

# Java with BlueJ

Ron McFadyen

September 4, 2017

©2016 Ron McFadyen  
Department of Applied Computer Science  
University of Winnipeg  
515 Portage Avenue  
Winnipeg, Manitoba, Canada  
R3B 2E9

`r.mcfadyen@uwinnipeg.ca`  
`ron.mcfadyen@gmail.com`

This work is licensed under Creative Commons Attribution NonCommercial ShareAlike 4.0 International Public License. To view a copy of this license visit

<http://creativecommons.org/licenses/by-nc-sa/4.0/>

This work can be distributed in unmodified form for non-commercial purposes. Modified versions can be made and distributed for non-commercial purposes provided they are distributed under the same license as the original. Other uses require permission of the author.

The website for this book is  
[www.acs.uwinnipeg.ca/rmcfadyen/CreativeCommons/](http://www.acs.uwinnipeg.ca/rmcfadyen/CreativeCommons/)

**To Callum**



# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	Java, the beginning . . . . .	11
1.2	The Java Compiler and the Java Virtual Machine . . . . .	13
1.3	BlueJ . . . . .	14
1.4	A First Program . . . . .	15
1.5	Using BlueJ to Run HelloWorld . . . . .	16
<b>2</b>	<b>Basics</b>	<b>21</b>
2.1	Literals . . . . .	21
2.2	Variables . . . . .	21
2.3	Primitive Data Types . . . . .	25
2.3.1	Numeric Data Types: <code>byte</code> , <code>short</code> , <code>int</code> , <code>long</code> . . . . .	25
2.3.2	Numeric Data Types: <code>float</code> , <code>double</code> . . . . .	28
2.3.3	Numeric Expressions . . . . .	31
2.3.4	<code>boolean</code> Data Type . . . . .	38
2.3.5	<code>char</code> Data Type . . . . .	42
2.4	Operators . . . . .	45
2.5	The <code>String</code> Class . . . . .	48
2.6	Output . . . . .	56
2.6.1	<code>System.out</code> . . . . .	56
2.6.2	Redirecting <code>System.out</code> . . . . .	60
2.6.3	<code>JOptionPane</code> . . . . .	62
2.7	Input . . . . .	63
2.7.1	The <code>Scanner</code> Class . . . . .	63
2.7.2	The <code>JOptionPane</code> Class . . . . .	67
<b>3</b>	<b>Control Structures</b>	<b>69</b>
3.1	<i>Compound</i> statements . . . . .	69
3.2	<code>while</code> . . . . .	70

3.3	<code>if</code>	80
3.4	<code>for</code>	91
3.5	<code>do ...while</code>	106
3.6	<code>switch</code>	111
<b>4</b>	<b>Classes in the Java Class Libraries</b>	<b>117</b>
4.1	Random	117
4.2	Character	122
4.3	Scanner	129
4.4	Math	136
4.5	Integer	139
<b>5</b>	<b>ArrayLists</b>	<b>143</b>
<b>6</b>	<b>One-Dimensional Arrays</b>	<b>151</b>
6.1	Initializing arrays	153
6.2	Storage of arrays and copying arrays	154
6.3	The enhanced for	156
6.4	Passing string values into <code>main()</code>	158
6.5	Parallel arrays	159
6.6	Partially filled arrays	161
6.7	Array utilities in Java class libraries	164
<b>7</b>	<b>Introduction to Methods</b>	<b>169</b>
7.1	Example - Sieve of Eratosthenes	170
7.2	Using Methods	173
7.2.1	Value-returning methods	174
7.2.2	Parameters	177
<b>8</b>	<b>Designing Java Classes</b>	<b>183</b>
8.1	Using Multiple Classes	185
8.2	Fields	186
8.3	Methods	189
8.4	Constructors	194
8.5	Visibility Specifications: Public, Private	198
8.6	Overloading	200
8.7	Associations	201
8.8	Reusing code	206
8.9	Parameter lists and arguments	208
8.10	Varargs: a variable number of arguments	211

8.11 Code listings: Student, Subject . . . . . 213

**9 A Brief Introduction to Graphical User Interfaces 221**

9.1 Brief Introduction to Simple GUI Builder . . . . . 223

9.1.1 Listings . . . . . 232





# Preface

This book is Part I of a two-part set that introduces the Java programming language. The text assumes the student will be using the BlueJ development environment and provides some introductory BlueJ material. Our experience has been that BlueJ is easy to learn and provides a good programming environment for the beginner programmer.

The material in chapters 1 through 5, and 7 and 8 are required topics.

- Chapter 1: This is a high-level introduction to Java. The typical HelloWorld program is discussed along with how to run HelloWorld in BlueJ.
- Chapter 2: Basic concepts having to do with constants, variables, data types, expressions and input/output are covered.
- Chapter 3: This chapter covers the major control structures a programmer uses.
- Chapter 4: Java provides a great deal of functionality in its class libraries. In this chapter we introduce several of these classes such as Random . . . Random gives the programmer the ability to simulate throwing dice or tossing coins. As well, useful functionality in utility classes such as Math, Integer, and Character are covered.
- Chapter 5: Many applications require a program to work with collections of data. For example, the set of courses at a university is a collection. Java programs must be able to manage such a set and the ArrayList data structure is well-suited to the task.
- Chapter 7: This chapter introduces methods. Methods are used by programmers to enable code to be reused and to make a program more readable. The basic idea is to create a module with a name that represents what it does and then call that module from elsewhere.

- Chapter 8: The program code in a Java system is managed in structures where the basic component is the *class*. A Java class contains data and executable code. This chapter covers concepts that must be understood if one is to design and implement a Java-based system.

Chapters 6 and 9 are considered optional and are covered as time permits. Chapter 6 covers one-dimensional Arrays . . . arrays provide some of the capability of the ArrayList, but programming arrays is much more difficult than programming ArrayLists. Chapter 9 introduces concepts on Graphical User Interfaces (GUIs) as provided for in a BlueJ extension. GUIs are required if one is going to create interactive programs, but there are many concepts to master and the topic is typically covered in great detail in advanced courses.

The examples in the text, and solutions to many exercises, are available on the website for this text.

# Chapter 1

## Introduction

This book is about programming in Java. We begin with short descriptions of Java and BlueJ. We feel that BlueJ is one of the simplest development environments for the beginning programmer to use. All of the examples in this text have been tested using BlueJ. Sample solutions for most exercises are available on the website for this text.

### 1.1 Java, the beginning

James Gosling is referred to as the father of the Java programming language. He graduated with a BSc (1977) from the University of Calgary and a PhD (1983) from Carnegie Mellon University. Later, in 1994 at Sun Microsystems he created the Java language while leading a team that was purposed with developing a handheld home-entertainment controller targeted at the digital cable television industry. That project did not produce the expected outcome, but in 1995, the team announced that the Netscape Navigator Internet browser would incorporate Java technology, and from there its adoption for implementing systems began.

