pointer access (trying to access an object that hasn't been created). In other cases run time errors may be caused by invalid user input. For example, the user may enter a value that causes a division by zero in a calculation or the user enters "12a" when you asked for an integer value. In both cases the programmer is responsible for fixing or catching these errors.

Even though you may feel that user input errors are not the programmer's responsibility, good programmers will anticipate and deal with these potential problems in their code. For example, if the user's input could cause a division by zero error in a calculation, you should check for this before the calculation and then deal with the situation appropriately.

The Java language usually reports runtime errors by throwing an object called an *Exception*. Some of the more common exceptions that you may see include: *StringIndexOutOfBoundsException*, *ArrayIndexOutOfBoundsException*, *FileNotFoundException* and *NullPointerException*. See Appendix G for more on exceptions and exception handling.

B.2 Tips to Avoid Errors

Here are some tips you can follow to minimize the number of errors in your Java programs.

1) Plan out and test your algorithm before writing any code

Before you begin writing any code you should make sure you have planned out how you are going to solve the particular problem first. Don't forget to test your plan as well using your test cases (see Appendix B.3)

2) Write your comments before you write your code

By laying out your logic ahead of time you can avoid logic errors. This also helps if someone else is trying to find errors in your code.

3) Keep your code simple

Simple code is easier to follow and easier to debug. If you find your code is getting too complicated, you should re-think your original algorithm or look for a different approach to the problem. Surprisingly, even complicated problems can usually be solved with fairly simple well thought out code.

4) Follow the Java Style Guide in Appendix A

It is easier to find errors and debug your code when your code has a consistent look and format. Using descriptive variable names helps to avoid logic errors.

5) Indent/Format your code

The Eclipse IDE will automatically indent (format) your code (use Ctrl+Shift+F) making your code easier to read and also making it easier to find errors. The following examples illustrate some of the common mistakes that can be found by formatting your code. In each case you should notice how the indenting of the code highlights the error.

a) Extra semi-colon after an "if" or a "while"

Original Code:

```
if (hoursWorked > 40);  
    overtimeHours = hoursWorked - 40;
```

Code after indenting with Ctrl+Shift+F:

```
if (hoursWorked > 40)  
    ;  
    overtimeHours = hoursWorked - 40;
```

b) Missing curly braces (Both statements should be part of the "if")

Original Code:

```
if (hoursWorked > 40)  
    overtimeHours = hoursWorked - 40;  
    System.out.println("Hard worker");
```

Code after indenting with Ctrl+Shift+F:

```
if (hoursWorked > 40)  
    overtimeHours = hoursWorked - 40;  
    System.out.println("Hard worker");
```

B.3 Using Test Data to Find Errors

Since your program will not run with compiler errors, these errors are usually easy to find and correct. On the other hand, logic and run time errors are harder to find because your program will run and, in some cases, will run correctly most of the time. However, under certain conditions, your program will produce incorrect results or crash.

To help catch these intermittent errors it is important to test your program with a good set of test data. Your test data should include a variety of test cases. In particular you should make sure you test the extremes (zero, one, maximum and minimum values) and any boundary values in any loops or decisions. Also, looking at your code, make sure that your test cases give complete “code coverage” (every section of your code should be reached at least once). This is especially important for any decision or loop structures. Finally, when dealing with user input, you may want to check if the input is valid and give the user a chance to re-enter.

To organize our test data we create a test plan. A test plan should show the predicted outputs for different inputs. The number of tests cases depends on the nature of the problem, but in most cases you should include at least half a dozen different test cases. Here is an example test plan for problem 17 in Chapter 2.

Case	Input No of Slices	Predicted	Output	Reason for including this test case	OK ✓
		Hours	Minutes		
1	0	0	0	No pizza is a valid case	
2	-1	n/a	n/a	Should give invalid message	
3	1	0	39	Check 0 hours (rounds up)	
4	2	1	17	Need to round down	
5	3.5	2	16	Haven't you ever had half a slice of pizza	
6	7	4	31	Someone is hungry	
7	154.929	100	0	Should still work (Checks 0 minutes)	

The final column indicates that you have checked your program to see if it matches the test data. It is important that you actually check your program and don't just assume it is correct. You should also create your test plans as soon as possible so that you can use the test plan to test both your algorithm and your program.

Since it is very important that your programs produce correct output, you should create a test plan for every program you write. Since you usually cannot test all possible inputs it is important that you check a variety of inputs to catch any errors. In some cases you will be asked to hand in your test plan and explain your choice of test cases.

