Introduction to Programming with Java

By G. Ridout

Version 2.4

Table of Contents

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/OAC 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 I have taught over the past six years 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.

Gord Ridout
July 2006

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.

Using these assembly language mnemonics made the program code more programmer friendly. However, since the computer still only understands binary machine language, before you can run a program written in assembly language you must translate (**assemble**) it into machine language using a special program called an **assembler**. In most cases each assembly language instruction translates one to one into a 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 (C)
If the total sales are greater than $100, calculate a discount that is 10% of these sales.	`if (totalSales > 100)` `  discount = totalSales * 0.10;`

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 C statement would translate into more than 4 machine language instructions. Therefore, when translating high-level languages into machine language is it is important to select a compiler that produces good 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++, Java and C#.

1.2 The Java Programming Language

The Java language was developed at Sun Microsystems in 1991. 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 on Sun's web site, the Java language is described as follows:

> "Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language"[1].

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 been used in smaller systems such as PDA's and cell phones. You can even program a Lego Mindstorms RCX with Java.

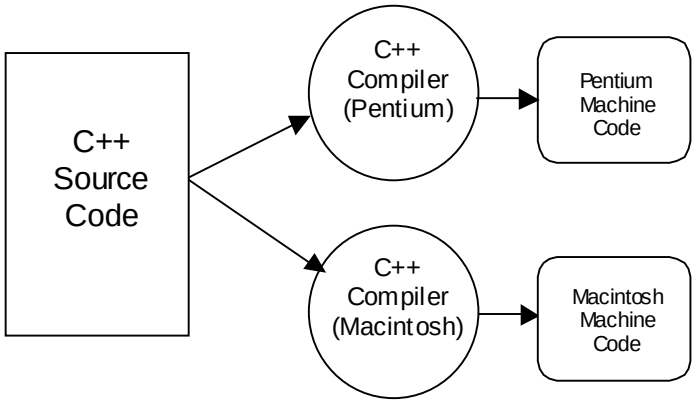
---

[1] http://java.sun.com/docs/overviews/java/java-overview-1.html
This paper explains how Java demonstrates each of these characteristics.

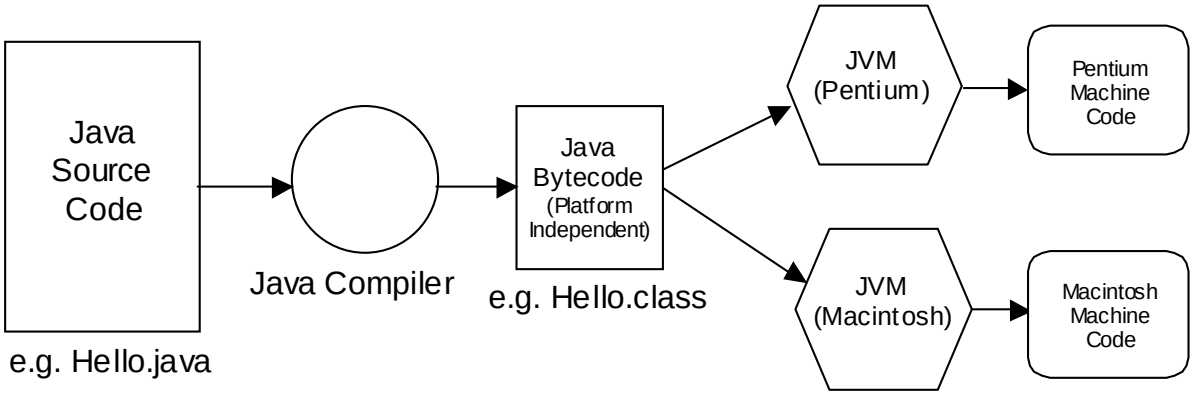
1.3 Translating Java Code to Machine Code

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. There are even Java virtual machines available for small devices such as PDA's, cell phones and the Lego Mindstorms RCX.

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 is platform independence.

1.4 The JDK and the Ready to Program with Java IDE

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 5.0) can be downloaded for free from Sun Microsystems (`java.sun.com`). Once you have downloaded and properly installed the JDK, you will need a text editor (such as notepad) to create your source code (`.java` file). Then you compile your source code to bytecode using the following command:

```
javac MyJavaProgram.java
```

This will create a bytecode file called `MyJavaProgram.class`. To run this class file you need to pass it to the JVM with the following command:

```
java MyJavaProgram
```

Notice the `.class` extension is not required when using the JVM. If there are any errors in your program, you will need to go back to the editor to change the code and then repeat the two steps to compile and then 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 but we will be using the Ready to Program with Java IDE developed by Holt Software Associates. The Ready IDE was chosen because it does not require a powerful computer with a lot of RAM to run and it is very easy to use.

The Ready to Program with Java IDE will be used to edit, compile and run your Java programs. Everything is included in one package, so if you install Ready you do not need to download or install the JDK.

As an editor, Ready offers all the normal text editor options plus syntax colour highlighting that makes it easier to read your Java programs. Without exiting the Ready program we can select the Run option (F1) to compile and run your program all in one step. At the compile stage, Ready will also find and highlight any syntax errors.

1.5 Chapter Summary

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

assemble
assembler
assembly code
binary
bytecode
class
compile
compiler
inheritance
IDE
JDK
JVM
machine code
method
mnemonic code
object code
OOL
OOP
platform
source code

1.6 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) Name 3 non object-oriented high-level languages.

6) Name 3 object-oriented languages.

7) Acronyms (a word formed from the first letters of other words) were used to name languages such as BASIC and COBOL. However other languages do not use acronyms in their name. . Do some research to find out where other languages such as Ada, Pascal, C, C++, Turing, Smalltalk and Java got their names.

8) Do some research to find out more about the Java language including what its original name was?

9) What is JVM short for?

10) What is bytecode and how is bytecode different than machine code?

11) 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++).

12) What is IDE short for? What does an IDE do?

13) There are different editions of the JDK available (J2SE, J2EE and J2ME). What are the different versions used for? This will require some research.

Chapter 2 - Getting Started in Java

In Java, one of the easiest ways to output information to the screen or to input information from the keyboard is to use a `Console` object created using the Holt Software Associates Console class. Even though the `Console` class is not part of the standard Java language it is included in the Ready to Program with Java IDE. One of the strengths of the Java language is that we can easily add in third party classes.

2.1 A Simple Java Program

Load up the Ready IDE and type in the following Java program. Since you are going to be modifying this program to make it your own, include your own name and today's date in the introductory comments. Java is case sensitive, so make sure you type the upper and lower case letters exactly as shown:

```
import hsa.Console;

/** The "ConsoleOutput" class.
  * Purpose: Displays a Welcome Message
  * @author put your name
  * @version put date here
 */

public class ConsoleOutput
{
    public static void main (String [] args)
    {
        Console myConsole = new Console ("First Java Program");
        myConsole.println ("Welcome to Java");
    } // main method
} // ConsoleOutput class
```

When you are done, select the `SaveAs option` in the `File` menu and save your code using the default file name: `ConsoleOutput.java`. In Java, the file name of your source code must include the name of the `public` class in your program (e.g. `ConsoleOutput`) followed by the `.java` extension. The Ready IDE will usually pick the correct name for you. In fact, if you try to save your code under a different name you will get a warning.

Now try running this program by pressing the Run button in the toolbar or by pressing F1. If the program reports any error message, check your code carefully and fix any typing mistakes. The line of code that contains the error is usually highlighted and an error message is given at the bottom of the code window. Before you ask for help, please read the error message carefully.

Before we begin looking over the above program line by line, 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 (more on this later) when creating our programs.

Now, let's look over the code you just typed in.

The first thing you may notice is the syntax colouring (highlighting) that is part of the Ready IDE. The following summarizes some of this colouring:

- Program comments appear in green.
- Java keywords such as **`import`**, **`public`**, **`class`** appear in **boldface**.
- Java class library identifiers such as `String` appear in black.
- Any punctuation (`{ } or ;`) also appears in black.
- All of your identifiers or any third party identifiers appear in blue.
  (e.g. `Console` and `myConsole`)
- `String` or `char` constants (e.g. `"Welcome to Java"`) appear in red.

This syntax colouring can be very helpful, especially when you are trying to find errors in your code.

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

```
import hsa.Console;
```

The above statement is included because we are using the `Console` class in our program. This statement tells the compiler to look for the `Console` class code in the `hsa` 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. To find out which package a new class belongs to, check the online help.

For example, if you look up the `Color` class in online help you will see that it is called `java.awt.Color`. This is the full name of the class. This also tells us that the `Color` class is part of the `java.awt` package. The `awt` is short for "abstract window toolkit", a simple Java graphics toolkit. To use the `Color` class we will need to include the statement: **`import`** `java.awt.Color`.

If you want to use more than one class in a package, you can use a wild card. For example: `import awt.*` lets us use any classes in the `awt` package.

Instead of using an import statement you can write the full name of a class in the code. For example, without an import statement, we would need to write `hsa.Console` instead of `Console` and `java.awt.Color` instead of just `Color`. 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.

You may have also noticed the semi-colon (`;`) at the end 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 use to putting a semi-colon at the end of each statement. However, you must be careful because, as you will see below, not every line in your Java program requires a semi-colon.

```
/** The "ConsoleOutput" class.
  * Purpose: 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 ConsoleOutput
```

The above line of code is the heading for the `ConsoleOutput` class. In Java we uses classes to describe the framework for objects. Notice that there is no semi-colon (`;`) at the end of this line since this is a heading. Since a program is an object every program needs at least one class. Since the `ConsoleOutput` 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, remember the name of your program file must match the name of the public class. One final point, when naming classes you should begin with a capital letter and then use 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 `ConsoleOutput` class (our program). At the bottom of the code, the } indicates the end of the `ConsoleOutput` 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`. We will talk about these in Chapter 5. Just remember, every main program you write will start with this same heading. Once again, since it is a heading, there is no semi-colon at the end of this line.

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.

```
Console myConsole = new Console ("First Java Program");
```

This line of code actually does two things. We could break it down into two separate lines:

```
Console myConsole;
```

This first line of code declares a reference to a `Console` object. The reference is called `myConsole`. When we want to output to or input from this `Console` we will use the name `myConsole` to refer to it. In this line of code the word `Console` describes the type of object we will be referring to. Notice the identifier that is used to describe the `Console` object has a similar format to the class name identifier. The only difference is that it begins with a small letter. This is another Java convention. This notation where all of the letters are small except for the first letter of each word after the first word is sometimes called camel notation. (See Appendix A for more detail on naming identifiers in Java).

```
myConsole = new Console ("First Java Program");
```

This line of code creates a Console object. Whenever we want to create a new object in Java we use the **new** command. In this line of code the word `Console` calls a special method called a **constructor** that creates the `Console` object. When this statement is run the Console window appears on your screen. This newly created object is sometimes referred to as an **instance** of the `Console` class. The `"First Java Program"` string is a **parameter** to the `Console` constructor that tells the constructor the title to put at the top of the window. You can also tell the constructor how many rows and columns to put in the Console window and which font to use.

We broke this statement into two separate lines to help explain how it works, but generally we want to do both things at the same time so usually a single line of code is all that is required. Since this is a complete Java statement, we need to put a semi-colon at the end of this statement.

```
myConsole.println ("Welcome to Java");
```

Finally, we come to the main body of program. In this case we are telling the computer to print the words `"Welcome to Java"` on the screen in the Console window we just created. `println()` (short for print line) is the method (action) we are going to perform on the `myConsole` object. We separate the object's name from the method name using a period (`.`). This is known as "dot notation". The message `"Welcome to Java"` is a parameter to the `println` method telling it what to print. Since we said `myConsole.println` the message is printed in the `myConsole` window. Once again, we need a semi-colon at the end of the line to complete the statement.

```
    } // main method
} // ConsoleOutput class
```

As mentioned earlier, the last two lines close off the main method and the `ConsoleOutput` 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 Ready IDE will automatically indent and line up your code when you press F2. If, after pressing F2, your code is not lined up, your code probably has an error or an extra blank line.

It seems like a lot for such a simple program, but this program 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 must ask questions or seek extra help.

Now that you understand the above code, let's try adding some new statements. Since we will be using the `Color` class in these statements you need to add the statement:

```
import java.awt.Color;
```

at the top of your program under the other import statement.

Note: When writing Java code you must spell colour without the 'u' i.e. Color.

Now try these modifications:

1) Before the `println()` statement, add the following line of code:

```
myConsole.setTextColor(Color.GREEN);
```

Run your program and see how the output changed. Try some other colours. If you type in a legitimate colour name the text will change from blue to black in the Ready IDE. Java is case sensitive so the word `Color` must have a capital 'C' and the constant colour names (e.g. `ORANGE`) are in uppercase.

2) To change the background colour, try the following statement:

```
myConsole.setTextBackgroundColor(Color.BLUE);
```

Since this only changes the back ground colour of the text you output using `println()`, you may want to add the statement: `myConsole.clear()` after the `setTextBackgroundColor` command to change the background of the entire window all at once.

3) Now try adding this line of code, also before the `println`. You can leave in the earlier modifications.

```
myConsole.setCursor (3, 10);
```

One again, run your program and note how the output changed.

The `setCursor` method has two parameters (3 and 10 in the above example). By changing these parameters, try to figure out what each parameter represents.