James Gosling has received several awards, including:

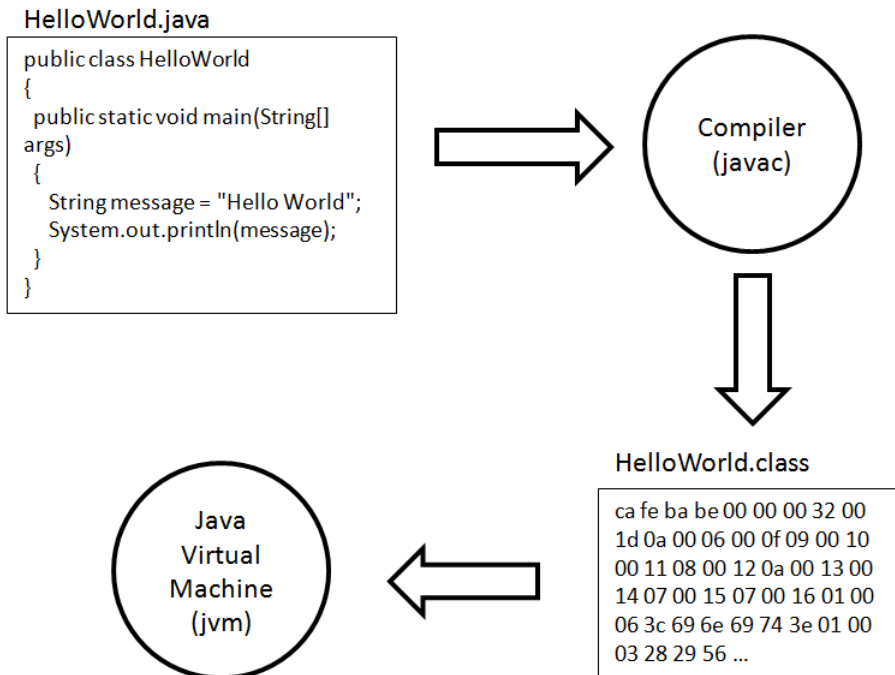
- 2007 - appointed an Officer of the Order of Canada.[1]
- 2013 - named an Association of Computing Machinery Fellow for "Java, NeWS, Emacs, NetBeans, and other contributions to programming languages, tools, and environments".[2]
- 2015 - awarded the IEEE John von Neumann Medal for "the Java programming language, Java Virtual Machine, and other contributions to programming languages and environments".[3]

In 2010 Oracle acquired Sun Microsystems and took over the development of the language. The language has gone through a number of updates, and at the time of writing the current release is referred to as Java 8. All programs in this text have been tested on Java 8.

This text is about programming Java applications. The student may be interested Java applets (these run in a web browser) which are discussed in a future appendix.

## 1.2 The Java Compiler and the Java Virtual Machine

When someone develops a Java program they must first enter the Java code in a text file. Such files have names that end with ".java" and are known as *source code* files. In order to execute a Java program the program must first be translated into Java *bytecode*. We say source code files are human-readable but bytecode files are just 0's and 1's and are not human-readable. A program that performs this translation is called a *compiler*, and we say that the source code is *compiled* into bytecode. The compiler made available by Oracle is called `javac`. Bytecode files always have a name that ends with ".class". The bytecode is not directly executable on a computer - bytecode is not machine code, but it is close to that. Bytecode is "executed" by a special program call the Java Virtual Machine, or JVM. Java programs are portable in the sense that you can write a program and deploy it anywhere - as long as there is a JVM for that platform. The process of developing, compiling, and running a Java program is shown below.



### 1.3 BlueJ

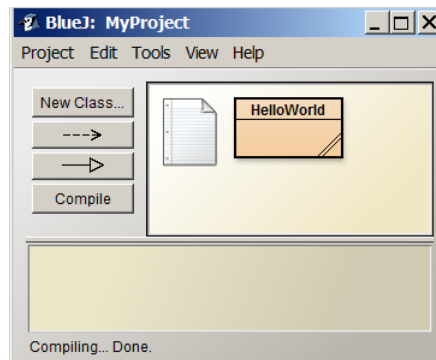
BlueJ is an integrated development environment that provides a programmer with a framework that includes an editor, a compiler, and a runtime environment. It is our experience that BlueJ is very suitable for the beginning Java programmer.

BlueJ is available as a free download from

<http://www.bluej.org>.

We expect that if you are reading this text then BlueJ is already installed on available student workstations. If not please consult your technical services staff. If you need BlueJ on your own computer then please visit <http://www.bluej.org> and follow their download and installation instructions.

Below is a picture showing HelloWorld in a BlueJ project. Note the button available to compile the source code.



## 1.4 A First Program

Shown in Listing 1.1 is the traditional first program, HelloWorld, that appears in many Java texts. When executed, this program does one simple thing: it displays the message "Hello World".

Listing 1.1: HelloWorld.java

```
1 public class HelloWorld
2 {
3     public static void main(String[] args)
4     {
5         String message = "Hello World";
6         System.out.println(message);
7     }
8 }
```

When you inspect this program one thing that is immediately obvious is that there is a lot of overhead to do just one thing. Each line of the program is explained below:

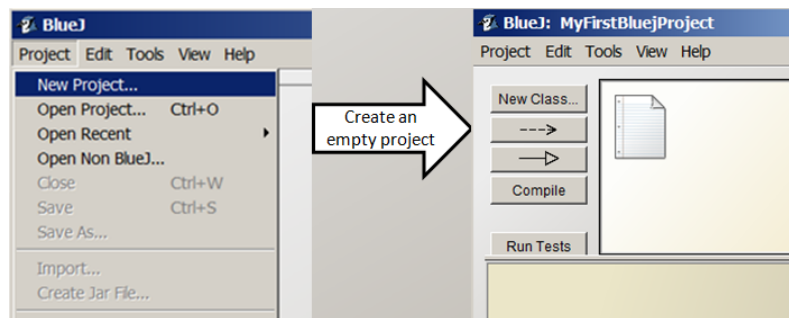
1. The first line gives a name to the program: HelloWorld.
2. The program is actually a Java *class* and the lines making up the class are delimited by the { in line 2 and the } in the very last line.
3. Line 3 begins the definition of a *method* named main. In general, a method can take arguments and the text `String[] args()` is the way those are indicated for a main method - much more on this in later chapters.
4. The lines that comprise the main method begin with the { in line 4 and end with the } in line 7.
5. Line 5 is an assignment statement that says the value to be assigned to the variable `message` is the text `Hello World`. When this line executes the string `Hello World` is stored in memory locations reserved for the variable `message`.
6. Line 6 is an example of how output is obtained. When this line executes the contents of `message` are transferred to a display unit.

## 1.5 Using BlueJ to Run HelloWorld

Very little instruction is required to learn how to use BlueJ. This text assumes that BlueJ is demonstrated in lectures and/or a laboratory setting. More information is available at the BlueJ web site; for instance, there is a tutorial at <http://www.bluej.org/tutorial/tutorial-201.pdf>.