Appendix C – Unicode Characters

C.1 Unicode Characters Displayable in the Eclipse Console

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
002		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
003	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
004	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
005	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
006	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
007	p	q	r	s	t	u	v	w	x	y	z	{		}	~	□
00a		ı	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	­	®	¯
00b	°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿
00c	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
00d	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
00e	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
00f	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
015	?	?	Œ	œ	?	?	?	?	?	?	?	?	?	?	?	?
016	Š	š	?	?	?	?	?	?	?	?	?	?	?	?	?	?
017	?	?	?	?	?	?	?	?	Ÿ	?	?	?	?	Ž	ž	?
019	?	?	f	?	?	?	?	?	?	?	?	?	?	?	?	?
02c	?	?	?	?	?	?	^	?	?	?	?	?	?	?	?	?
02d	?	?	?	?	?	?	?	?	?	?	?	?	~	?	?	?
201	?	?	?	—	—	?	?	?	`	'	,	?	"	"	"	?
202	†	‡	•	?	?	?	...	?	?	?	?	?	?	?	?	?
203	%	?	?	?	?	?	?	?	?	<	>	?	?	?	?	?
20a	?	?	?	?	?	?	?	?	?	?	?	?	€	?	?	?
212	?	?	™	?	?	?	?	?	?	?	?	?	?	?	?	?

The row heading gives the first 3 hex (hexadecimal or base 16) digits for the Unicode character and the column heading gives the last hex digit. For example to display: ©, use `\u00a9` or to display: •, use `\u2022`. Note: the ? characters (except for `\u003f`) are characters that will not display in the Eclipse Console. Also, you may have noticed that the numbers on the left are not consecutive since lines that had no displayable characters have not been included.

Appendix D -- The Math and Character Classes

For more operations on numbers and characters we can use methods in the Java Math or Character class. Since these classes are in the standard `java.lang` package, an import statement is not required to use these methods. However, since these are all `static` methods, you must precede each method name with `Math` or `Character`. For example, to use `sqrt()` you would use `Math.sqrt()` and to use `toUpperCase()` you would use `Character.toUpperCase()`

D.1 The Math Class in Java

Most of the methods listed are overloaded to handle different types. For example, `abs()` can handle `double`'s, `float`'s, `int`'s or `long`'s. For simplicity only one version is listed. In each case the return type is usually the same as the argument (parameter) type. Note: `pow()` and `sqrt()` will always return a `double`.

Variables (constants)

PI - value of π

e.g. `area = Math.PI * radius * radius`

Methods

`abs(number)` - Returns the absolute value (distance from zero) of a number.

e.g. `Math.abs(-3.23)` would return 3.23

`max(firstNumber, secondNumber)` - Returns the larger of two numbers.

e.g. `Math.max(3,9)` would return 9

`min(firstNumber, secondNumber)` - Returns the smaller of two numbers.

e.g. `Math.min(3,9)` would return 3

`pow(base, exponent)` - Returns a power given the base and the exponent.

e.g. `Math.pow(2,4)` would return 16.0 (i.e. 2^4)

`random()` - Returns a random number between 0.0 (inclusive) and 1.0 (exclusive)

e.g. `Math.random()*10` would return a random number between 0 and 9.9

`round(number)` - Returns the number rounded to the closest integer (returns a `long`)

e.g. `Math.round(2.56)` would return 3

`sqrt(number)` - Returns the square root of a number.

e.g. `Math.sqrt(16)` would return 4.0

Additional methods, not listed, include trigonometric functions (`sin`, `cos` and `tan` and their inverses). See the `Math` class in the Java API help for a complete list.

D.2 The Character Class in Java

Methods

The following "is" methods return true if the character fits the description and false otherwise.

`isDigit(char ch)` - checks if the character is a digit.

`isLetter(char ch)` - checks if the character is a letter.

`isLetterOrDigit(char ch)` - checks if the character is a letter or digit.

`isLowerCase(char ch)` - checks if the character is a lowercase character.

`isUpperCase(char ch)` - checks if the character is an uppercase character.

Examples:

```
Character.isDigit('7') will return true
Character.isDigit('A') will return false
Character.isLetter('a') will return true
Character.isLowerCase('A') will return false
```

The following methods will convert the character to upper or lower case. In both cases, they return the converted character. If no conversion is possible or required, the original character is returned.

`toLowerCase(char ch)` - converts the character to lowercase.

`toUpperCase(char ch)` - converts the character to uppercase.