4) Next, try the following statements, one at a time:

```
myConsole.setCursor (3, 75);

myConsole.setCursor (3, -10);
```

What happens in each of the above cases? How about:

```
myConsole.setCursor (3.5, 10);
```

Why doesn't this statement work? Later you will be introduced to a method called `drawString()`, which gives you a little more control over where you place your text.

2.2 The Console Class

In most cases your programs should only have one console for input and output. However, to demonstrate some different features of the Console class, the following program uses two consoles. Load up the Ready IDE and type in the following Java program. To save time and space, the introductory comments for this program have been left off. Don't forget to save your program.

```
// Working with more than one console
import hsa.Console;

public class OutputWithTwoConsoles
{
    public static void main (String [] args)
    {
        // Create a console with 15 rows and 30 columns
        Console firstConsole = new Console (15, 30, "First Console");
        firstConsole.println ("This is the first console");
        firstConsole.print ("No of Rows: ");
        firstConsole.println (firstConsole.getMaxRows ());
        firstConsole.print ("No of Columns: ");
        firstConsole.println (firstConsole.getMaxColumns ());
        firstConsole.println ("Press any key to continue");
        firstConsole.getChar ();

        // Create a console with 10 rows and 40 columns
        // All text in this console will use a 16-point font
        Console secondConsole = new Console (10, 40, 16, "Second Console");
        secondConsole.println ("This is the second console");
        secondConsole.print ("No of Rows: ");
        secondConsole.println (secondConsole.getMaxRows ());
        secondConsole.print ("No of Columns: ");
        secondConsole.println (secondConsole.getMaxColumns ());
        secondConsole.println ("Press any key to continue");
        secondConsole.getChar ();
        secondConsole.close ();

        firstConsole.println ("Back to the first console");
        firstConsole.println ("Program is complete");

    } // main method
} // OutputWithTwoConsoles class
```

Now try running this program by pressing the Run button in the toolbar or by pressing F1. If the program reports any error message, check your code carefully and fix any typing mistakes. Before reading ahead, look over the code and the resulting output and see if you can figure out what each of the statements does.

Now, let us look at some of the new statements presented above.

```
Console firstConsole = new Console (15, 30, "First Console");
```

Like before, the above statement declares `firstConsole` as a reference to a `Console` object and then creates the `Console` object. In this example the `Console` constructor has 3 parameters. (See Appendix C for a complete list of all of the versions on the `Console` constructor). The new parameters 15 and 30 tell the constructor to create a console window with 15 rows and 30 columns. If you don't include these parameters the default size for the console window is 25 rows and 80 columns.

```
firstConsole.println ("This is the first console");
firstConsole.print ("No of Rows: ");
firstConsole.println (firstConsole.getMaxRows ());
firstConsole.print ("No of Columns: ");
firstConsole.println (firstConsole.getMaxColumns ());
```

These lines output a message that you are using the first console and then they tell you the number of rows and columns in this console window. You may have noticed that the second line above uses `print` instead of `println`. Looking at the output you should be able to figure out the difference between these two methods. The expressions `firstConsole.getMaxRows()` and `firstConsole.getMaxColumns()` call the Console methods `getMaxRows()` and `getMaxColumns()` to find out the number of rows and number of columns in the `firstConsole` object. Each of these methods has no parameters so there is nothing between the `()`.

```
firstConsole.println ("Press any key to continue");
firstConsole.getChar ();
```

These two statements ask for and wait for any key to be pressed. The method `getChar()` waits for any character to be entered before continuing on. Again this is a method with no parameters. When you press a key the value of the key pressed is just ignored and the program continues. If we wanted to find out which key you pressed, we would have to store that key using a statement like: `keyPressed = firstConsole.getChar ();`. Then the value of the key you pressed would be stored in the variable called `keyPressed`.

```
Console secondConsole = new Console (10, 40, 16, "Second Console");
```

This line of code creates a second console. In this example, the constructor has 4 parameters. The new parameter 16 gives the font point size to be used for all of the text in the console window.

```
secondConsole.println ("This is the second console");
secondConsole.print ("No of Rows: ");
secondConsole.println (secondConsole.getMaxRows ());
secondConsole.print ("No of Columns: ");
secondConsole.println (secondConsole.getMaxColumns ());
secondConsole.println ("Press any key to continue");
secondConsole.getChar ();
```

The above statements are the same as the ones described earlier except they deal with the `secondConsole` object instead of the `firstConsole` object. So when you call a method such as `println()` or `getMaxRows()` it is important to include an object name in front of the method to indicate which console you are referring to.

```
secondConsole.close ();
```

This statement closes the second console window.

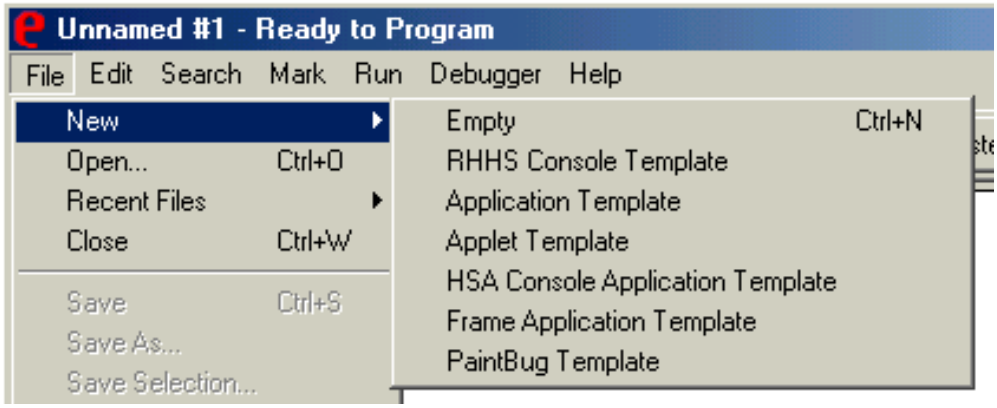
```
firstConsole.println ("Back to the first console");
firstConsole.println ("Program is complete");
```

These statements print a final pair of message back in the first console.

Once you understand the above statements, change the background and text colours in both of the Console windows. Choose any colours you like but make the colours different in each window.

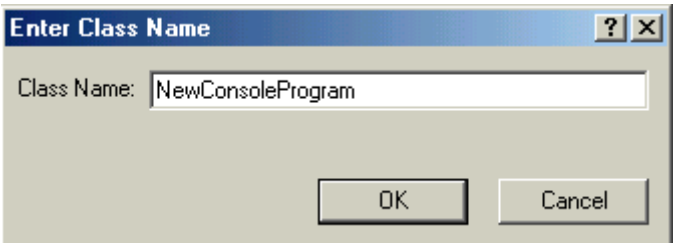
2.3 Using Templates to Create New Console Programs

To save time when writing programs that include a HSA Console object for input and output, the Ready IDE has some built in templates (partly written shells of a program to get you started). Load up the Ready IDE and select the `New` option from the `File` menu. You should see the following on your screen:



If you are working on your home computer, the RHHS Console Template may be missing. You can get an updated `new_menu.ini` file from the class web page that includes this extra template. You can also modify the `new_menu.ini` file to create your own templates.

Using your mouse, click on the `RHHS Console Template` option or the `HSA Console Application Template`. You will then be presented with a window asking for the class name of your program. Type in the name: `NewConsoleProgram` and then select OK.



You should then get the following code in your editor window. If you select the `HSA Console Application Template`, you will get a slightly different version.

```
import java.awt.Color;
import hsa.Console;

/** The "NewConsoleProgram" class.
  * Purpose:
  * @author
  * @version
  */
public class NewConsoleProgram
{
   public static void main (String[] args)
    {
        Console c = new Console ("NewConsoleProgram");

    } // main method
} // NewConsoleProgram class
```

This template includes the required `import` statements to use both the `Console` and `Color` classes as well as the introductory comments section and the `class` and `main` method definitions. Using these templates can save you a few keystrokes. The one disadvantage of using a template is that you don't know what the included code means. That is why we went over this code before templates were introduced.

One other change you may have noticed in this program is that the variable which refers to the console has been shortened to just the letter 'c'. Normally we want to use descriptive names for our object variables, but since we will only have one console in each program and we will be using it a lot, using the single letter is allowed in this case.

To use the newly created Console 'c' in the above program, try typing in the following statements under the Console declaration:

```
c.setTextColor(Color.PINK);
c.setCursor(10, 20);
c.println("Templates are easy");
```

2.4 The Color Class

When you were setting the text colour and background colour you may have found the following 13 built in colours:

```
Color.BLACK    Color.BLUE     Color.CYAN          Color.DARK_GRAY
Color.GRAY     Color.GREEN    Color.LIGHT_GRAY    Color.MAGENTA
Color.ORANGE   Color.PINK     Color.RED           Color.WHITE
Color.YELLOW
```

Using the Color class we can create a new Color object with a choice of over 16 million different colours.

Create a new program using the RHHS or HSA Console Application Template. Add a title to the statement that creates the Console and then add the following lines of code to the program.

```
Color myNewColour = new Color (200, 100, 180);
c.setTextColor (myNewColour);
c.println ("Text in a new Colour");
```

The first line of code creates a new `Color` object referred to as `myNewColour`. The 3 parameters to the `Color` constructor indicate the colour in RGB format. In RGB format, the Red, Green and Blue components of a colour are each represented by an integer in the range 0-255. A value of 0 indicates that there will be no red, green or blue in the new colour and a value of 255 indicates that there will be the maximum intensity of red, green or blue in the new colour. Therefore `new Color (0, 0, 0)` will give you black and `new Color (255, 255, 255)` will give you white. The above combination gives a light purple colour.

It is not always necessary to create a reference to a new Color object in order to use a new colour. Instead of the first two lines of code above, we could have written the single line:

```
c.setTextColor (new Color (200, 100, 180));
```

This creates a temporary object that is then used by `setTextColor` to set the colour of the text. This is the preferred method, if you won't be re-using this colour in your program. Just like there were different versions of the Console constructor, there are different version of the `Color` constructor. If you are interested, check out the online help.

Try playing around to create different colours for both the text colour and the background colour.

2.5 Chapter Summary

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

awt	hsa Console	package
templates	indenting (paragraphing)	parameter
constructor	instance	program comments
dot notation	main	return value
execute	method	RGB

Java key words/syntax introduced in this chapter:

`{ and }`	`new`
`;`	`public`
`class`	`static`
`import`	`void`

Methods introduced in this chapter:

Console methods

`clear()`	`print()`
`close()`	`println()`
`Console() - constructors`	`setCursor()`
`getChar()`	`setTextBackgroundColor()`
`getMaxColumns()`	`setTextColor()`
`getMaxRows()`	

Color methods

`Color()` - constructor

2.6 Questions, Exercises and Problems

1) 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.

2) In Java, what is the purpose of the `import` statement? Can you use a `Color` object in a Java program without including an `import` statement? Explain.

3) Give and explain at least two advantages of syntax highlighting in the Ready IDE.

4) What is a parameter? Why do we use parameters with methods?

5) What are the advantages of using templates when creating new programs?

6) Each of the following statements constructs a different `Console` window. In each case, describe what the new window will look like.

```
Console myConsole = new Console("Example 1");
Console myConsole = new Console(20, 40, "Example 2");
Console myConsole = new Console(20, 40, 14, "Example 3");
Console myConsole = new Console(15);
```

7) Write the Java code to declare and create a `Console` window called `newConsole` that has 15 rows and 60 columns and has the text "My Program" in the title bar.

8) Describe what would happen if you ran the following section of code. Be specific.

```
Console first = new Console ("First Console");
Console second = first;
first.println ("Dog");
second.println ("Cat");
```

9) When creating new colours using the `Color` constructor, it was mentioned you have a choice of over 16 million colours. Where do you think this number comes from? What is the exact number of colours available?

10) Modify the program in section 2.1, so that it uses `getMaxRows` and `getMaxColumns` to write your name in the centre of the Console window in red text with a cyan background. If you change the size of your console window your program should automatically adjust to the new size keeping your name in the centre of the window.

11) We can also draw graphics in a Console window using the Console graphics methods given in Appendix C.3. When using these methods the window becomes an x and y coordinate grid with the top left corner of the window being position (0,0). The x and y coordinates are in pixels. For example, `drawRect(50, 100, 30, 60)` will draw a 30 pixel by 60 pixel rectangle 50 pixels over and 100 pixels down from the top left corner of the window. The `drawRect` method draws the outline of a rectangle and `fillRect` draws a solid rectangle. The `setColor` method is used to set the drawing colour. Using the graphics methods in C.3, write the Java code to draw the following. Each graphic should be in a separate Java program.

a) A happy face.
b) A Canadian flag.
c) A simple house with doors and windows.
d) Be creative and create your own original picture.

Chapter 3 - Working with Variables in Java

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.

3.1 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.

We saw in the last chapter when we declared and created a Console object we used the statement:

```
Console myConsole = new Console ("First Java Program");
```

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 4 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 Unicode $2^{16}-1$	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
`double`	64 bits	$\pm 10^{-324}$ to 10^{308}	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 Ready, the names of primitive types appear in bold.