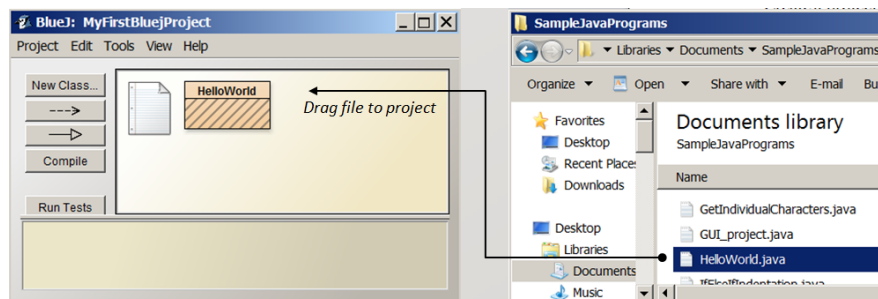
In this section we discuss typical steps one can follow to run HelloWorld in the BlueJ environment.

1. Download the sample programs from the text's web pages.
2. Unzip the sample programs storing them in a folder on your computer. Open the folder and locate the file HelloWorld.java.
3. Start BlueJ and then create a new project:



An empty project actually contains one item  
 ... a file named `ReadMe.txt` that will be discussed later on in the text.

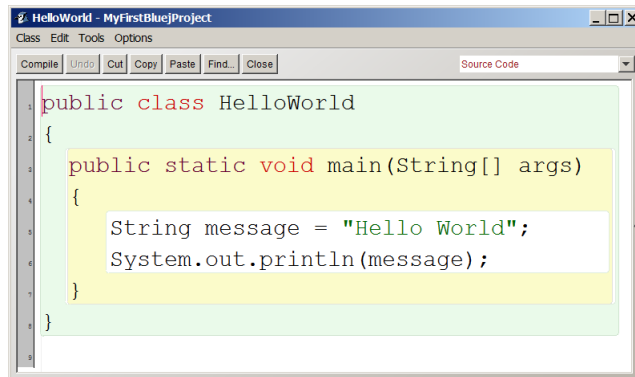
4. Now, to get a copy of `HelloWorld` ... Click the `HelloWorld.java` file, hold the mouse button down, drag the file to your new BlueJ project window, and then release the mouse button:



This action copies the file and now you have `HelloWorld` in your project.

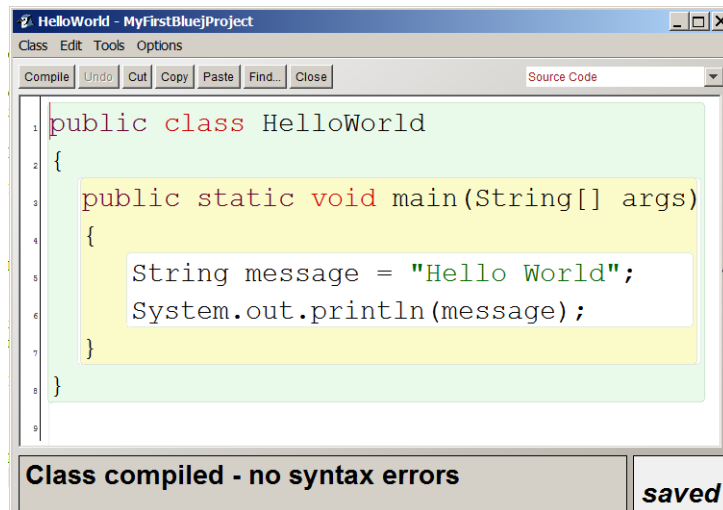


5. Double-click the image in the project representing HelloWorld ...the BlueJ editor opens showing you the contents. You should see the 8 lines shown in Listing 1.1. You should see the editor open as shown below:



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

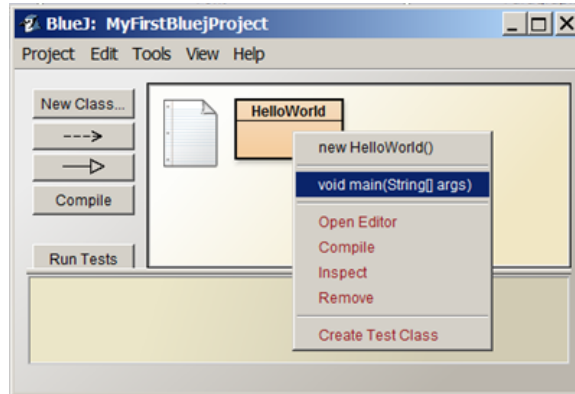
6. The next step is to compile the program. There are two ways to do this ...use the *compile button* on the editor window, or use the compile button on the project window with HelloWorld selected. If you click the compile button on the editor window the response will be that the code compiled with no errors:



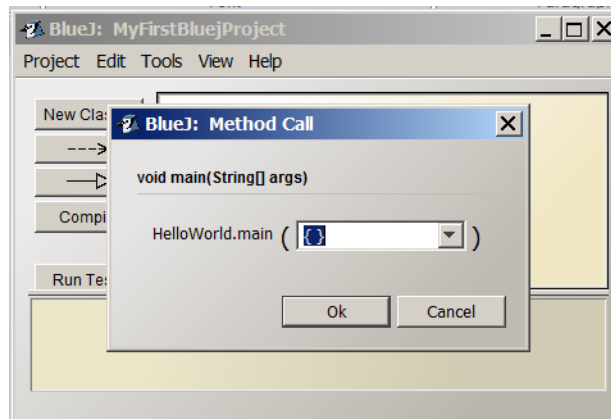
```
public class HelloWorld
{
    public static void main(String[] args)
    {
        String message = "Hello World";
        System.out.println(message);
    }
}
```

**Class compiled - no syntax errors** **saved**

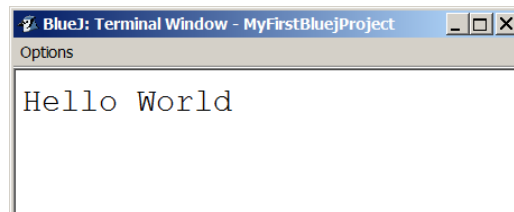
7. Finally, to run the program you must close the editor by clicking the *close button*. You are now back at the BlueJ project where you must right-click the HelloWorld icon and select, from the options shown, to execute the main method:



8. As a result of the above, BlueJ is ready to run the main method and prompts you for any argument values for `main`. Since there are none (arguments are discussed much later in the text), click the OK button:



9. The program runs and you see the output in a window (named the Terminal Window) that pops up:



### **Exercises**

1. Run the Hello World program.
2. Modify the Hello World program so it displays your name instead of "Hello World". To do this you must use the BlueJ editor and alter line 5. Then you must recompile the program and run the new version.



# Chapter 2

## Basics

This chapter covers material that gives you the necessary information to run basic programs that use constants and variables, perform calculations, obtain input from a user, and generate output. The topics covered are literals, variables, primitive data types, the `String` class, input, and output. Java is an *object-oriented* language and with the last three topics you will begin to get an understanding of what object-oriented means. Literals, variables, and the primitive data types are concepts you will find in most (if not all) programming languages you encounter.