Examples:

```
Character.toUpperCase ('a') will return 'A'
Character.toLowerCase ('G') will return 'g'
Character.toLowerCase ('h') will return 'h'
Character.toUpperCase ('9') will return '9'
```

Appendix E – Reading from and Writing to Text Files

The following sections cover reading from text files using either a `Scanner` (more convenient) or a `BufferedReader` (faster) and writing to text files using a `PrintWriter`.

E.1 Using a `Scanner` to read from a text file

To read data from a text file we can use a `Scanner` object created with a `File` object parameter instead of the `System.in` parameter used for keyboard input. For example to read and display all of the lines in a text file, we can use the following code:

```
Scanner inFile = new Scanner(new File("data.txt"));
while (inFile.hasNextLine())
{
    String nextLine = inFile.nextLine();
    System.out.println(nextLine);
}
inFile.close(); // Close the file asap
```

The `hasNextLine` method returns `true` if there is still data left in the file. The `nextLine` method reads the text file line by line. We can also use the `Scanner` methods `next`, `nextInt`, `nextDouble`, `nextLong` etc. to read individual elements from the file one at a time. If we don't know how much data is in the file, we can use the methods: `hasNext`, `hasNextInt`, `hasNextDouble`, `hasNextLong` etc. to check if there is still data left in the file.

When opening the file, if the given file doesn't exist, a `FileNotFoundException` will be thrown. To deal with this checked exception, you need to include **`throws`** `FileNotFoundException` at the end of the main method heading. For example:

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

In this case, if there is an error opening the file we would just ignore the error and our program would terminate with a run-time error. This is a simple way of dealing with a checked exception. In Appendix G we will see how we can use a `try catch` block to properly deal with checked exceptions. Note: To use the `FileNotFoundException` and `File` classes you will also need to include: **`import java.io.*;`** at the top of your program.

Here is another example that reads a simple text file used to track of the top player and his/her score for a game. This code assumes the text file would contain the name of the player on the first line and their score on the second line. For example, assume the following data is in a text file called `"topScores.txt"` :

```
Jim Nasium
3456
```

```
// Open the file for input
// "topScores.txt" is the name of the file
Scanner fileIn = new Scanner (new File ("topScores.txt"));

// Read in the topPlayer and his/her score
// This is very similar to keyboard input
String topPlayer = fileIn.nextLine();
int topScore = fileIn.nextInt();

// Close the file
fileIn.close();
```

E.2 Using a `BufferedReader` to read from a text file

Using a `Scanner` to read from a text file is very convenient since you can use methods such as `nextInt` and `nextDouble` to read numbers from the file even if the numbers are not on the same line. However, since `Scanners` can be quite slow, for larger text files we should use a `BufferedReader` which is much faster. Below is the code to find the total of a file of integer numbers using a `BufferedReader`.

We will assume that we don't know how many numbers are in the file so we will keep reading the file until the `readLine` method returns a `null` value which indicates the end of file. One disadvantage of `BufferedReaders` is that they don't have specific methods to read integers and doubles etc. so we need to read each line as a `String` and then convert it to a number using `Integer.parseInt` or `Double.parseDouble`.

```
// Since this is large file, the total could get big
long total = 0;

// Open the file
BufferedReader fileIn =
    new BufferedReader(new FileReader("numbers.txt"));

// Keep reading the file until null is returned (end of file)
String line;
while ((line = fileIn.readLine()) != null)
{
    // Need to convert and add in each number
    int next = Integer.parseInt(line);
    total += next;
}

// Close the file and display the total
fileIn.close();
System.out.println(total);
```

This code assumes that the file contains one number per line. If the file contained more than one number on each line, we would need to use a `StringTokenizer` to break down each line.

Note: The code on the previous page can throw an `IOException` so we need to use a `try...catch` block or add **throws** `IOException` to the heading of our main program (see example in E.1). The `IOException` class is a superclass of the `FileNotFoundException` so if you include **throws** `IOException` you don't need to include a **throws** `FileNotFoundException` as well.

To use the `BufferedReader`, `FileReader` and the `IOException` classes you will also need to import each class or include **import** `java.io.*`; at the top of your program to import them all at once.

E.3 Using a `PrintWriter` to write to a text file

To write data to a text file we can use a `PrintWriter` object and the `print`, `println` and `printf` methods. Writing to a text file is very similar to writing to the System console using `System.out.println`.