Declaring variables:

```
boolean gameOver;
char  answer;
int  noOfStudents;
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 = c.readInt();
```

Which type of variable you will 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 GST_RATE = 0.06;
```

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*GST_RATE` is clearer than: `priceWithTax = price + price*0.06`. 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.

3.2 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:

```
String name;
```

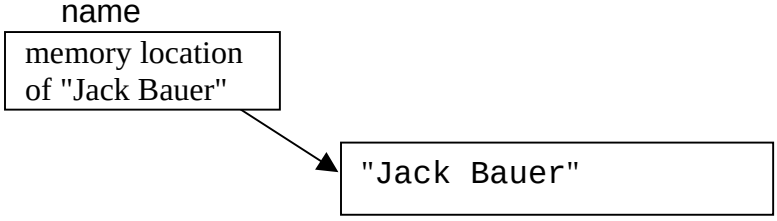
Then to create the `String` object that keeps track of the actual name, we need to use the **new** command. For example:

```
new String("Jack Bauer");
```

Since we want `name` to refer to this new `String` object, we usually combine the above two statements into the following:

```
String name = new String("Jack Bauer");
```

The following diagram represents what is happening in memory:



As you can see, the above code is very similar to the code we used in the last chapter to declare and create a `Console` object. Like `Console` objects, `String` objects have their own set of methods and properties. 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 `String`s. When you type the above code into Ready, 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. We will talk more about `String` objects in Chapter 6.

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

3.3 Using the Console 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. Using the RHHS or HSA Console Application Template to get you started type in the following Java program:

```
// The "ConsoleInputOutput" class.

import hsa.Console;

public class ConsoleInputOutput
{
    public static void main (String [] args)
    {
        Console c = new Console ("Input and Output");

        // Input the information from the keyboard
        // We also declare each new variable as needed
        c.println ("Console Input");
        c.print ("Please enter your name: ");
        String name = c.readLine ();
        c.print (name + ", please enter an integer number: ");
        int myInteger = c.readInt ();
        c.print (name + ", please enter a real number with decimals: ");
        double myDouble = c.readDouble ();
        c.print ("Press any key to continue ");
        char myChar = c.getChar ();
        c.clear ();

        // Output the information into the console window
        c.println ("Console Output");
        c.print ("User Name: ");
        c.println (name);
        c.print ("The integer number entered was: ");
        c.println (myInteger);
        c.print ("The double number entered was: ");
        c.println (myDouble);
        c.print ("The character entered was: ");
        c.println (myChar);
        c.println ("End of Program");

    } // main method
} // ConsoleInputOutput class
```

Now try running this program by pressing the Run button in the toolbar or by pressing F1. If the program reports any error message, check your code carefully and fix any typing mistakes. Before reading ahead, look over the code and the resulting output and see if you can figure out what each of the statements does.

Now lets look at some of the new statements added since Chapter 2. The first thing you may have noticed is that we declared the variables `name`, `myInteger`, `myDouble` and `myChar` 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.

```
c.print ("Please enter your name: ");
String name = c.readLine ();
```

To read in a string variable we must first ask for the information. The first line above is called a prompt for input. On the next line of code we declare a `String` variable called `name` and then read in this name using the `readLine()` 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 Console "c" we use `c.readLine()`. We used `readLine()` instead of `readString()` because a person's name usually includes two words and `readString()` reads in one word only. For the prompt we used `print()` instead of a `println()` so that the name is entered beside the prompt. Change the `print()` to a `println()` and you will notice the difference.

```
c.print (name + ", please enter an integer number: ");
int myInteger = c.readInt ();
c.print (name + ", please enter a real number with decimals: ");
double myDouble = c.readDouble ();
```

To read in an integer we use `readInt()` and to read in a double we use `readDouble()`. Like `readLine()`, `readInt()` and `readDouble()` 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"`, you would get `"Marcus, please enter an integer number: "`.

```
c.print ("Press any key to continue ");
char myChar = c.getChar ();
c.clear ();
```

This will read in a single character. By using `getChar()` instead of `readChar()`you don't need to press enter after typing in the single character. This is a nice way to read in single characters.

```
c.println ("Console Output");
c.print ("User Name: ");
c.println (name);
c.print ("The integer number entered was: ");
c.println (myInteger);
c.print ("The double number entered was: ");
c.println (myDouble);
c.print ("The character entered was: ");
c.println (myChar);
c.println ("End of Program");
```

These statements will output the information that was input. We can also add some formatting to our output (see the next section). See also Appendix C for a complete list of the methods in the Console class.

3.4 Formatting Console Output

Starting with the program in section 3.3, make the following changes to the section that outputs the results. (The changes are in bold)

```
c.println ("12345678901234567890");
c.print ("User Name: ", 15);
c.println (myInteger, 5);
c.println (myDouble, 6, 1);
```

Run the program and see if you can notice any changes. Generally the numbers you added define field sizes to help format your output. Within each field, strings are left justified and numbers are right justified. The first `println()` above gives you the column positions to help you check the spacing. When printing doubles the second number is the number of decimal places you want. If there are more decimals in the number than you select, the displayed number is automatically rounded.

If you used formatting with Turing, it is a similar set up. Change the numbers above a few times and try entering different numbers to get an idea how the formatting works. Notice that if you pick a field size too small for the number, it is just ignored. Since you will want to format your output in your own programs, it is important that you understand how this formatting works. So take a few minutes to play around with this program, entering different data with different field sizes.

3.5 Arithmetic Expressions in Java

Arithmetic expressions are used to manipulate `int`, `double` 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: `+` `–` `*` `/` and `%` (add, subtract, multiply, divide and modulo)
`++` `--` (Increment and decrement)
`+=` `-=` `*=` `/=` (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, then subtract
`(4 + 3) * (7 - 9)` will add and subtract, then multiply

NOTE: You must be careful if you are mixing types in arithmetic expressions. 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 multiple 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 3.6, 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.

Shorthand Expression	Code Equivalent	Meaning
`x += y`	`x = x + y`	Add y to x
`x -= y`	`x = x - y`	Subtract y from x
`x *= y`	`x = x * y`	Multiply x by y
`x /= y`	`x = x / y`	Divide x by y

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
`(3+7)*(4-9)`	`-50`	`int`
`3.0+7*4-9`	`22`	`double`
`56 / 15`	`3`	`int`
`56 % 15`	`11`	`int`
`2 / 3 + 4 / 5`	`0`	`int`
`(2 * 15 + 7) % 5`	`2`	`int`
`2.3 * 7 - 20.5`	`-5.3`	`double`

The following is an example using the `++` and `+=` operator. The output of the following section of Java code, is shown to the right:

```
int firstNumber = 17;
int secondNumber = 9;
firstNumber++;
c.println(firstNumber);
secondNumber += 25;
c.println(secondNumber);
```

Output

```
18
34
```

To further manipulate both numbers and characters, you can use methods in the `Math` class and the `Character` class. See Appendix D for more information on these classes and their methods.

3.6 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 visa 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. 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

```
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 roll 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.

We will also use casting when using some of the `Math` class methods. To keep things simple we are using only 4 of the 8 primitive types available, but methods such as `Math.round()` return a `long` type (a 64-bit integer) which we haven't talked about. Therefore if we use `Math.round()` we need to cast the result to `int` before storing it in an integer variable.

For example to round `doubleVar` to the nearest integer we would use:

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

We will also be using a cast when we want to generate a random integer. For example to pick a random integer between 1 and 10 inclusive we would do the following:

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

In this case `Math.random()*10 + 1` returns a double value between 1.0 and 10.999… inclusive. When we cast this result to `int` we get an integer number between 1 and 10 inclusive.

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 use this feature regularly when creating output strings for the `println()` method. More information on converting to and from strings is presented in section 6.7.

3.7 Chapter Summary

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

arithmetic expressions	increment operator	promotion
assignment compatible	initialize	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:

Console methods

getChar()	readDouble()
print()	readInt()
println() with formatting	readLine()
readChar()	readString()

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()	

3.8 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) `58 / 6`
b) `31 % 10`
c) `17 * 2 + 3 * 4.5`
d) `3.0 + 117 % 7`
e) `14 + 18 / 4`
f) `4.2 * 3`
g) `(int) Math.round(3.78)`
h) `Math.max(15, 14.2)`
i) `Math.sqrt(7)`
j) `Math.min(12, 7)`
k) `Math.pow(3,2)`

10) Given an integer variable called count, show 4 different ways to add 1 to count.

11) Write the necessary Java code to generate a random integer number between –5 and 17 inclusive (including both –5 and 17).

12) Given the following declarations, which of the following assignment statements are allowed in Java. If a statement is not allowed, explain why.

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

a) `intVar = doubleVar;`
b) `doubleVar = intVar;`
c) `charVar = intVar;`
d) `charVar = doubleVar;`
e) `strVar = doubleVar;`
f) `doubleVar = charVar;`
g) `strVar = charVar;`
h) `charVar = strVar;`
i) `intVar = (int) doubleVar;`
j) `doubleVar = (double) strVar;`

13) Without using the computer, predict the exact output of the following sections of Java code. Make sure you show the exact spacing in your answer.

```
Console c = new Console ();
String name = "Sam Cooke";
int count = 76;
double number = 912.3456;
c.print ("Name: ", 10);
c.println (name);
c.println (count, 5);
c.println (number);
c.println (number, 8, 2);
c.println (number, 4, 1);
```

14) Using an RHHS or HSA Console Application Template to start, enter the following Java program:

```
Console c = new Console ("Working with Integers");
int number = 2147483647;
c.println (number);
number++;
c.println (number);
```

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

15) Write a program that reads in the length and the width of a rectangle and then calculates and displays the area.

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

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

17) 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.

18) 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 rounded to two decimal places. Extra: Give your answer in hours and minutes (rounded to the nearest minute).

19) A store clerk is interested in a program that calculates the exact change to give a customer with the minimum number of bills and coins. You should input the amount of the purchase and the amount of money tendered (given to the clerk) and then output the change required and the number of each bill and coin to make up the change.

Remember you want the minimum number of coins and bills in each case. You can assume the total change will be less than $20. A sample run of the program is shown below. Don't forget to plan out this program first, before writing the code.

```
                    Change Making Program

Please enter the total purchase: $1.42
Please enter the amount tendered: $20.00
The change will be: $18.58
To make up this amount you will need:
     1 ten-dollar bill
     1 five-dollar bill
     1 two-dollar coin
     1 loonie
     2 quarters
     1 nickel
     3 pennies

Thank you for using the Change Making Program
```

20) Create your own program based on the ideas covered in this chapter.

Chapter 4 - 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.

4.1 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
==	a == b	a equals b
<	a < b	a less than b
>	a > b	a greater than b
<=	a <= b	a less than and equal to b
>=	a >= b	a greater than and equal to b
!=	a != b	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) and (number >= 10 && <= 15)

4.2 Selection Structures

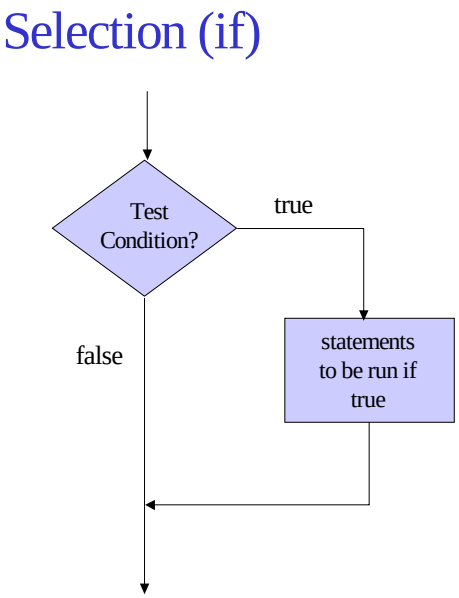
Sometimes in our programs we only want to execute a certain statement if a certain condition is first met. For example, if we wanted to count the number of students with honours (a mark of 80 percent or above), we could use the following section of Java code:

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

Note that the boolean expression (condition) is between parenthesis. 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 your `if` structures to make them easier to follow. Fortunately, the Ready IDE will automatically indent your code if you press F2. If pressing F2 fails to properly indent your code, look for extra spaces or errors in your code.

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



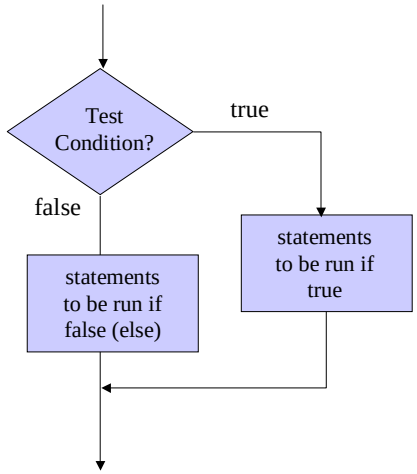
Normally, only the first statement after the `if` condition is executed when the condition is true. However, if we want to include more than one statement inside of our `if`, we can group statements into a compound statement using curly braces.

For example:

```
if (mark >= 80)
{
    c.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:



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

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

In this example, the second set of curly braces are 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.

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

```
if (temperature  >= 30)
   c.println("It is a very hot day");
if (temperature < 30 && temperature  >= 20 )
   c.println("A nice summer day");
if (temperature < 20)
   c.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)
   c.println("It is a very hot day");
else if (temperature  >= 20)  // and temperature < 30
   c.println("A nice summer day");
else // temperature < 20
   c.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:

```
if (condition)
    statement_TRUE;
```

```
if (condition)
    statement_TRUE;
else
    statement_FALSE;
```

```
if (condition_1)
    statement_1 TRUE;
else if (condition_2)
    statement_1 FALSE, 2 TRUE;
else
    statement_BOTH 1 & 2 FALSE;
```

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.