### 2.1 Literals

It is common for programs to include constants; in Java these are referred to as literals. Examples include: `123`, `123.45`, `'a'`, `"Gosling"`, `true`. Most of the time a programmer codes numeric literals and boolean literals in the same way we would normally write them down (e.g. `123`, `123.45`, `true`, `false`). With practice you will get used to using single quotes to specify a single character (e.g. `'a'`), or double quotes to specify a text string (e.g. `"Gosling"`).

### 2.2 Variables

The *variable* is a fundamental concept in programming. In general terms we say a variable is a named location in a computer's memory, and the value stored in that location is controlled during the execution of programs. A variable is a name associated with a piece of computer memory - a piece of

memory that holds a value that a program can use and change as it executes.

### *primitive types*

The Java programming language requires us to declare the type of data to be associated with a variable. Java makes a distinction between *primitive* data types and other data types defined via classes. In the Java *class libraries* there are many pre-defined classes, for example: `String` and `System`. The Java language contains eight primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`:

- `byte`, `short`, `int`, and `long` are used for cases where the data is to be treated as whole numbers (numbers without a fractional component). For example, 33, 498, -100 are whole numbers. These data types differ with regards to the magnitude of number they can represent.
- `float` and `double` are used for cases where the data is numeric and where one expects values to have a fractional component such as: 101.5, 26.334, -55.5. When written we show them with a decimal point. Again, these two types differ with regards to size in terms of the number of significant digits and in the magnitude of the number they can represent.
- `char` is used when there are individual characters to be handled. Examples of individual characters are `'a'`, `'b'`, `'q'`, `'$'`. Values are enclosed in single quotes.
- `boolean` is used when the situation requires one to work with logical values of `true` and `false`. In a Java program these values are written just as we do in English: `true`, `false`.

A Java programmer declares a variable in a *declaration* statement, and then uses the variable name later in a program to assign a value, to alter the current value, and to reference the value currently stored. Two example programs follow; in Listing 2.1 the program defines and uses a variable, and in Listing 2.2 the program alters the value stored in a variable.

In Listing 2.1 note the following:

- In line 10 an `int` variable named `i` is declared,
- In line 11 the value 14 is assigned to `i` (that is, the value 14 is stored in the memory location reserved for `i`),
- Line 12 displays the value of `i`

Listing 2.1: Using a variable

```
1  /**
2   * This Java class declares
3   * an int variable named i,
4   * assigns the value 14 to i,
5   * and displays i.
6   */
7  public class Variable
8  {
9      public static void main(String[] args){
10         int i;
11         i = 14;
12         System.out.println(i);
13     }
14 }
```

Listing 2.2 starts off like Listing 2.1 but modifies the value of `i` to 30 in line 13 just before it is displayed for the second time. This program changes the value associated with the variable.

Listing 2.2: Changing the value stored in a variable

```
1  /**
2   * This Java class declares
3   * an int variable named i,
4   * assigns it a value and then
5   * changes its value.
6   */
7  public class VariableChanged
8  {
9      public static void main(String[] args){
10         int i;
11         i = 14;
12         System.out.println(i);
13         i = 30;
14         System.out.println(i);
15     }
16 }
```

## Naming Variables

A convention used by many Java programmers is to choose names that are concise yet meaningful. A name you choose should indicate the intent of its use. In situations where the intent of use involves more than one word a Java programmer will often name the variable in *camel case*. For instance, suppose you need a variable to keep track of net pay. In order to have a proper name a programmer could choose the name `netPay` for the variable. Two words are involved: *net* and *pay*. the first word is in lower case and other word is catenated to it, and only the first letter of the second word is capitalized. Camel case is a style where words are catenated together forming a variable name - the first word is all lower case, the second and subsequent words have only the first letter capitalized.

Some examples of variables named according to camel case:

<code>netPay</code>	<code>grossPay</code>
<code>dayOfWeek</code>	<code>shippingAddress</code>
<code>monthOfYear</code>	<code>billingAddress</code>
<code>studentNumber</code>	<code>lastName</code>

Camel case is a good convention to follow when declaring variables. However, Java will accept any variable name as long as the name starts with a letter and contains any mixture of letters, digits, and the underscore character ('\_'). Some valid variable names include: `a123`, `net_pay`, `gross_pay`.

Java variable names are case-sensitive. This means that variable names such as `NetPay` and `netPay` are different variables.

*keyword*

Java reserves the use of certain names . . . keywords. Keywords are reserved for special purposes and cannot be used as variable names. For example, `int` is reserved for situations where one declares a variable to be of type `int`; you cannot declare a variable with the name `int`. In the sample programs shown so far we have seen a few of these reserved words: `public`, `class`, `void`, `static`. In subsequent chapters we will see other keywords introduced such as `while`, `do`, `if`, `else`.



## Exercises

1. Java requires that all variables be declared. What type of message does the Java compiler report if a variable is not declared before it is used? Consider Listing 2.1. Change line 11 to read
 

```
    abc = 14;
```

 instead of
 

```
    i = 14;
```

 Compile the program. What is the response you get from the compiler?
2. Java does not permit reserved words to be used as variable names. Consider Listing 2.1 again. Change all references to the variable `i` to `public`, as shown here:
 

```
    int public;
    public = 14;
    System.out.println(public);
```

 Compile the program. What is the response you get from the compiler?
3. Java variable names are case sensitive so two variables named `Message` and `message` do not refer to the same thing. Modify line 6 in the `HelloWorld` so that the variable `message` is misnamed as `Message` with a capital M. What is the response you get from the Java compiler?

## 2.3 Primitive Data Types

### 2.3.1 Numeric Data Types: `byte`, `short`, `int`, `long`

These data types are used for numeric values where there is no fractional component - all values are whole integers. These types differ with respect to the amount of memory used (and therefore minimum and maximum values):

data type	memory	minimum value	maximum value
byte	1 byte	-128	127
short	2 bytes	-32768	32767
int	4 bytes	-2147483648	2147483647
long	8 bytes	-9223372036854775808	9223372036854775807

Calculations can involve any of addition, subtraction, multiplication, division, and modulo operations are represented by `+`, `-`, `*`, `/`, and `%` respectively. Some examples follow:

operator	example of use	example's result
+	7+11	18
-	12-5	7
*	3*4	12
/	13/5	2

## Integer Arithmetic

If the operands of an arithmetic operation are both integers, the result is an integer. Consider division - there is no remainder ...  $13/5$  evaluates to 2 and not 2.6. Modulo gives the remainder when the first operand is divided by the second operand ...  $13\%5$  evaluates to 3.

### Example: Division and Modulo

The following example program uses division and modulo to obtain the last two digits of an integer. The output follows the listing.

Listing 2.3: Obtain the last two digits of an integer

```
1 public class IntegerArithmetic
2 {
3     public static void main(String[] args)
4     {
5         // Use integer arithmetic
6         //   Division: no remainder
7         //   Modulo: yields the remainder
8         int number, digit;
9         number = 1297;
10        // Get right-most digit
11        digit = number % 10;
12        System.out.println(digit);
13        // Decrease number by a factor of 10
14        // and get next digit
15        number = number / 10;
16        digit = number % 10;
17        System.out.println(digit);
18    }
19 }
```

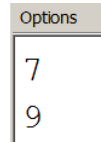


Figure 2.1: Last two digits

## Default Integer Data Type

If a numeric literal has no decimal point (such as 10025) then the data type used is `int`. If a programmer wanted to use the `long` data type the value would have a suffix of `L` or `l`; for example, 10025L. `L` is recommended since the lowercase `l` appears too much like digit 1. We say that `int` is the *default* integer data type.