Here is an example section of code that opens up a text file and then writes the name of the top player and his/her score to this file. This code will create the same file discussed in sections E.1. Note: In this code we used `println` but we could have also used `printf` if we wanted formatted output.

```
// Opens the file for output
// "topScores.txt" is the name of the file
PrintWriter fileOut =
    new PrintWriter(new FileWriter("topScores.txt"));

// Writes the top players name and score to the file
// Assumes topPlayer and topScore have been defined
// Writes one value per line (makes it easier to read in)
fileOut.println(topPlayer);
fileOut.println(topScore);

// Close the file as soon as you are finished with it
fileOut.close();
```

Note: The above code can throw an `IOException` so we will need to use a `try...catch` block or add **throws** `IOException` to the heading of our main program (see example in E.1). The `IOException` class is a superclass of the `FileNotFoundException` so if you include **throws** `IOException` you don't need to include a **throws** `FileNotFoundException` as well.

To use the `PrintWriter`, `FileWriter` and the `IOException` classes you will also need to import each class or include **import** `java.io.*`; at the top of your program to import them all at once.

Appendix F -- Robot World Methods and Constants

F.1 The world Class

The following are used to create a new World object

```
World (int noOfRows, int noOfColumns)
    -creates a World with the given rows and columns

World (String worldFilename)
    -creates a World from a text file. This method can be used to create maze worlds.
    Characters in the text file determine the size and contents of the world. For example:
    '1' – wall          '0' – empty square  's' – starting point  'e' – exit
    'A' – 'Z' character Items (red)
    See sample files for proper file format.
```

The following methods allow you to add Robots and Items to a World

```
void addRobot (Robot robotToAdd, int row, int column,
               Direction direction)
    -adds a Robot to the world in the given row and column and facing in the given
    direction

void addRobotAtStart (Robot robotToAdd)
    -adds a Robot to the world at the start (start is indicated by an "s" in the world
    text file)

void addItem (Item itemToAdd, int row, int column)
    -adds an Item to the world in the given row and column
```

F.2 The Robot Class

The following constructors are used to create a new Robot object:

```
Robot (String name)
    -creates a Robot with the given name and default colour (blue)

Robot (String name, Color colour)
    -creates a Robot with the given name and colour
```

Robot's direction and basic movements:

```
boolean isFacing (Direction direction)
    -checks if the robot is facing in the given direction

void turnLeft ()
    -turns the robot left 90°

void turnRight ()
    -turns the robot right 90°
```

void move ()
-moves the robot forward one square

When moving around a Robot World, the robot can check the World for various objects:

boolean isWallAhead ()
-checks if there is a wall directly ahead of the robot

boolean isRobotAhead ()
-checks if there is another robot directly ahead of the robot

boolean isMarkerAhead ()
-checks if there is a marker directly ahead of the robot

boolean isMarkerHere ()
-checks if there is a marker on the robot's current square

boolean isExitHere ()
-checks if there is an Exit on the robot's current square

boolean isItemHere ()
-checks if there is an Item object on the robot's current square

The robot can drop coloured markers and pick up and drop Item objects (see Items on next page) using the following commands:

void dropMarker ()
-drops a marker (the same colour as the robot) on the robot's current square

void pickUpItem ()
-picks up an item on the robot's current square

void dropLastItem ()
-drops the last item picked up by the robot onto the robot's current square

void dropFirstItem ()
-drops the first item picked up by the robot onto the robot's current square

boolean isLastItem (Item checkItem)
-checks if the last item picked up by the robot is the same as the checkItem

int compareLastItemToItemHere ()
-compares the last item picked up by the robot to the Item on the robot's current square. Used mainly to compare character Items. Returns the following:
 Last Item < Item Here returns a value < 0
 Last Item = Item Here returns a value of 0
 Last Item > Item Here returns a value > 0

int getNoOfItems ()
-returns the number of items currently held by the robot

F.3 The Direction Class

The following constants are used to indicate directions e.g. `Direction.EAST`

```
static final Direction EAST  
static final Direction SOUTH  
static final Direction WEST  
static final Direction NORTH
```

If you want to pick a random direction, you can use the following static method

```
static Direction random()  
    -returns a random direction (EAST, SOUTH, WEST or NORTH)
```

F.4 The Item Class

Constants used to define different types of items e.g. `Item.CIRCLE`

```
static final int CIRCLE  
static final int SQUARE  
static final int TRIANGLE
```

Constructors used to create different Item objects

```
Item (int type, Color colour)  
    -creates a circle, square or triangle Item with the given colour  
Item (char ch)  
    -creates a single character item with the default colour (black)  
Item (char ch, Color colour)  
    -creates a single character item with the given colour  
  
int compareTo(Item compareItem)  
    -compares two Items. Used by compareLastItemToItemHere ()
```