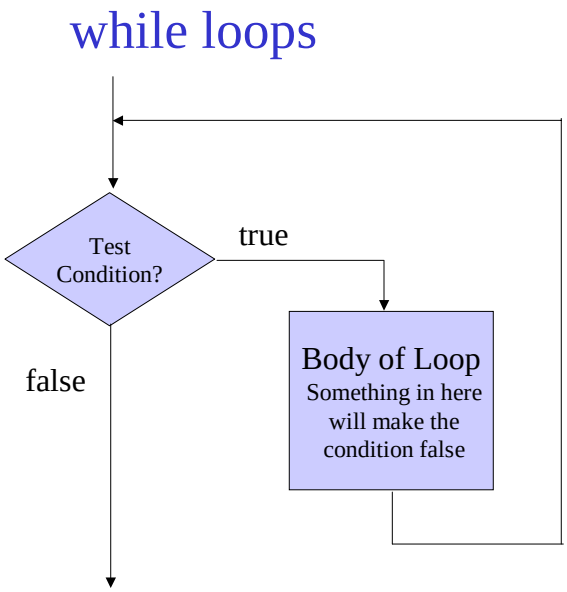
In Java we have 3 possible types of loops. We will group these into two basic types, conditional loops and counter loops.

4.3 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



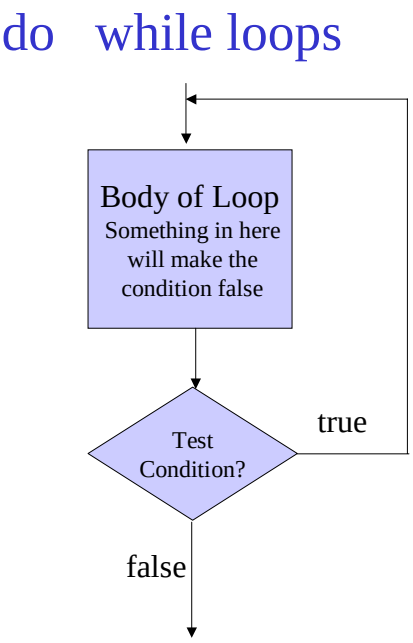
The Java code for the `while` loop is given below:

```
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:



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.

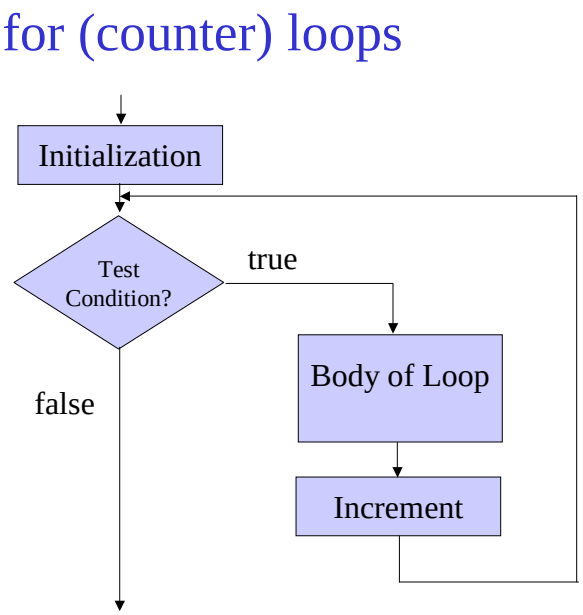
As you can see these two loops are very similar. Which loop you use will depend on the situation. If you have a situation where you are going to do something at least once then you should probably use a `do while` loop. On the other hand, if you have a situation where you may never enter the loop in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

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 <u>all</u> Java loops tells the computer when to <u>continue the loop</u>, instead of when to exit the loop.

4.4 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



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++)
{
   c.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.

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
}
```

4.5 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.

In this program there are two error check loops to check the user's input. The first error check is used to check if the `typeOfConv` input is valid (1 or 2), while the second error check tests if the `tryAgain` input is valid ('y' or '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. Later in the course we will introduce more advanced error checking techniques to handle these types of errors (see Appendix G for an example).

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. If we declared this variable when it was first used its scope would be limited to the inside of the loop in which it was declared. This would cause a problem since this variable is used in the while condition which is outside of the loop.

```
import hsa.Console;

/** The "TemperatureConversion" class.
  * Purpose: To convert a temperature from Fahrenheit to Celsius or
  *          from Celsius to Fahrenheit
  * @author G. Ridout
  * @version February 2001
 */
```

```
public class TemperatureConversion
{
    public static void main (String [] args)
    {
        Console c = new Console ("Temperature Conversion");
        double celsiusTemp, fahrenheitTemp;
        char tryAgain;

        c.println ("Welcome to the Temperature Conversion Program");

        // Loop to process each conversion
        do
        {
            // Inputs the type of conversion you would like to do
            // \u00b0 is the Unicode for the degrees symbol
            c.println ("Enter 1 to convert from \u00b0F to \u00b0C");
            c.println ("Enter 2 to convert from \u00b0C to \u00b0F");
            c.print ("Enter your Choice: ");
            int typeOfConv = c.readInt ();

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

            // 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)
            {
                c.print ("Enter the temperature in degrees Fahrenheit: ");
                fahrenheitTemp = c.readDouble ();

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

                c.print (fahrenheitTemp, 0, 1);
                c.print ("\u00b0F is equal to ");
                c.print (celsiusTemp, 0, 1);
                c.println ("\u00b0C");
            }
            else
            {
                c.print ("Enter the temperature in degrees Celsius: ");
                celsiusTemp = c.readDouble ();

                fahrenheitTemp = celsiusTemp * 9 / 5 + 32;

                c.print (celsiusTemp, 0, 1);
                c.print ("\u00b0C is equal to ");
                c.print (fahrenheitTemp, 0, 1);
                c.println ("\u00b0F");
            }
```

```
            // Ask if the user would like to try again
            // Include an error check to see that answer is y or n
            c.print ("Would you like to convert another temperature (y/n)? ");
            do
                tryAgain = Character.toLowerCase (c.getChar ());
            while (tryAgain != 'y' && tryAgain != 'n');
            c.println (tryAgain);  // Included because we are using getChar()
        }
        while (tryAgain == 'y');

        c.println ("Thank you for using the Temperature Conversion Program");
    } // main method
} // TemperatureConversion class
```

4.6 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
compound boolean expressions	exception	run time error
compound statement	flowchart	selection
condition	increment	syntax error
conditional loops	initialization	test data
control structures	logic error	test plan
	program efficiency	

Java key words/syntax introduced in this chapter:

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

4.7 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)
   c.print("1");
else if (first > second && second >= third)
   c.print("2");
else
   c.print("3");

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

Predict the output of this section of Java code, given the following values of the `first`, `second` and `third`:

a) `first = 7`, `second = 12` and `third = 7`
b) `first = 18`, `second = 13` and `third = 5`
c) `first = 34`, `second = 24` and `third = 24`

3) 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.

a)
```
for (int count=1; count<=5; count++)
{
   double product = 2.22*count;
   c.print(count, 3);
   c.println(product, 6, 1);
}
```

b)
```
int first = 45;
int second = 10;

c.println ("Working with While Loops");
while (first > second)
{
   c.print(first, 5);
   c.println(second, 5);
   first -=5;
   second +=6;
}
c.print(first, 10);
c.println(second, 10);
c.println("That's all folks");
```

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

```
if (noOfDimes == 1)
   c.println("1 dime");
if (noOfDimes > 1)
   c.println(noOfDimes + " dimes");
```

```
if (noOfDimes == 1)
   c.println("1 dime");
else if (noOfDimes > 1)
   c.println(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;
}
c.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 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 Console;
class ErrorProgram
{
    public static void main ()
    {
        Console c = new Console (Program with Errors);
        double mark;
        int passCount;
        c.println ("Counting the passes and failures");
        do
            c.print ("Please enter a mark (Enter -1 to quit): ");
            mark = c.readInt ();
            if (mark < 50)
               passCount++;
            else
               failCount++;
        while (mark = -1);
        println ("Number of passing marks: " + passCount);
        c.println ("Number of failures: " + failCount);
        c.println ("Total marks entered: " + passCount + failCount);
        c.println ("Thank you for using this program");
    } // main method
} // ErrorProgram class
```

8) Assume 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 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 fields to display the numbers in nice columns (see section 3.4). 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 either 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.

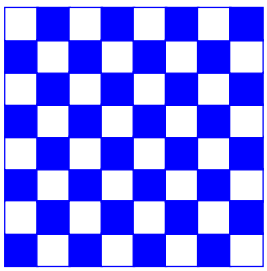
14) Orion Entertainment sells CD's for $13.99 and DVD's for $19.99. If the total number of items purchased by a customer is 10 or more, the customer receives a 10% discount on his or her entire purchase. Both provincial tax (8%) and GST (6%) are then added to the price after the discount to calculate 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 PST and GST (show both taxes separately) 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 both tax rates should be stored as constants (final) defined at the beginning of the program. For example: `final double CD_PRICE = 13.99;`

15) Write a program that draws the following checkerboard pattern on the screen. In your checkerboard, make the dark and light squares different colours. To draw each square in the console, you can use the method `fillRect()` described in Appendix C. Note: In console graphics the upper-left corner of the console window is position (0,0).



Chapter 5 - Methods

5.1 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.

For example, if you have a Console object called `myConsole` you can use the `println()` method to print to `myConsole`. Since we need to tell the computer what object to do the printing with, we never call the `println()` method by itself. We have to associate this method with an object using dot notation. For example:

```
myConsole.println ("Message in myConsole");
```

In this case, we are getting the `myConsole` object to output the message. The particular message we want to be displayed is sent as a parameter (data in parentheses after the name of the method).

In Java, every method is contained within a class. Each class can have static methods (also know as class methods) and instance methods. To use an instance method we must first create an instance (object) of that class. To create an object we call a special method called a constructor. The class constructor has the same name as the class. Since we haven't created the object yet we don't use dot notation when calling a constructor. For example:

```
Console myConsole = new Console ("Creating a Console object");
```

In this example the first "`Console`" tells you what type of object you are creating and the second "`Console()`" is the constructor method that creates the object. The parameter is the title of the console window. Methods (including constructors) can have more than one parameter or in some cases no parameters.

To send information into a method we use parameters. If the method wants to send information back to the calling section of code, it can return a value. For example:

```
age = myConsole.readInt()
```

In this case the `readInt()` method returns an integer value that is then stored in a variable called `age`. Since you don't need to pass any information to the method when using `readInt()`, there are no parameters. Even with no parameters you still need to include the `()` to indicate that `readInt()` is a method and not a variable.

If you look in the Java on-line help you will see that there are many different classes available to help us solve a variety of programming problems. To help us use these classes there are many different methods within each class. As part of the on-line help description, method headings are provided to help us identify the return type and parameters for each method. For example, the headings for the two methods given above are:

```
public void println(String str);
public int readInt();
```

The first part of the heading tells us that this is a public method that is available for use outside of the class. To save on space, if no modifier is given you can assume that it is `public`. The second part of the heading tells us what information the method will return. Since the `println()` method returns nothing we put the word `void` in front of the method name. On the other hand, since `readInt()` returns an integer we put the word `int` in front.

The heading also lists each of the parameters and their type. For example, for one of the versions of `println`, there is a single parameter that is a String. The word `str` is the name of the parameter. In most cases, the name of the parameter helps describe what the parameter should be. For example, the heading for another version of println is:

```
void println (int number, int fieldSize)
```

In this case we have two integer parameters. The names of the parameters tell us that the first parameter is the number and the second parameter is the field size. This allows us to display an integer within a certain size field (column). If you look in appendix C, you will see there are many more versions of `println`. For example:

```
void println(int i)           // displays an integer
void println(double d)        // displays a double
void println(String s)        // displays a String
```

The difference between each version is the number and type of the parameters. Since the code to display a String is different than the code to display an integer, we need different methods. Instead of using different method names such as `printString()` and `printInteger()`, Java allows us to use the same name. The Java compiler then determines which version of the method to call based on the number and type of the parameters. Re-using the same method name for different versions of a method is known as **overloading**. By overloading methods, we have fewer method names to remember.

In addition to instance methods, some classes contain static methods. These **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. 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:

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

Since `sqrt()` is a static method, we use the name of the class instead of the name of an object to call it up. Since you do not need to create an object to use them, static methods are similar to the procedures and functions used in procedural (non object-oriented) languages. To help identify static methods in the on-line help, the `static` modifier is included in the method heading.

5.2 The main method

Now that we know a little more about methods, let's look at the main method that is part of every Java application:

```
public static void main (String [] args)
{
    // Your code goes here
} // main method
```

To review, the first line of the method is called the method heading. The words `public` and `static` are know as **modifiers**. The `public` means that you can call this method outside the class and `static` means we don't have to create an instance of your program object to use this method. Remember each program you create is a new class.

The `void` is the return type and it means that the main method returns nothing. Next is `main` which is the name of your method. Every application in Java needs a main method to tell the computer where to get started.