## Exercises

4. We have seen some simple expressions in Java. Modify the program in Listing 2.2 to use a simple expression. Alter the statement
 

```
i = 30;
```

 to the following (so the value in `i` is multiplied by 3):
 

```
i = i*3;
```

 This statement causes `i` to be multiplied by 3 and the result is stored back in `i`.
5. Modify the program in Listing 2.3 so that each of the four digits are displayed on separate lines.
6. Write a program to determine what happens when:
  - (a) 1 is added to the largest `int` value,
  - (b) 1 is subtracted from the smallest `int` value,
  - (c) an integer is divided by zero.
7. What values are produced by the following expressions:
 

```
99 / 10
99 / 10 * 10
99 % 10
99 - 99 % 10
99 - 99 / 10
99 / 10 * 10 + 99 % 10
```

### 2.3.2 Numeric Data Types: `float`, `double`

These data types are used to represent values that have decimal places. For example, the numbers 11.5, 12.25, -300.123, and 0.0 are written with decimal places. Even the value zero written as 0.0 is a double.

The `float` and `double` types differ with respect to the number of significant digits they store (approximately 7 for float and 16 for double) and the overall magnitude of a value that can be represented. The table below shows the amount of memory used and the maximum value per type:

data type	memory	maximum
float	4 bytes	$3.4028235 \times 10^{38}$
double	8 bytes	$1.7976931348623157 \times 10^{308}$

Of course a programmer can perform calculations on `doubles` and `floats`. The operators we will discuss at this time include `+`, `-`, `*`, and `/` as shown in the following table.

operator	example of use	example's result
<code>+</code>	<code>7.1 + 1.1</code>	8.2
<code>-</code>	<code>12.1 - 5.0</code>	7.1
<code>*</code>	<code>2.2 * 2.2</code>	4.84
<code>/</code>	<code>10.0 / 4.0</code>	2.5

Listing 2.4 illustrates some simple `double` calculations in order to compute and display fuel consumption as litres per 100 kilometres travelled.

Listing 2.4: Perform simple double calculations

```

1 public class FuelConsumption
2 {
3     public static void main(String[] args)
4     {
5         // Calculate fuel consumption as
6         // litres per 100 kilometres travelled.
7         // All calculations involve doubles.
8         double litres, km, km100;
9         litres = 60.6;
10        km = 500.25;
11        km100 = km/100.0;
12        // calculate litres per 100km
13        double consumption = litres/km100;

```

```
14         System.out.println(consumption);
15     }
16 }
```

### Doubles as approximations

Programmers must be aware that not every number can be represented exactly as a `double` or `float`. You know that some fractions cannot be written, using decimals, exactly or completely. Most people use the decimal number system where we can write out the fraction  $1/4$  exactly as `0.25`. However, the fraction  $1/3$  is a repeating decimal. If we write it out as a decimal number we either stop at some number of digits, or, we write it as `0.33333 ...`, to indicate the number has an infinite representation. A similar case arises with computers - there are fractions that cannot be represented exactly in a computer. With limited space we are often storing just an approximation.

One must be aware that round-off can occur when calculations are done with `float` and `double`. Hence they are not appropriate for certain situations: for example if your program needs to represent monetary values. A highly recommended book on Java is *Effective Java*[4]. This is a great reference for the experienced programmer ... for monetary calculations the `BigDecimal` class is recommended. More about this much later on in the text.

The following program demonstrates a calculation: using the minus operator to have one value subtracted from another. If you performed the calculation yourself, you would say the answer is `0.05`. However this program prints a different answer. We have included this example to illustrate how some values are just approximate values.

Listing 2.5: Approximations.java

```
1 public class Approximations
2 {
3     public static void main(String[] args)
4     {
5         // the following result should be
6         // 0.05 but the value printed is
7         // 0.049999999999999716
8         System.out.println(100.25-100.20);
9     }
10 }
```

The output from the above is:

```
Options
0.0499999999999999716
```

### Default Decimal Data Type

When a numeric literal (such as 100.25) appears in a program and has a decimal point, the data type used is `double`; we say that `double` is the default data type for values with a decimal point. If the programmer wanted to use a `float` value then the suffix `f` would be used, as in `100.25f`.

To a non-programmer literals such as 100.0 and 100 may seem the same, but a Java programmer knows the first is represented internally as a `double` and the second is represented as an `int`.

We say that `double` and `int` are the *default* numeric data types. We focus on these numeric data types for the rest of this text.

### Exercises

8. Modify the program in Listing 2.4 to calculate and display fuel consumption as the number of kilometres travelled per litre of fuel.
9. Write a program that converts a value in centimetres to an equivalent value in inches. Use the conversion: one inch equals 2.54 centimetres. Use variables of type `double`.
10. Write a program that converts a value in dollars to an equivalent value in euros. Use the conversion: one euro equals 1.5 dollars. Use variables of type `double`.
11. Modify Listing 2.5 to run some different calculations, such as:

```
System.out.println(100.5-100.00);
System.out.println(100.33-100.00);
```

### 2.3.3 Numeric Expressions

Calculations arise in almost every computerized application. For instance,

- calculating gross pay
- calculating tax payable
- resizing of text on a computer monitor
- direction of a ball when it strikes a border

Calculations are defined using Java expressions which comprise operators and operands. The operators we consider here are addition, subtraction, multiplication, division, and modulo. Operands are either literals, variables, or sub-expressions. Subtraction, multiplication, division, and modulo are represented by `+`, `-`, `*`, `/`, and `%` respectively. All of these operators are binary operators, meaning that they have two operands. Expressions involving these are written in an infix manner where one operand is on the left of the operator and the other operand in on the right of the operator. Sub-expressions are expressions enclosed in parentheses, ( and ).

Some examples of expressions, going from simple to more complex are:

Some Java Expressions	
1	32.0
2	9.0/5.0
3	105%10
4	9.0/5.0*c
5	9.0/5.0*c+32.0
6	5.0*(f-32.0)/9.0

Expressions 3 through 5 are complex and to fully understand how Java evaluates these requires knowledge of operator priorities and associativity.

### Operator Priorities

Java gives each operator a priority and then uses those priorities to control the order of evaluation for an expression. Higher priority operators are executed before lower priority operators. Sometimes a programmer may need to override these priorities and would use a sub-expression for that purpose; a sub-expression is always evaluated before the expression in which it is contained is evaluated. Consider the following table of operator priorities:

Operator Priorities
Highest to lowest
* / %
+ -

Multiplication is given the same priority as division and modulo, and addition is given the same priority as subtraction. However, the priority of multiplication, division, and modulo is higher than that of addition and subtraction. The following table shows expressions, the order of evaluation shown with equivalent sub-expressions, and the final result.