Appendix G -- Exceptions and Exception Handling

The Java language reports run-time errors by throwing exceptions. An exception is a special type of object that reports on certain run-time errors. To handle an exception you write code that "catches" these "thrown" exceptions and then deals with them. For example, the following section of code inputs an integer. If the user inputs an invalid integer, he/she is given a chance to re-enter:

```
Scanner keyboard = new Scanner (System.in);
boolean isValidInteger;
do
{
    try
    {
        System.out.print ("Please enter an integer number: ");
        number = Integer.parseInt (keyboard.next());
        isValidInteger = true;
    }
    catch (NumberFormatException e)
    {
        System.out.println ("Error - Not a legal integer, please re-enter");
        isValidInteger = false;
    }
}
while (!isValidInteger);
```

In this example, the section of code in the curly braces after the `try` statement is called the try block. The computer tries to run each statement in the try block. If one of these statements, throws an exception (causes an error) the program immediately jumps out of the try block and looks for a `catch` statement that catches the exception just thrown.

For example, if the user entered "34g", this would be read OK as a string but when `parseInt()` tried to convert this to an integer, a `NumberFormatException` would be thrown. The first `catch` block can catch a `NumberFormatException` (indicated in the parentheses after the keyword `catch`) so, when this exception is thrown, program control is passed to the first line in this `catch` block. This would cause the error message to be printed and `isValidInteger` would be set to `false`. After completing the catch block, the program would jump to the bottom of the last catch block. Since `isValidInteger` is `false` the program would then repeat the outer `do while` loop, giving the user a chance to re-enter.

You can have many catch blocks following a try block to catch and deal with different types of exceptions. If your catch blocks do not catch a particular exception it will be passed along (re-thrown) for some other section of code to deal with (catch). In most cases you should deal with exceptions as soon as possible.

On the other hand, if the user had input a valid integer, no exception would be thrown and all of the statements in the try block would have been completed, setting `isValidInteger` to `true`. When the try block is successfully completed, control passes to the first statement after the last catch block. In this case, the program would then exit the outer `do while` loop since you have entered a valid integer.

The `NumberFormatException` in the example above is an unchecked exception. In Java, you don't have to write code to deal with unchecked exceptions. If, however, this exception gets thrown a message will be reported in an error window and the program may stop running. For small practice problems you don't have to deal with unchecked exceptions, however when you are writing code that will be used by others, you should anticipate and deal with all exceptions.

Along with unchecked exceptions, we have checked exceptions. If a section of code can throw a checked exception, you must write code to deal with this exception. For example, the `Thread.sleep()` method, which is used to delay your program for a certain number of milliseconds, throws an `InterruptedException` which you must deal with. You could catch this exception, as shown below: (In this example we do nothing in the catch block).

```
try
{
    Thread.sleep (3000);
}
catch (InterruptedException e)
{
}
```

or you could pass this exception along by modifying your main heading by adding "throws `InterruptedException`" as shown below:

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

In the second case we would not need to put the call of the method `Thread.sleep()` in a try block since we are not catching the exception we are just passing it along.

We can also throw exceptions in our own code if we want to report on errors. For example, when we are writing our own methods, if the value of a parameter sent to a method is out of range we may throw an `IllegalArgumentException`.

References

Adams, Joel, Nyhoff, Larry, and Nyhoff, Jeffery
Java, An Introduction to Computing
Prentice Hall, Upper Saddle River, NJ 2001

Bloch, Joshua
Effective Java, Programming Language Guide
Sun Microsystems, Inc, Palo Alto, CA 2001

Burnette, Ed
Eclipse IDE Pocket Guide
O'Reilly, Sebastopol, CA 2005

Flanagan, David
Java In A Nutshell, 2nd Edition
O'Reilly, Sebastopol, CA 1997

Horstmann, Cay S. and Cornell, Gary
Core Java 2, Volume 1- Fundamentals
Sun Microsystems Press, Palo Alto, CA 2001

Hume, J. N. Patterson and Stephenson, Christine
Introduction to Programming in Java
Holt Software Associates, Toronto, ON 2000

Naughton, Patrick and Schildt, Herbert
Java 2: The Complete Reference, 3rd Edition
Osborne/McGraw-Hill, Berkeley, CA 1999

Oracle's Java web site including the Java help files and tutorials
The latest version of the Java api can be found at:

<http://docs.oracle.com/javase/8/docs/api/>

The latest versions of the Java tutorial can be found at:

<http://docs.oracle.com/javase/tutorial/>

Schneider, G. Michael and Gersting, Judith L.
An Invitation to Computer Science, Java Version
Brooks/Cole, Pacific Grove, CA 2000

van der Linden, Peter
Just Java 5th Edition
Sun Microsystems Press, Palo Alto, CA 2002