Following in the parenthesis is your list of parameters (called formal parameters by some). In this case, the parameters to main are an array of Strings. These are used to pass command line parameters to your program. Since we are not running our programs from the command line we don't usually use these parameters. The following simple example demonstrates how to use the parameters passed to main by displaying all of the command line arguments in a System window.

```
// The "MainParameters" class.
public class MainParameters
{
    public static void main (String [] args)
    {
        // Display each of the command line arguments (parameters)
        for (int i = 0 ; i < args.length ; i++)
            System.out.println (args [i]);

    } // main method
} // MainParameters class
```

To test this program in Ready you can select the `Run with Args...` option in the `Run` menu to enter the command line arguments.

5.3 Writing Your Own 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. For the area of a triangle method we can summarize the method inputs and outputs as follows:

Method Inputs (Parameters)	Method Output (Return Value)
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.

```
// The "TriangleArea" class.
import hsa.Console;

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)
    {
        Console c = new Console ("Triangle Area");
        c.println ("Finding the area of a triangle");

        c.print ("Please enter the base of the triangle: ");
        double enteredBase = c.readDouble ();
        c.print ("Please enter the heightof the triangle: ");
        double enteredHeight = c.readDouble ();

        double triangleArea
            = areaOfTriangle (enteredBase, enteredHeight);

        c.print ("The area of the triangle with base: ");
        c.print (enteredBase, 0, 1);
        c.print (" and height: ");
        c.print (enteredHeight, 0, 1);
        c.print (" is: ");
        c.print (triangleArea, 0, 1);
        c.println (" units" + '\u00b2');

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

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 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 `enterHeight` 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) 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 are negative
 */
private static double areaOfTriangle (double base, double height)
{
    if (base < 0 || height < 0)
        throw new IllegalArgumentException
                  ("Base and Height must be greater than zero");
    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 or deal with the resulting exception accordingly.

5.4 Method Overloading

We saw earlier that we can overload a method by using the same name but a different number or type of parameters. 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.
import hsa.Console;

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)
    {
        Console c = new Console ();

        c.println (max (5, 12));
        c.println (max (4.6, 7.9));
        c.println (max (78, 34.7));
        c.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.

5.5 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);
```

5.6 Chapter Summary

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

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

Java key words/syntax used in this chapter

`@param`	`return`
`@return`	`static`
`@throws`	`throw`
`IllegalArgumentException`	`String [] args`
`private`	`void`
`public`	

5.7 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) Given the following method heading, explain what each part of this heading means:

```
public void drawRect(int x, int y, int width, int height)
```

6) We have already discussed the modifiers `public`, `private` and `static`. Using the on-line help, list at least three different modifiers that you found. Explain what each of these modifiers means.

7) Write a simple Java program that adds 1 or more integer numbers. The integer numbers should be given to your Java program as command line arguments.
Hint: You will need to use the `parseInt()` method in the Integer class.

8) What is the difference between an actual parameter and a formal parameter? What is an argument?

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. 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 `drawChar()` that draws a character on the screen in a certain location in a certain colour.

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

```
import hsa.Console;
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 (Console mc, char ch, int times)
    {
        for (int count = 1; count <= times; count++)
            mc.print(ch);

        mc.println();
    }

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

    public static void main (String [] args)
    {
        Console c = new Console ("Trace Program");
        c.println (average(5, 10));
        c.println (average (3.5, 5.5));
        c.println (average (7.0, 6));
        c.println (abs(-3));
        c.println (abs(4.5));
        printChar (c, '+', 3);
        printChar (c, 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) A method called `drawDiamond()` that draws a small (20 pixels by 20 pixels) red diamond in different locations in a console window. For example, `drawDiamond(c, 50, 50)` should draw a red diamond in the Console c with the upper left corner in position (50, 50). Hint: Use the `drawPolygon` method (see Appendix C.3).

18) A method called `drawHappyFace()` that draws a happy face in a Console window. The happy face should be approximately 100 × 100 pixels. Parameters should be included to indicate the Console and the position within the Console where to draw the happy face. For example:

```
drawHappyFace(c, 200, 50);
```

would draw a happy face in Console c with the upper-left corner at position (200, 50). Hint: see Appendix C.3 for Console graphics methods such as `fillOval` and `drawArc`.

19) For a positive integer n, the notation `n!` (read as 'n factorial') is used for the product `n×(n-1)×(n-2)...×2×1`. For example `4!=4×3×2×1=24` and `10!=10×9×8×7×6×5×4×3×2×1=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`.

20) 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.

21) 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.

22) 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").

23) 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 21) to find the largest prime number with this property. Note: The answer is less than `Integer.MAX_VALUE`, the maximum integer value.

Chapter 6 - 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 everytime 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.

6.1 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.

6.2 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:

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

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

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

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

6.3 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";
c.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 = "abcdefg";
c.println(sentence.charAt(3)) // would output d
c.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 an `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 sample 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++;
}
c.println("There are " + spaceCount + " spaces in " + sentence);
```

6.4 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);
```

6.5 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
```

6.6 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 = name.toUpperCase();
```

`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
c.println(str.toUpperCase());

// to output: this is a mixed case string
c.println(str.toLowerCase());

String str = "   Extra spaces          ";
// to remove the extra spaces and store the result back in str
str = str.trim()
```

For a list of other String methods, check out the Java on-line help.

6.7 Converting Other Types of Variables to 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 stringVar = "The number is: " + number;
```

`number` will be converted to a string before joining it with "`The number is: `" to create the new string "`The number is: 1234`".

If we want to convert a primitive type to a string, without concatenating it to another string, 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 output intNumb using drawString
c.drawString(String.valueOf(intNumb), 100, 100);

// to output realNumb using drawString
c.drawString(String.valueOf(realNumb), 200, 200);
```

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.

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 on-line help documentation. Some objects include a constructor that allows you to create a new object from a `String`.

6.8 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())
{
   c.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");

c.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())
   c.print(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.

6.9 Chapter Summary

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

concatenate	immutable	substring
delimiter	index	token
exceptions		

Methods introduced in this chapter:

String methods

charAt()	String() - constructor
compareTo()	substring()
equals()	toLowerCase()
equalsIgnoreCase()	toUpperCase()
indexOf()	trim()
lastIndexOf()	valueOf() - static
length()	

Integer and Double methods

parseInt()	parseDouble()

StringTokenizer methods

countTokens()	nextToken()
hasMoreTokens()	StringTokenizer() - constructors

6.10 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:

```
String enteredPassword;
do
{
   c.println("Please enter a password");
   enteredPassword = c.readString();
}
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.

```
Console c = new Console ();
//                          1         2         3         4         5
//                 012345678901234567890123456789012345678901234567890123456789012
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         ";

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

c.println (firstStr.charAt (4));
c.println (secondStr.charAt (35));

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

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

5) Given the following section of code:

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

 if (firstStr.compareTo (secondStr) > 0)
    c.println ("firstStr is greater than secondStr");
 else if (firstStr.compareTo (secondStr) < 0)
    c.println ("firstStr is less than secondStr");
 else
    c.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);
}
c.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:	Name Input	Name Output
	Sue Johnson	Johnson, Sue
	Prince	Prince
	Billy Bob Joe Carson	Carson, Billy Bob Joe

Note: There should no `println` or `readLine` statements in the following methods. All input and output should happen 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 an other 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 "`CAACCCCDCD`" 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
`sizeOfLargestGroup ("xxyyXxxxxx")`	5
`sizeOfLargestGroup ("AAACCCBBBCCCBBBAA")`	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 <u>even</u> numbered digits of the number beginning with the second digit from the left. For example:

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

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 <u>odd</u> numbered digits in the original number. For example:

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

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`.

Chapter 7 - Arrays and Simple Classes

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];
```

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 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 [] namesOfStudents = 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 `namesOfStudents` array created above we could use the following:

```
for (int index=0; index < namesOfStudents.length; index++)
   c.println(namesOfStudents [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.

```
public static void main (String [] args)
{
    Console c = new Console ("Marks Program");

    // Find out how many marks are to be entered and set up the array
    c.println ("Welcome to the Marks program");
    c.print ("How many marks will you be entering?: ");
    int noOfMarks = c.readInt ();
    int [] marks = new int [noOfMarks];

    // Enter the marks, totalling the marks as they are entered
    int totalMark = 0;
    c.println ("Please enter the marks");
    for (int markNo = 0 ; markNo < marks.length ; markNo++)
    {
        c.print ("Please enter mark #" + (markNo + 1) + ": ");
        marks [markNo] = c.readInt ();
        totalMark += marks [markNo];
    }
    c.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;
    c.print ("The average mark is: ");
    c.println (average, 0, 1);

    c.println ("\nThe following marks are above the average");
    for (int markNo = 0 ; markNo < marks.length ; markNo++)
    {
        if (marks [markNo] > average)
        {
            c.println (marks [markNo]);
        }
    }

    c.println ("End of Marks Program");
} // main method
```

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 intializer 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 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 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 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:

```
c.println("List of Students Above the Class Average");
c.println("    Name             Mark");
for (int student = 0; student < marks.length; student++)
   if (marks[student] > classAverage)
   {
      c.print(names[student], 20);
      c.println(marks[student], 3);
   }
```

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`).

7.5 Simple Classes (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. This class consists of two public data fields for the student's name and mark and a public constructor method to create a new Student object. The class code is given below:

```
public class Student
{
   public String name;
   public int mark;

   public Student(String newName, int newMark)
   {
      name = newName;
      mark = newMark;
   }
}
```

By making the data in this class public we are creating a simplified class that is more like a C structure type or a Turing record type than a traditional Java class. Normally, to make your Java classes more versatile, you should make all of the data fields private and then include public accessor methods (e.g. `getName()`) to access these private data fields.

To use this new class we first create a single array of Student objects with the following statement:

```
Student [] listOfStudents = new Student [noOfStudents];
```

To create a new student object we use the `new` command that calls the Student constructor. For example:

```
listOfStudents[0] = new Student("Chan, Kitty", 85);
```

Once this student object has been created, we can refer to individual data elements within the object using dot notation since both the `name` and `mark` data fields are public. For example, to refer the student's name we would use: `listOfStudents[0].name` and to refer to the mark we would use: `listOfStudents[0].mark`

The complete code for the program that uses this class to display the names of students above the average is given below:

```
c.println("List of Students Above the Class Average");
c.println("    Name             Mark");
for (int student=0; student < listOfStudents.length; student++)
   if (listOfStudents[student].mark > classAverage)
   {
      c.print(listOfStudents[student].name, 20);
      c.println(listOfStudents[student].mark, 3);
   }
```

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;
```

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:

`ArrayIndexOutOfBoundsException`	initializer list
element	pass "by reference"
index	subscript

Methods and variables introduced in this chapter:

Array methods and variables

`length`

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) Explain how you could use arrays and a simple class to keep track of the above information about the basketball players for the entire season. You should expand your class to keep track of not only the points per game for each player but also the number of assists and rebounds for each game.

4) Given the following Java code, predict the output of the given statements:

```
boolean [] used = new boolean [8];
```

a) `c.println(used [3]);` b) `c.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) `c.println(numbers.length)` b) `c.println(numbers[4])`

c)
```
for (int i = 0; i < numbers.length; i++)
   if (numbers[i] % 2 == 0)
      c.print (i, 5);
```

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];
```

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)
```
c.println(matrix[2][3]);
```

b)
```
double total = 0;
for (int i = 0; i < 3; i++)
   total += matrix[1][i];

c.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];

c.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 be true since 2000 is a leap year
`isValidDate(2006,9,31)` would be false since September has only 30 days
`isValidDate(2007,13,6)` would be 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) `generateArray` b) `displayArray` c) `averageOfArray`
d) `indexOfSmallest` e) `sortArray` f) `mergeArrays`

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.

```
import hsa.Console;

public class ArrayMethods
{
    // The code for all of the methods (including comments) goes here

    public static void main (String [] args)
    {
        Console c = new Console ("Using Methods with Arrays");

        // Title and Introduction
        c.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
        c.println ("Here are the numbers: ");
        displayArray (c, firstList);

        // Find and display the average of the numbers in the array
        c.print ("The average of the numbers in the above array is: ");
        c.println (averageOfArray (firstList), 0, 1);

        // Find and display the index and value of the smallest number
        int index = indexOfSmallest (firstList);
        c.print ("\nThe index of the smallest number is: ");
        c.println (index);
        c.print ("The smallest number in the above list is: ");
        c.println (firstList [index], 0, 2);

        // Sort and then display the array
        sortArray (firstList);
        c.println ("\nHere is the sorted array: ");
        displayArray (c, 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
        c.println ("Here is the merged array: ");
        displayArray (c, mergedList);

        // Closing Remarks
        c.println ("The Using Methods with Arrays 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 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.

a) Every month each customer is charged a $2.00 account fee. Add this $2.00 fee to each of the customer's balances.

b) Display a list (with proper column headings) of the names and balances of all customers who have a balance greater than $200.00

c) 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 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          11.3%
            T           90           9.8%
            A           85           9.2%
            .            .             .
            .            .             .
            .            .             .
            Z            1           0.1%
            X            0           0.0%

         Totals        920         100.0%

End of File Analysis
```

Since the default `Console` window only contains 25 rows, you will need to make a larger window to fit the results for the letters A to Z including the totals. If you forgot how to construct a larger `Console` window, see appendix C.1.

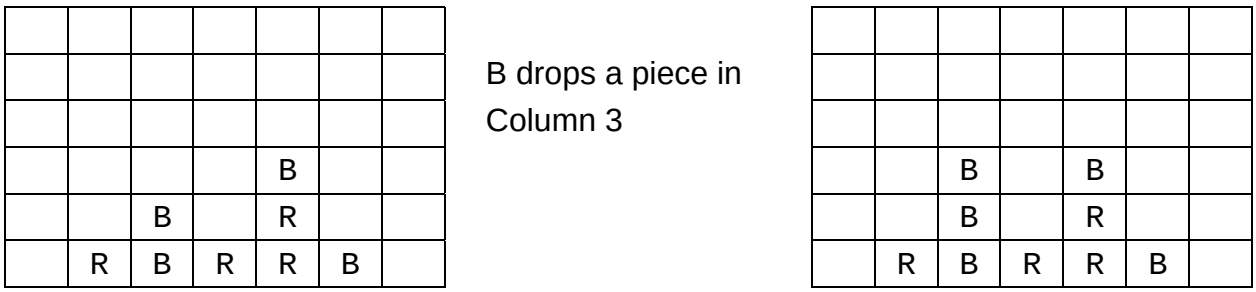
Plan out your code carefully before beginning. I would suggest you break the problem into two parts:

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 an array of a simple class that keeps track of each letter and its frequency.

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.



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  TTTFFTTFFTTFTTFFTTTTFFTTFFFTTFFTTFF
Row 2  TTTTTFFTTTTFFFFFTTFFTTTTTTTTTTTTTTT
Row 3  TTTTTFFFFFFFFTTTTFFFFFTTFFTTTTTTTTT
```

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.