Java expressions involving priorities		
expression	equivalent evaluation	final result
9.0 / 5.0 + 32.0	(9.0 / 5.0) + 32.0	33.8
105 - 105 % 10	105 - (105 % 10)	100
1 + 3 * 2	1 + (3 * 2)	7

The next two examples show situations where operator priorities must be overridden in order to have correct calculations:

### Example: Calculate Net Pay

Suppose we must calculate an employee's net pay. Suppose for the employee we have their gross pay, deductions from gross, and their tax rate in variables named `grossPay`, `deductions`, and `taxRate` respectively. Suppose net pay is calculated by subtracting deductions from gross pay and then multiplying by the tax rate. If we code this as

```
grossPay - deductions * taxRate
```

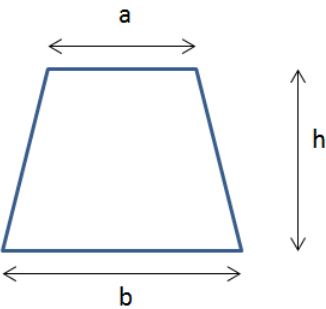
we will get the wrong result since `*` has higher priority than `-`. We can code this using a sub-expression as:

```
(grossPay - deductions)* taxRate
```



**Example: Calculate Area of Trapezoid**

Consider the formula for calculating the area of a trapezoid:

$$area = \frac{a + b}{2} h$$


The diagram shows a blue trapezoid. The top horizontal edge is labeled 'a' with a double-headed arrow above it. The bottom horizontal edge is labeled 'b' with a double-headed arrow below it. To the right of the trapezoid, a vertical double-headed arrow is labeled 'h', representing the height.

If we were to code the formula as

```
a + b / 2 * h
```

the area would be calculated incorrectly due to division and multiplication having higher priority than addition. To force the correct evaluation we can use a sub-expression and override operator priorities: the formula can be written as:

```
(a + b) / 2.0 * h
```

Sub-expressions are used to change the order of evaluation ... in this case to have `a + b` evaluated first before the division and multiplication.

## Operator Associativity

*left associative*

When an expression involves more than one operator of the same priority it is necessary to understand the order in which they are evaluated. If more than one multiplication or division appears they are evaluated from left to right; similarly for addition and subtraction. In programming terms we say these operators are *left associative*.

Suppose if we want to convert a temperature in Celsius to an equivalent Fahrenheit temperature using the formula

$$f = \frac{9}{5} c + 32$$

a programmer can code this as `9.0 / 5.0 * c + 32.0`. This would be correct as the expression is evaluated by Java as required: the division, `9.0 / 5.0`, is performed, then the multiplication, and then the addition. If division and multiplication were right-to-left associative the result of the above would be incorrect.

## Mixed Mode Expressions

Expressions could contain a mixture of types. Java permits conversions between integer and floating-point types.

*widening*

There are two types of conversions: widening and narrowing conversions. By widening we mean that the type being converted to contains all values of the other type. For example, any value of the `short` type (a 2-byte integer) can be represented as an `int` type (a 4-byte integer). In the following program (see line 12) the value of `s`, a 2-byte integer, will be converted to an `int` value, a 4-byte integer.

Listing 2.6: Example of widening

```

1  /**
2   * This Java class declares a short variable,
3   * assigns it a value, and then assigns the
4   * value to a variable of type int
5   */
6  public class ShortToInt
7  {
8      public static void main(String [] args){
9          short s;
```

```
10         int t;
11         s = 100;
12         t = s;
13         System.out.println ("s is: "+s);
14         System.out.println ("t is: "+t);
15     }
16 }
```

Java allows these widening conversions automatically:

- from byte to short, int, or long, float, or double
- from short to int, long, float, or double
- from int to long, float, or double
- from long to float or double
- from float to double
- from char to int, long, float, or double

#### Example 1

Consider the expression

$$(100 - 10) * 0.10$$

The sub-expression involves integers and the result is an integer producing a value of 90. Next in the evaluation will be the multiplication involving 90 and 0.10; note these are different types: `int` and `double`. Java automatically converts the 90 to 90.0 before the multiply is performed.

#### Example 2

Consider the expression

$$(9/5) * 1 + 32.0$$

The sub-expression,  $9/5$ , involves integers and the result is an integer yielding a value of 1. Next in the evaluation will be the multiplication involving 1 and 1, which yields the `int` value of 1. Now we have 1 plus 32.0. For this to be performed the 1 is converted to 1.0 and the final result is 33.0. Note that this would be considered inaccurate (wrong) for the conversion of 1 degree Celsius to Fahrenheit. To obtain a more accurate result  $9.0/5.0$  should be used instead of  $9/5$ .

Narrowing conversions are cases where there could be a loss of precision going from one type to another. For example converting from a `double` to

*narrowing*

an `int` is not allowed unless the programmer directly indicates that `casting` is to be performed. We will leave casting until a later chapter.

## Unary Minus

There are several unary operators where the operator takes one operand. The unary minus is one most people would be familiar with. A unary minus, `-`, can be placed immediately in front of an expression to negate the value of the expression. For instance the value of `-(50-75)` is 25. The unary minus precedes its operand as shown above. Its priority is higher than multiplication, division, and modulo.

## Exercises

- Write a program to calculate an employee's gross pay where variables `hoursWorked` and `rateOfPay` hold the employees hours worked and rate of pay respectively. Gross pay is calculated as hours worked times rate of pay. Test your program with `hoursWorked = 40` and `rateOfPay = $7.50` per hour.
- Write a program to calculate the provincial sales tax and the general sales tax payable for an item where the variables `pstPayable`, `gstPayable`, `price` represent the provincial sales tax payable, the general sales tax payable, and an item's price. Use the formulas:

$$pstPayable = price * 0.05$$

$$gstPayable = price * 0.08$$

Test your program with `price = $50.00`.

- Suppose a customer is charged an amount and the customer gives the clerk an amount that is larger. An amount equal to the difference between the amount given and the amount charged must be returned to the customer. Write a program that calculates this amount to be returned to the customer. Use variables `amountCharged`, `amountGiven`, `amountReturned` to represent the different amounts. Test your program with `amountCharged = $75.50` and `amountGiven = $100.00`.
- Write a program that uses the formula

$$\frac{9}{5}c + 32$$

where  $c$  represents degrees Celsius to calculate the equivalent Fahrenheit value. Test your program using  $c = 22.0$ .

16. Write a program that uses the formula

$$\frac{(f - 32) 5}{9}$$

where  $f$  represents degrees Fahrenheit to calculate the equivalent Celsius value. Test your program using  $f = 22.0$ .

17. Write a program that uses the formula

$$\frac{1}{2} h$$

to calculate the area of a triangle. Run your program for  $h=25$ .

### 2.3.4 boolean Data Type

The `boolean` type has two values: `true` and `false`. We will see that the boolean type can be useful when we discuss control structures in the next chapter. There are three operators defined for booleans: *and*, *or* and *not* represented in Java as `&&`, `||`, and `!` respectively. `&&` and `||` are binary operators where the operator appears between the two operands; `!` is a unary operator that precedes its operand.

Boolean operators		
operator	in Java	meaning
AND	<code>&amp;&amp;</code>	Evaluates to true if and only if both operands are true; evaluates to false otherwise.
OR	<code>  </code>	Evaluates to true if at least one operand is true; evaluates to false if both operands are false.
NOT	<code>!</code>	NOT: negates the operand.

Three *truth tables* below show the results for Boolean operators for all possible values of their operands.

Boolean operation of AND		
a	b	a &b
true	true	true
false	true	false
true	false	false
false	false	false

Boolean operation of OR		
a	b	a   b
true	true	true
false	true	true
true	false	true
false	false	false

Boolean operation of NOT	
a	! a
true	false
false	true

The following table gives some example boolean expressions. The last 3 examples are complex expressions. To understand those evaluations we need

to know Java rules for evaluating these expressions (discussed in the notes below).

Examples		
boolean x, y, z; x=false; y=true; z=true;		
	example	result
1	y	true
2	y && z	true
3	x   y	true
4	!x	true
5	x && y   ! z	false
6	! x   y	true
7	! (x   y)	false

A Boolean expression may have multiple operators. Consider example 5 from above,

```
x && y || ! z
```

We need to be clear on how Java evaluates such an expression. Java assigns priorities to these Boolean operators: ! is highest, followed by &&, followed by ||. Higher priority operators are evaluated before lower priority operators. So, for the above expression !z is evaluated first yielding **false**. So now the expression effectively becomes:

```
x && y || false
```

As && has higher priority than || it is evaluated next yielding **false**. So now the expression effectively becomes:

```
false || false
```

which evaluates to false.

*priorities*

Java allows sub-expressions; these are expressions enclosed in parentheses. A sub-expression is evaluated before the expression of which it is a part, and that value is substituted in its place. Consider examples 6 and 7 from above. They are the same except for the use of parentheses. In example 6:

```
! x || y
```

the ! is evaluated first and the expression effectively becomes:

```
true || y
```

which evaluates to true. In example 7:

```
! (x || y)
```

the sub-expression (x || y) is evaluated first, and then ! is evaluated. Since x||y evaluates to **true**, the result for the expression is **false**.

*sub-expressions*

Some further points about the Boolean operators (not very important to us yet):

- When `&&` is being evaluated and if the first operand is `false`, then the result must be `false` and so the second operand is not evaluated.
- When `||` is being evaluated and if the first operand is `true`, then the result must be `true` and so the second operand is not evaluated.
- There are other operators (`&`, `|`, and `^`) that you may be interested in learning about subsequent to this course. These, amongst other features, are discussed in *Java in a Nutshell* [5].

*other operators*

## Relational Operators

There are a number of operators defined for comparing one value to another. These are summarized in the table below (assume `x` and `y` are of type `int`). These operations evaluate to a boolean (`true` or `false`):

Relational operators		
operator	meaning	example
<code>&lt;</code>	less than	<code>x &lt; y</code>
<code>&gt;</code>	greater than	<code>x &gt; 5</code>
<code>&lt;=</code>	less than or equal to	<code>22 &lt;= y</code>
<code>&gt;=</code>	greater than or equal to	<code>x &gt; y</code>
<code>==</code>	equal to	<code>x == y</code>
<code>!=</code>	not equal to	<code>x != 0</code>

Note that `==` is the operator used to test for equality, and `!=` is used to test two operands to determine if they are not equal. Listing 2.7 illustrates the use of relational operators. Later in the chapter on control structures we will use relational operators in many examples.

Listing 2.7: Comparing char values

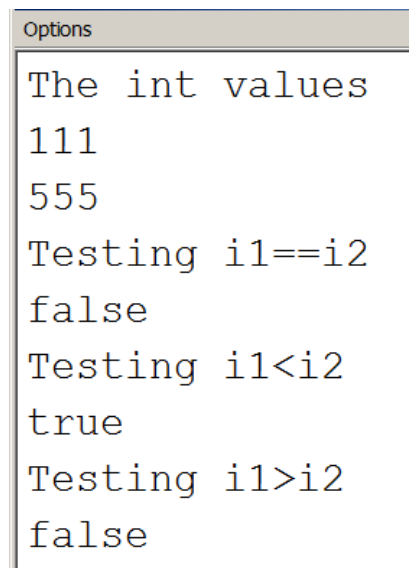
```

1 public class CompareNumber
2 {
3     public static void main(String[] args)
4     {
5         // i1 and i2 are two char variables
6         int i1 = 111;
7         int i2 = 555;
8         // Display i1 and i2

```



```
9         // Display true or false according
10        // to whether they are equal or not
11        System.out.println("The int values");
12        System.out.println(i1);
13        System.out.println(i2);
14        System.out.println("Testing i1==i2");
15        System.out.println(i1==i2);
16        System.out.println("Testing i1<i2");
17        System.out.println(i1<i2);
18        System.out.println("Testing i1>i2");
19        System.out.println(i1>i2);
20    }
21 }
```



```
Options
The int values
111
555
Testing i1==i2
false
Testing i1<i2
true
Testing i1>i2
false
```

Figure 2.2: Results of comparing numeric values

### 2.3.5 char Data Type

`char` is used when you need to handle characters individually. When you see a `char` value in a program you see it enclosed in single quotes, as in: `'a'`, `'A'`, `'q'`, `'%'`.

Java organizes memory for `char` values so that each value is stored using two bytes of memory. One byte of memory is eight bits. We can consider a bit as being either *on* or *off*, or 0 or 1. A byte of memory can be considered to be a sequence of eight 0's and 1's. Just for interest, the bit sequences and corresponding integer value for a few characters is listed below:

character	bit sequence	corresponding integer value
'a'	0000000001100001	97
'a'	0000000001100010	98
'a'	0000000001100011	99
'a'	0000000001000001	65
'a'	0000000001000010	66
'a'	0000000001000011	67
'a'	0000000000110000	48
'a'	0000000000110001	49
'a'	0000000000110010	50
'a'	0000000000100100	36
'a'	0000000000001001	9
'a'	0000000000001010	10

Recall the relational operators defined for comparing one value to another; these operators produce a `boolean` value of `true` or `false`. These are shown again in the table below; assume `x` and `y` are of type `char`.

Relational operators		
operator	meaning	example
<code>&lt;</code>	less than	<code>x &lt; y</code>
<code>&gt;</code>	greater than	<code>x &gt; 'a'</code>
<code>&lt;=</code>	less than or equal to	<code>'a' &lt;= y</code>
<code>&gt;=</code>	greater than or equal to	<code>x &gt; y</code>
<code>==</code>	equal to	<code>x == y</code>
<code>!=</code>	not equal to	<code>x != '\$'</code>

Recall that `==` is the operator used to test for equality, and `!=` is used to test two values to determine if they are not equal. Listing 2.8 illustrates their

use in a program. Later on in the section on the Character class you will see useful techniques for analyzing individual characters in a string.