Chapter 8 - Robot World

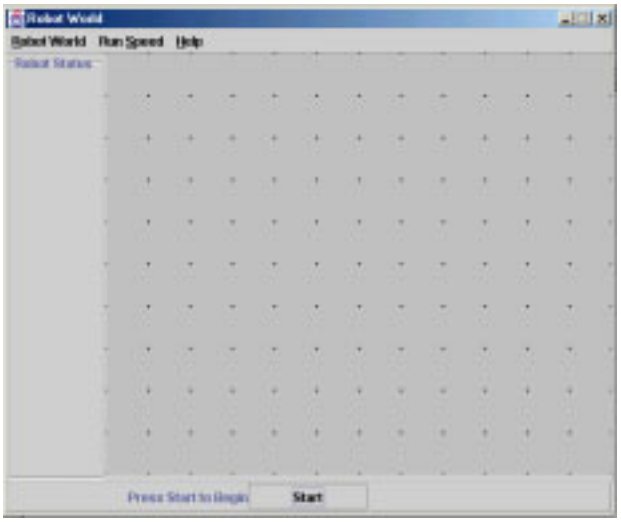
The Robot World package is a collection of classes developed by G. Ridout in the fall of 2001. Robot World was based on/inspired by the University of Waterloo's JKarel robot used in the first year Computer Science courses. The latest version of the Robot World classes is available at the following web site: http://www.richmondhill.hs.yrdsb.edu.on.ca/ics3m/robot/. This web site also includes instructions on how to set up the Ready IDE to use the Robot World classes in the robot package. This package can also be used in other environments by setting the proper class path. In all cases you will need to include the statement `import robot.*` in your Java program to make this package available.

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. Creating a `World` object is very similar to creating a `Console` object. 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.



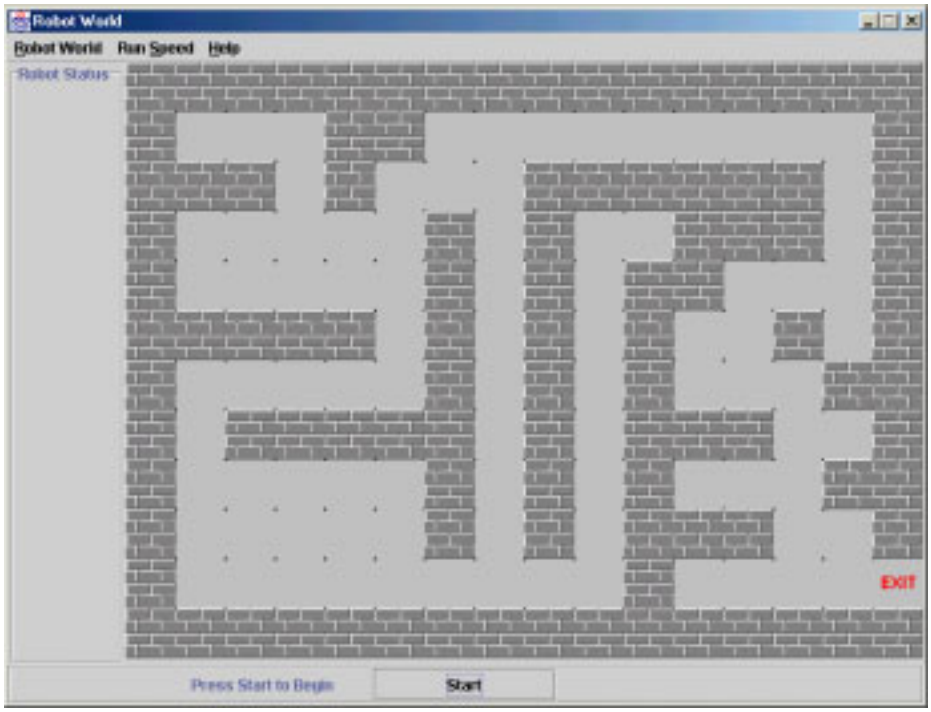
The main part of the Robot World is the grid area in which the robot moves. In this example, this area is the 10 rows high (numbered from 0 to 9) and 12 columns wide (numbered from 0 to 11). The rows and columns are numbered starting with 0 to match the Java array indices. The smallest possible Robot World is 7 by 7 and the largest possible world is 24 rows by 34 columns. The large grid size shown above is used for Robot Worlds up to 12 rows and 16 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 now 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:

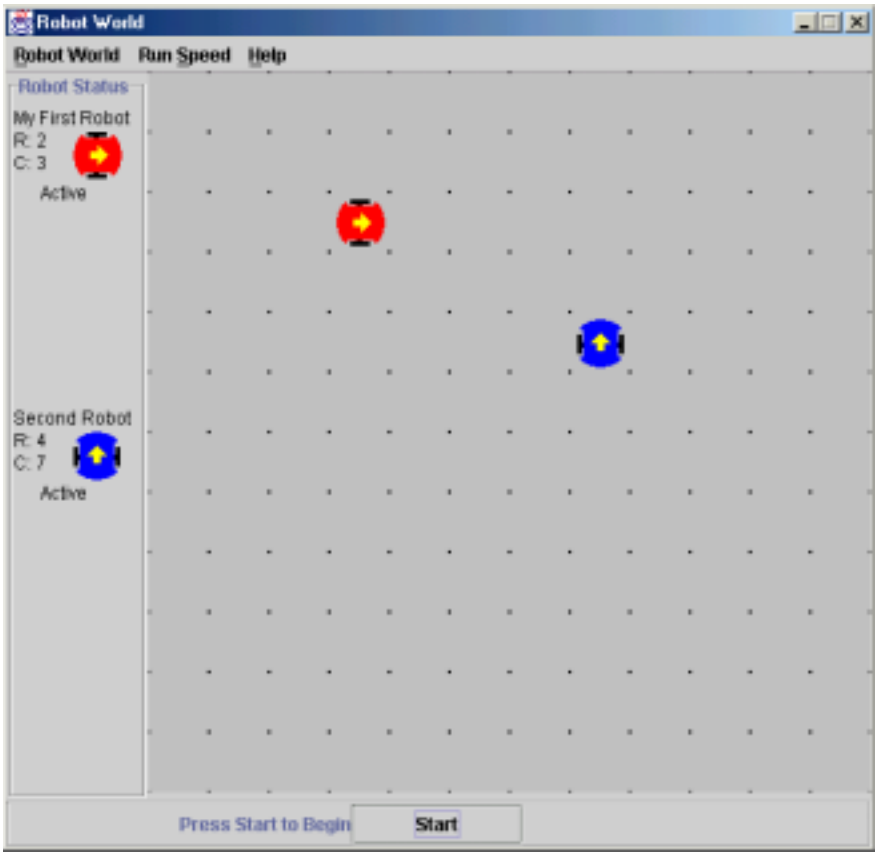
```
myWorld.addRobot (myRobot, 2, 3, Direction.EAST);
```

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 the web site given earlier at the start of this chapter). 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, 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.

You may want to try changing this code to see how it affects the robot's movements. If the robot moves two slowly you can change the speed of the robot by selecting the "Run Speed" menu option before you start your robot. It is possible to change the run speed while the robot is moving but on some machines this can cause the program to hang so you should pause the robot before changing speeds. If you are working on an older machine, the robot may still be too slow, so you can add the following statement right 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 ("Scanvenger", 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 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 180º, 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 ()
    {
        turnLeft ();
        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 turnLeft(). Normally, when we call the turnLeft() method, we use dot notation to indicate which robot we want to turn left. But since these methods are called within the RobotPlus class, the computer assumes that you want the same robot to turn left that called the turnAround() method in the first place. Therefore, myNewRobot turns left 2 times to complete the turn.

Although not necessary in this case, to specify that we wanted "this" Robot (myNewRobot) to turn left we could have used the statement this.turnLeft(). The "this" qualifier is optional in this example, 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.

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 `RobotPlus`: Example Code

The following example shows the modified code for the `RobotPlus` class that would allow your robot to keep 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.

```
public class RobotPlus extends Robot
{
    // New private data fields to keep track of the Robot's movements
    private int changeNS;
    private int changeWE;


    /** 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)
    {
        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;
    }
```

```
    /** Moves the robot and keeps track of its change in position
      * This method "overrides" the move() method in the Robot class
      */
    public void move ()
    {
        // Calls the original "Robot" move
        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++;
    }


    /** Turns the robot to face a given direction
      * @param direction The direction to face the robot
      */
    public void turnToFace (int direction)
    {
        while (!isFacing (direction))
            turnRight ();
    }


    /** 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);
        while (changeNS != 0)
            move ();

        // Adjust the west/east position
        if (changeWE > 0)
            turnToFace (Direction.WEST);
        else
            turnToFace (Direction.EAST);
        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.

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

`compareLastItemToItemHere ()`
`dropFirstItem ()`
`dropLastItem ()`
`dropMarker()`
`getNoOfItems ()`
`isExitHere ()`
`isFacing ()`
`isItemHere ()`
`isLastItem()`
`isWallAhead ()`
`move ()`
`pickUpItem ()`
`Robot()` - 2 constructors
`turnLeft ()`
`turnRight ()`

`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.

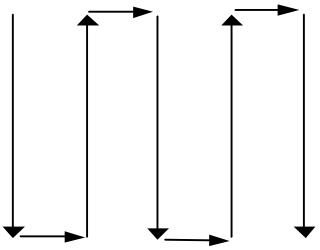
3) Write a Java program to do the following:

a) Create a new 12 by 12 world
b) Create a new robot (pick your own colour and name). Place your robot in the world in position (3, 10) facing West
c) 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.

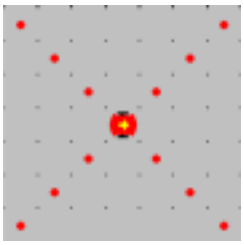
4) Look over the code for `RobotWorldDemoTwo`. Modify this code to satisfy the following:

a) Create a new 17 by 17 world
b) Create and add a new orange robot called "Pumpkin" to this world starting in position (15, 15)
c) 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.
d) 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 <u>any</u> 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).



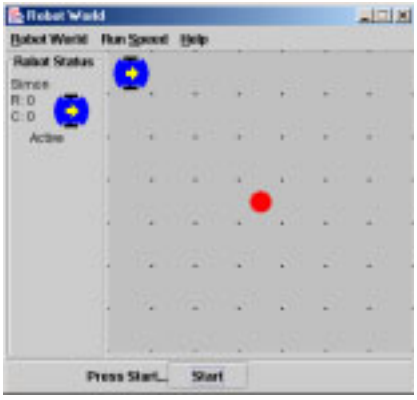
7) Write a Java program to get a robot to run through a maze. The maze world will be loaded from a file. A sample file called "`maze1.txt`" can be found in the Robot World Demo folder. You should use the `addRobotAtStart()` method to place your robot in the maze. For example:

```
myWorld.addRobotAtStart(myRobot);
```

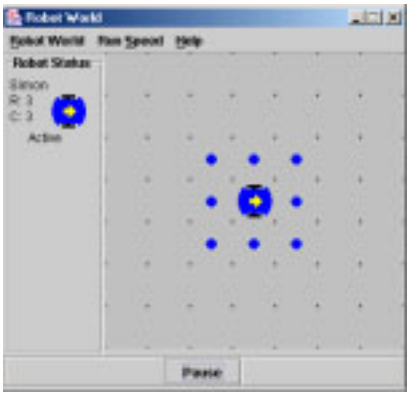
Your objective is to find the exit (use `isExitHere()` to check for the exit). Your program should work for any maze file. Hint: To run the maze, have your robot hug the left or right wall. To test your program, try making a maze of your own.

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.

Before

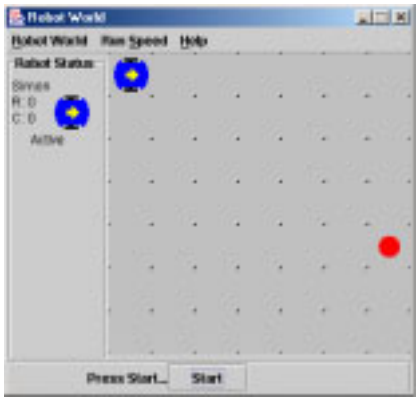


After

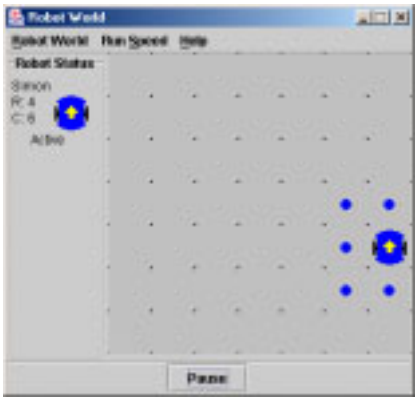


b) Modify your program so that the circle may be placed anywhere in the World. In this case, your row and column will be randomly generated between 0 and 6. For your modified program make sure your Robot can handle the situations when the circle is placed in a corner or along an edge. For example:

Before



After



9) Add the following methods to the RobotPlus class. Read each description carefully. Don't forget to include complete comments for each method:

`turnToFace(int direction)`
-turns the robot to face in a given direction

`goToCorner(int directionNS, int 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 visa 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: `readInt()` or `readDouble()`.

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 GST_RATE = 0.07;
```

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 is good for
multiple line comments.  It is also good 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 title, purpose of program, programmer's name (`@author`), and date last modified (`@version`). For example:

```
/** The "ConsoleOutput" class.
  * Purpose: 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 = c.readInt();     // 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 Your Code

To help identify different structures (loops and selections) within your program code you should use indents and spaces to help highlight these structures. By doing so, you will make your code a lot easier to follow and debug. When using the Ready IDE we can press F2 to automatically indent your code. If pressing F2 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)
{
   c.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:

```
Console c = new Console("Simple Payroll Program");

// Read in the hours worked and the rate of pay
c.print("Please enter your hours worked: ");
double hoursWorked = c.readDouble();
c.print("Please enter your rate of pay: ");
double rateOfPay = c.readDouble();

// Calculate and display the Gross Pay
double grossPay = hoursWorked * rateOfPay;
c.print("Gross pay: ");
c.println(grossPay, 0, 2);
```

Appendix B -- Finding and Correctiong 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.

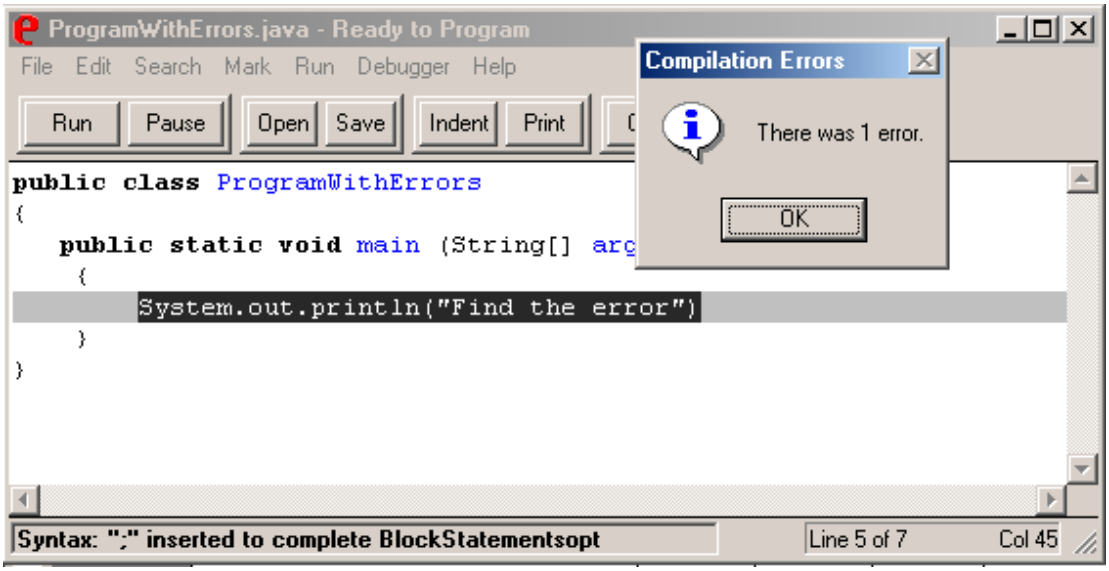
B.1 Types of Errors in Java Programs

The types of errors found in programs can usually be classified into syntax errors, logic errors and run-time errors. As a programmer, you are responsible for writing error (bug) free code that can handle all possible situations including errors in user input.

Syntax Errors

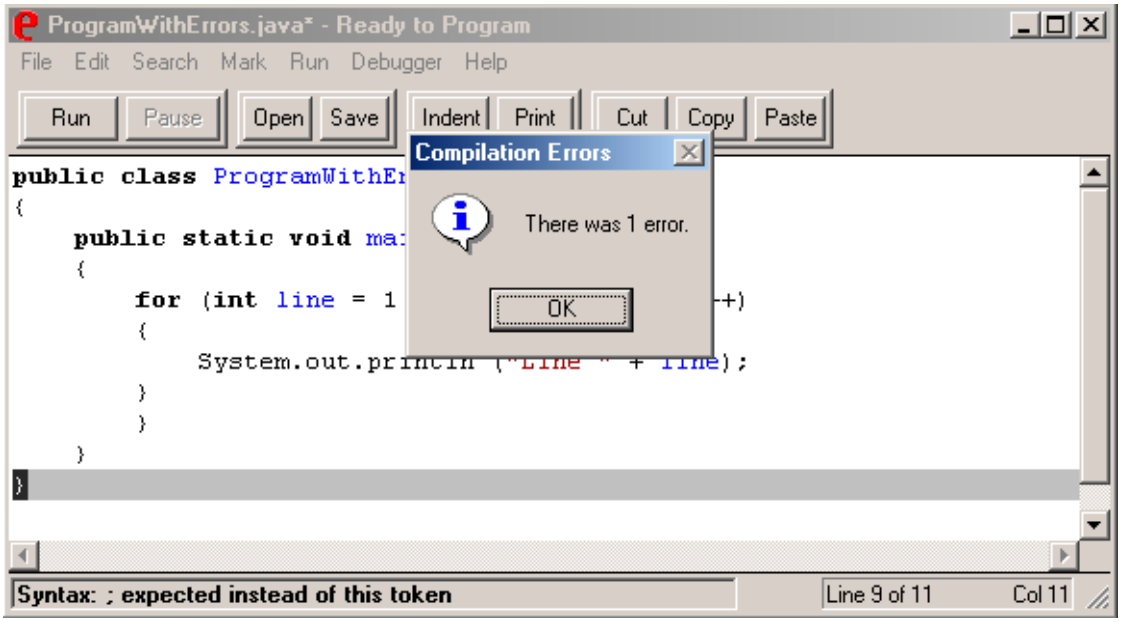
If you misspell a keyword, forget some punctuation such as a ; or a ", or forget to close off a pair of curly braces, you will get a syntax error when you try to compile your program. Since the compiler can't understand the syntax (structure or the form) of the language used, it will be unable to properly translate your code.

The Ready IDE reports syntax errors and usually highlights the line where the error occurred. A brief description of the error is also given in the status bar at the bottom of the Ready window (see example below).

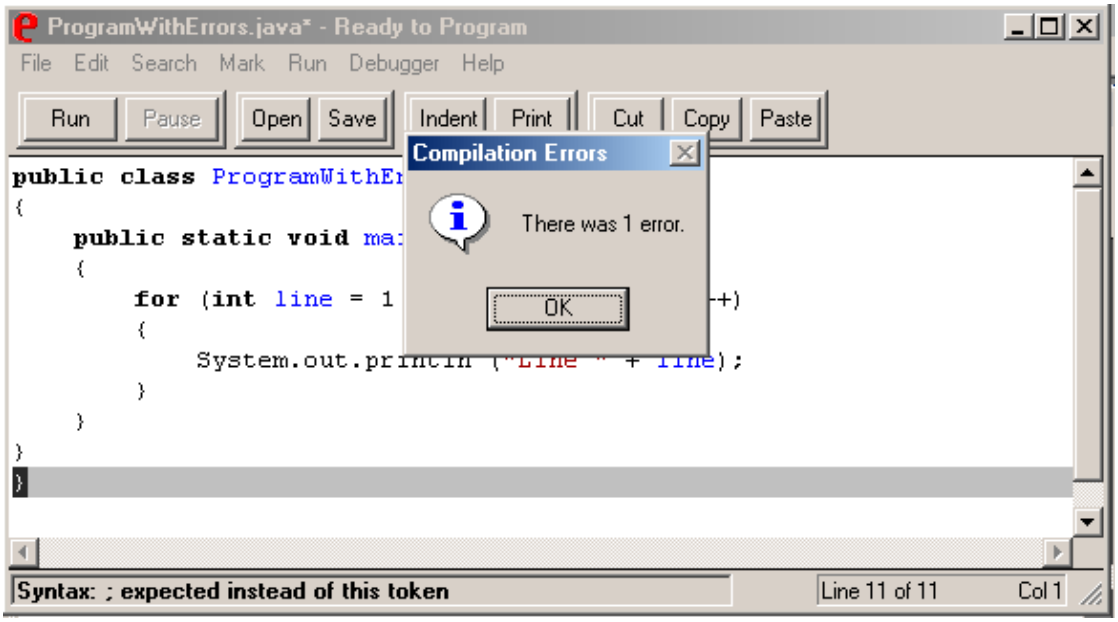


In this case the error message indicates that a ";" is missing.

In some cases the error messages aren't as clear as they could be. For example, on the next page, the error is an extra "}" but the error message reports a missing ";".



Using F2 to properly format your code can also help to find syntax errors. In the above example, after properly indenting your ocde it is easier to find the extra "}" since F2 lines up matching curly braces.



In other cases, the error may be somewhere else in the code but was not recognized until this line was reached. Therefore, don't forget to look for errors in the lines above or below the highlighted line.

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 your syntax 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)
   c.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 in if's and loops where you make a logical choice. 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 reports runtime errors by throwing an object called an `Exception`. 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) **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.

3) **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.

4) **Follow the Java Style Guide in Appendix A**

It is easier to find errors and debug your code when your code has a consitent look and format. Using descriptive variable names helps to avoid logic errors.

5) **Indent your code**

The Ready IDE will automatically indent your code (use F2) 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 using the F2 key to indent your code. In each case you should notice how the indenting of the code highlights the error.

a) Extra semi-colon after an "if"

Original Code:

```
if (hoursWorked > 40);
   overtimeHours = hoursWorked - 40;
```

Code after indenting with F2:

```
if (hoursWorked > 40)
       ;
overtimeHours = hoursWorked - 40;
```

b) Extra semi-colon after a "while"

Original Code:

```
while (age < 0);
{
   c.print("Please re-enter: ");
   age = c.readInt();
}
```

Code after indenting with F2:

```
while (age < 0)
       ;
{
   c.print("Please re-enter: ");
   age = c.readInt();
}
```

c) Missing semi-colon

Original Code:

```
int length = 10;
int width = 15
int area = length * width;
```

Code after indenting with F2:

```
int length = 10;
int width = 15
    int area = length * width;
```

d) Missing curly braces (Both statements should be part of the "if")

Original Code:

```
if (hoursWorked > 40)
   overtimeHours = hoursWorked - 40;
   c.println("You worked overtime");
```

Code after indenting with F2:

```
if (hoursWorked > 40)
   overtimeHours = hoursWorked - 40;
c.println("You worked overtime");
```

B.3 Using Test Data to Find Non-Syntax Errors

Since your program will not run with syntax 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 variety of good test data. Your test data should include a variety of test cases. In particular you should make sure you test the extremes (zero, maximum and minimum values). Also, when dealing with user input, you should always 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 5 different test cases. For example, here is a test plan for problem 18 in Chapter 3.

Case	Input	Predicted Output		Reason for including this test case	OK ✓
	No of Slices	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 -- The HSA Console Class

The Holt Software Associates Console class is used to create a Console window for keyboard input and screen output. The Console window can hold up to 25 lines (rows) of 80 column text.

The Console class window has three buttons. The Save button saves the contents of the screen as a ".bmp" (Windows Bitmap) file. The Print button prints out the contents of the window. When the program is running, the Quit button quits the program immediately. When the program has finished running, the Quit button becomes a Close button that will close the console window.

C.1 Constructors

The following constructors can be used when creating a new console window. Each of the following constructors has the same name but they are distinguished by the type and number of their parameters. Even if you are using a console for graphics output, the size of the window is set using the number of rows and columns. Depending on the text font size, this will determine the size of the window in pixels.

`Console ()`
: Creates a Console window of 25 rows by 80 columns.

`Console (int fontSize)`
: Creates a Console window with the text size set to fontSize.

`Console (int rows, int columns)`
: Creates a Console window with the given number of rows and columns.

`Console (int rows, int columns, int fontSize)`
: Creates a Console window with the given number of rows and columns and the text size set to fontSize.

`Console (String title)`
: Creates a Console window and sets the window title to title.

`Console (int fontSize, String title)`
: Creates a Console window with the text size set to fontSize and sets the window title to title.

`Console (int rows, int columns, String title)`
: Creates a Console window with the given number of rows and columns and the window title set to title.

`Console (int rows, int columns, int fontSize, String title)`
: Creates a Console window with the given number of rows and columns, the text size set to fontSize, and the window title set to title.

C.2 Text Input and Output Methods

The following methods are based on text-based output with positions given by rows and columns. The top left corner of the screen is row 1 and column 1.

`void clear ()`
Clears the entire Console window and sets the cursor to the upper-left corner.

`int getMaxColumns ()`
Returns the width of the Console window in columns.

`int getMaxRows ()`
Returns the height of the Console window in rows.

`int getColumn ()`
Returns the column number of the current cursor position.

`int getRow ()`
Returns the row number of the current cursor position.

```
void print (int i)
void print (double d)
void print (boolean b)
void print (char c)
void print (String s)
```
Outputs the argument to the Console window beginning at the cursor position.

`void print (int number, int fieldSize)`
Outputs number to the Console window in a field of width fieldSize beginning at the cursor position. If the number (when converted to String) is larger than fieldSize, fieldSize is ignored. The output is right justified within the field.

`void print (double number, int fieldSize)`
Outputs number to the Console window in a field of width fieldSize beginning at the cursor position. The method adjusts the number of decimal places displayed to fit in the field, if possible. If the argument (when converted to String) is larger than fieldSize, fieldSize is ignored. The output is right justified within the field.

```
void print (boolean b, int fieldSize)
void print (char c, int fieldSize)
void print (String s, int fieldSize)
```
Outputs the argument to the Console window in a field of width fieldSize beginning at the cursor position. If the argument (when converted to String) is larger than fieldSize, fieldSize is ignored. The output is left justified.

`void print (double number, int fieldSize, int decimalPlaces)`
Outputs number to the Console window in a field of width fieldSize with decimalPlaces number of decimal places beginning at the cursor position. The field width is ignored if the number will not fit. The output is right justified.

`void println (same choices as above)`
Identical to each of the above print method with same arguments, except that a new line is output. To output a blank line we can use a println with no parameters (arguments) (e.g. `println()`)

`int readInt ()`
Returns the 4-byte integer (-2,147,483,648 to 2,147,483,647) read from the keyboard.

`double readDouble ()`
Returns the 8-byte double read from the keyboard.

`boolean readBoolean ()`
Returns the boolean value (either true or false, case insensitive) read from the keyboard.

`char readChar ()`
Returns the character read from the keyboard. See also getChar() in section C.4

`String readString ()`
Returns the String read from the keyboard. (Reads to the first white space)

`String readLine ()`
Returns the entire line of input read from the keyboard without the Return.

`void setTextColor (Color colour)`
Sets the colour for text output from print and println methods.

`void setTextBackgroundColor (Color colour)`
Sets the background colour for text output from print and println methods.

`void showCursor ()`
Shows the text cursor, if invisible.

`void hideCursor ()`
Hides the text cursor, if visible.

`void setCursor (int row, int column)`
Sets the cursor position to the given row and column.

C.3 Console Graphics Methods

The following methods are based on pixel graphics output with positions given by x and y coordinates. The top left corner of the screen is position (0,0). To find the size of the console window in pixels use the `getHeight()` and `getWidth()` methods.

```
void clearRect (int x, int y, int width, int height)
```
Clears the rectangle to the background color.

```
void copyArea (int x, int y, int width, int height, int deltaX,
               int deltaY)
```
Copies the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height* to a position moved by *deltaX* and *deltaY* pixels.

```
void draw3DRect (int x, int y, int width, int height,
                 boolean raised)
```
Draws a 3-D rectangle. It appears raised if *raised* is true.

```
void drawArc (int x, int y, int width, int height,
              int startAngle, int arcAngle)
```
Draws an arc. The arc is inscribed in the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height*. It starts at *startAngle* degrees and goes counterclockwise for *arcAngle* degrees.

```
void drawLine (int x1, int y1, int x2, int y2)
```
Draws a line from (*x1*, *y1*) to (*x2*, *y2*).

```
void drawMapleLeaf (int x, int y, int width, int height)
```
Draws a maple leaf. The maple leaf is inscribed in the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height*.

```
void drawOval (int x, int y, int width, int height)
```
Draws an ellipse. The ellipse is inscribed in the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height*.

```
void drawPolygon (int[] xPoints, int[] yPoints, int numPoints)
```
Draws a polygon. The *xPoints* and *yPoints* arrays define the coordinates of the array of vertices. *numPoints* specifies the number of vertices in the polygon.

```
void drawRect (int x, int y, int width, int height)
```
Draws a rectangle with upper-left corner at (*x*, *y*) with width of *width* and height of *height*.

```
void drawRoundRect (int x, int y, int width, int height,
                    int arcWidth, int arcHeight)
```
Draws a rectangle with rounded corners with upper-left corner at (*x*, *y*) with width of *width* and height of *height*. *arcWidth* and *arcHeight* are the width and height of the ellipse used to draw the rounded corners.

```
void drawStar (int x, int y, int width, int height)
```
Draws a star. The star is inscribed in the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height*.

```
void drawString (String str, int x, int y)
```
Draws the string *str* at the starting point (*x*, *y*). The *y* coordinate is the base line of the text.

```
void fill3DRect (int x, int y, int width, int height,
                 boolean raised)
```
Draws a filled 3-D rectangle. It appears raised if *raised* is true.

```
void fillArc (int x, int y, int width, int height,
              int startAngle, int arcAngle)
```
Draws a filled arc. The arc is inscribed in the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height*. It starts at *startAngle* degrees and goes counterclockwise for *arcAngle* degrees.

```
void fillMapleLeaf (int x, int y, int width, int height)
```
Draws a filled maple leaf. The maple leaf is inscribed in the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height*.

```
void fillOval (int x, int y, int width, int height)
```
Draws a filled ellipse. The ellipse is inscribed in the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height*.

```
void fillPolygon (int[] xPoints, int[] yPoints, int numPoints)
```
Draws a filled polygon. The *xPoints* and *yPoints* arrays define the coordinates of the array of vertices. *numPoints* specifies the number of vertices in the polygon.

```
void fillRect (int x, int y, int width, int height)
```
Draws a filled rectangle with upper-left corner at (*x*, *y*) with width of *width* and height of *height*.

```
void fillRoundRect (int x, int y, int width, int height,
                    int arcWidth, int arcHeight)
```
Draws a filled rectangle with rounded corners with upper-left corner at (*x*, *y*) with width of *width* and height of *height*. *arcWidth* and *arcHeight* are the width and height of the ellipse used to draw the rounded corners.

`void fillStar (int x, int y, int width, int height)`
Draws a filled star. The star is inscribed in the rectangle defined by the upper-left corner (*x*, *y*) with width of *width* and height of *height*.

`int getWidth ()`
Returns the width of the Console window in pixels.

`int getHeight ()`
Returns the height of the Console window in pixels.

`void setColor (Color c)`
Sets the color of the graphics context. The color is used for any draw methods.

`void setFont (Font f)`
Sets the font of the graphics context. The font is used with the drawString method.

`void setPaintMode ()`
Sets the graphics context into paint mode. All drawing in the graphics context draws over the background.

`void setXORMode (Color c)`
Sets the graphics context into XOR mode. All drawing in the graphics context is XOR'd with the background. The color specified by c is a special color so that any drawing done on a background of color c will not be changed.

C.4 Other Console Methods

`char getChar ()`
Returns the character read from the keyboard. Unlike `readChar()`, `getChar()` does not wait for `Enter` to be pressed. Also when using `getChar()`, the character entered is not echoed (shown) on the screen. If you want to show the character entered, you must display this character yourself using `print()`, `println()` or `drawstring()`.

`void close ()`
Closes the Console window and disposes of its contents. Note: In most case you don't need to use this method, since each console window has a built-in `Close` button.

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 `toUpper()` you would use `Character.toUpper ()`

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 $9.\dot{9}$

`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 trigometric functions (sin, cos and tan and their inverses). See the on-line 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 -- The TextInputFile and TextOutputFile Classes

The Holt Software Associates TextInputFile and TextOutputFile classes can be used to simplify reading from and writing to text files in Java. Since these classes are part of the hsa classes, you must include the proper import statement in order to use these classes (see example below)

A TextInputFile object uses the same methods as the Console class to input text information from a file. Instead of reading information from the keyboard you are reading information from a text file.

E.1 TextInputFile Constructors

`TextInputFile ()`
Opens standard input (keyboard) as a TextInputFile.

`TextInputFile (File file)`
Opens file as a TextInputFile.

`TextInputFile (String fileName)`
Constructor - Opens file with name fileName as a TextInputFile. If fileName is "Standard Input", "Keyboard" or "Stdin" (case irrelevant), standard input is opened as a TextInputFile.

E.2 TextInputFile Methods

`int readInt ()`
`double readDouble ()`
`boolean readBoolean ()`
`char readChar ()` (Note: there is no `getChar()` method for text files)
`String readString ()`
`String readLine ()`
Reads data from a text file (see the Console class (Appendix C) if more details are required.

`boolean eof ()`
Returns true if the next thing to be read from the file is the end-of-file marker. (When reading from standard input, you can generate an end-of- file by pressing either Ctrl+Z or Ctrl+D.)

`void close ()`
Closes the TextInputFile to prevent further reading.

See E.4 for an example of how to read from a text file.

A TextOutputFile object uses the same methods as the Console class to output text information to a file. Instead of printing information to the screen you are printing information to a text file. Since we are outputting to a text file, only the text-based methodss are allowed (no graphics methods). Also since text files are sequential, methods such as setCursor are not allowed when outputting to text files.

E.3 TextOutputFile Constructors

`TextOutputFile ()`
Opens standard output as a TextOutputFile.

`TextOutputFile (File file)`
Opens file as a TextOutputFile.

`TextOutputFile (String fileName)`
Constructor - Opens file with name fileName as a TextOutputFile. If fileName is "Standard Output", "Screen" or "Stdout" (case irrelevant), standard output is opened as a TextOutputFile.

`TextOutputFile (File file, boolean append)`
Opens file as a TextOutputFile. If append is true, any output will be appended to the file.

`TextOutputFile (String fileName, boolean append)`
Opens file with name fileName as a TextOutputFile. If append is true, any output will be appended to the file. If fileName is "Standard Output", "Screen" or "Stdout" (case irrelevant), standard output is opened as a TextOutputFile.

E.4 TextOutputFile Methods

`void close ()`
Closes the TextOutputFile to prevent further writing.

`void print ()`
`void println ()`
See Console `print()` and `println()` methods. The only difference is that output is to a text file, instead of a console screen.

E.5 TextInputFile and TextOutputFile Example

The following Java program will convert an entire text file to uppercase characters. It processes one line at a time.

```
/**
 * Purpose: Convert a text file to upper case letters
 * @author Ridout
 * @version July 2001
 */

import java.awt.*;

// We use hsa.* since we are using more than one of the hsa classes
// hsa.Console, hsa.TextInputFile and hsa.TextOutput
import hsa.*;

public class TextFileToUpperCase
{
    public static void main (String [] args)
    {
        Console c = new Console ();

        // Open a text file "input.txt" for input.  For the rest of
        // the program we will refer to this file using the text file
        // object reference "inFile"
        TextInputFile inFile = new TextInputFile ("input.txt");

        // Open a text file "output.txt" for ouput. For the rest of
        // the program we will refer to this file using the text file
        // object reference "outFile"
        TextOutputFile outFile = new TextOutputFile ("output.txt");

        c.println ("Converting to upper case");

        // The following loop, reads a line from the infile
        // This line of text is then converted to uppercase and
        // written to the outFile.  The loop continues while we are
        // not at the end of the inFile
        while (!inFile.eof ())
        {
            String lineOfText = inFile.readLine ();
            outFile.println (lineOfText.toUpperCase ());
        }

        // We should close any open files as soon as possible
        inFile.close ();
        outFile.close ();

        c.println ("Conversion complete");

    } // main method
} // TextFileToUpperCase class
```

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. 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,
               int 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 (int 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 `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 int EAST
static final int SOUTH
static final int WEST
static final int NORTH
```

If you want to pick a random direction, you can use the following static method

```
static int 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:

```
boolean isValidInteger;
do
{
   try
   {
      c.print ("Please enter an integer number: ");
      number = Integer.parseInt (c.readString ());
      isValidInteger = true;
   }
   catch (NumberFormatException e)
   {
      c.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 immediate 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 program control would be 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

Flanagan, David
Java In A Nutshell, 2nd Edition
O'Reilly, Sebastopol, CA 1997

Horstmann, Cay S. and Cornell, Gary
Core Java 2, Volumne 1- Fundamentals
Sun Microsystems Press, Palo Alto, CA 2001

Holt Software
Ready Home Page (includes information on the hsa packages)
http://www.holtsoft.com/ready/

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

Schneider, G. Michael and Gersting, Judith L.
An Invitation to Computer Science, Java Version
Brooks/Cole, Pacific Grove, CA 2000

Sun Microsystems' Java web site including the Java help files
java.sun.com

van der Linden, Peter
Just Java 5th Edition
Sun Microsystems Press, Palo Alto, CA 2002