Listing 2.8: Comparing char values

```
1 public class CompareChar
2 {
3     public static void main(String[] args)
4     {
5         // c1 and c2 are two char variables
6         char c1 = 'a';
7         char c2 = 'z';
8         // Display c1 and c2
9         // Display true or false according
10        // to whether they are equal or not
11        System.out.println("The char values");
12        System.out.println(c1);
13        System.out.println(c2);
14        System.out.println("Testing c1==c2");
15        System.out.println(c1==c2);
16        System.out.println("Testing c1<c2");
17        System.out.println(c1<c2);
18        System.out.println("Testing c1>c2");
19        System.out.println(c1>c2);
20    }
21 }
```

## Exercises

18. Write a program that will print the integer values for the characters '!', '@', '#', '\$', '%', '^', '&', '\*', '(', ')'. Note that Java allows a statement such as `int value = 'a';`
19. Since Java uses two bytes of memory there are 16 bits used to represent `char` values. How many different `char` values can be represented?

```
Options
The char values
a
z
Testing c1==c2
false
Testing c1<c2
true
Testing c1>c2
false
```

Figure 2.3: Results of comparing char values

## 2.4 Operators

We have seen arithmetic, relational, and boolean operators so far. The table below shows these operators and several others: method invocation, post-increment, post-decrement, conditional operator, and assignment. We will soon encounter these in sections and chapters to follow.

Priorities of Operators		
priority	Java operator	meaning
16	name of method(...)	method invocation
	++	post-increment
	-	post-decrement
15	-	unary minus
	!	boolean not
14	new	object creation
13	/	division
	*	multiplication
	%	modulo
12	+	addition
	-	minus
	+	string catenation
10	<	less than
	<=	less than or equal to
	>	greater than
	>=	greater than or equal to
9	==	equals
	!=	not equals
5	&&	boolean and
4		boolean or
3	? :	conditional operator
2	=	assignment
	+=	assignment with operation
	-=	assignment with operation
	*=	assignment with operation
	/=	assignment with operation
	%=	assignment with operation

Previously we stated some operators had the same priority as others, and that some had a lower/higher priority than some others. In the table above

you see the actual operator priorities. For example, both *unary minus* and *boolean not* have the same priority (15) which is much higher than most others. Assignment has the lowest priority (2).

There are several operators we do not discuss in this text. There are operators for every level from 1 to 16; we have not included any of the operators at levels 1, 6, 7, or 8. You could consult a reference such as *Java in a Nutshell* [5] at some future date.

## Complex Expressions

Expressions can be very complex . . . each operand can itself be an expression that evaluates to `true` or `false`. Consider the following complex expression where a, b, c, d, x, z are numeric types:

```
boolean answer = a+b > c+d && x<z
```

The `&&` operator has two operands:

```
a+b > c+d
```

and

```
x<z
```

and each will evaluate to either `true` or `false`. If you look at the priorities of operators you will see the additions will be done first, followed by the relational operators, followed by `&&`, and finally the assignment to the variable `answer`.

Some programmers prefer to include extra spaces and parentheses in expressions like the above . . . in order to make the expression more readable, as in:

```
boolean answer = ((a+b)> (c+d)) && (x<z)
```

In this example the parentheses do not change the order of operations; rather, they may make it easier for someone to *read*.

## The Assignment Operator

What is often referred to as the *assignment statement* is really a Java expression followed by a semicolon. The assignment operator, having a priority of 2, is usually the last operator to be evaluated. The assignment operator is right associative. That is, when several assignment operators appear in an expression they are evaluated/performed from right to left. So, if you have the statement:

```
int q = (j=1)+1;
```

then j will have the value 1 and q will have the value 2. That would be an

odd statement to include; what is more likely is to have several variables all initialized to the same value, as in:

```
int i = j = k = 1;
```

## 2.5 The String Class

It is very common for a program to work with text strings and the Java `String` class is provided to facilitate the many things that programmers need to do with text strings. String literals are written as a sequence of characters that are delimited by double quotes. For example:

```
"this is a line of text"
"my first name is Joe"
"Joe was born on May 1, 1999"
```

The assignment statement:

```
String fullName = "Joe Smith";
```

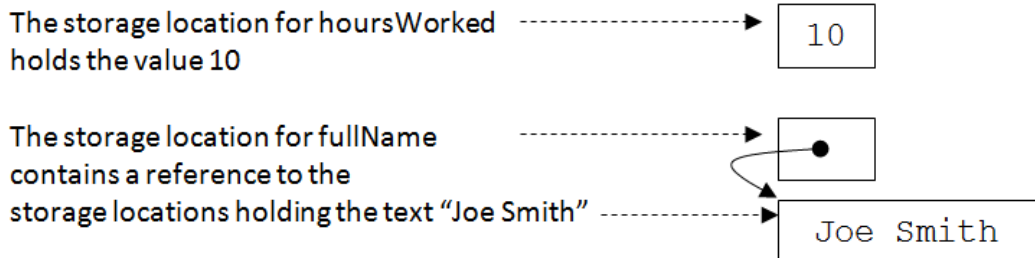
declares `fullName` to be of type `String`. `fullName` is a variable and the Java compiler allocates memory for `fullName`. The memory location for `fullName` will contain a reference (an address) to the storage location where the text string "Joe Smith" is actually stored. Memory for primitive types is handled differently. The memory location associated with a primitive type contains the value (not an address) of the variable. For example, an `int` variable will have a memory location reserved for it where the value of the variable is stored.

There is a subtle difference that may be hard to appreciate at this time: a variable (of type `String`) holds a reference to the value instead of holding the actual value. The diagram below attempts to show the difference.

### Object References

**Java code:**

```
int hoursWorked = 10;
String fullName = "Joe Smith";
```





Since text strings are objects of type `String` another way to declare `fullName` and assign it a value is to use the `new` operator:

```
String fullName = new String("Joe Smith");
```

In general, the `new` operator is used to *instantiate* (to create) an object. Because text strings are so common Java provides the *short cut* for allocating a string, such as:

```
String fullName = "Joe Smith";
```

*new operator*

The only way to work with objects is through the methods that are defined in the class from which the object is instantiated. The `String` class provides many methods for working with text strings such as:

Useful String methods		
method name	type	description
<code>charAt(...)</code>	<code>char</code>	returns the character at a specified position (provided as the argument) where position is one of 0, 1, ..., up to the length of the string.
<code>equals(...)</code>	<code>boolean</code>	used to determine if two strings are identical
<code>equalsIgnoreCase(...)</code>	<code>boolean</code>	used to determine if two strings are identical irrespective of case
<code>indexOf(...)</code>	<code>int</code>	returns the first position of a character provided as an argument, or -1 if it is not present
<code>length()</code>	<code>int</code>	returns the length of a string
<code>toLowerCase()</code>	<code>String</code>	converts all characters to lower case
<code>toUpperCase()</code>	<code>String</code>	converts all characters to upper case
<code>trim()</code>	<code>String</code>	removes leading spaces (blanks) and trailing spaces from a string

Table 2.1: Some of the useful `String` methods

At some time you should view the official documentation for the `String` class. Perhaps you will do this when you are developing a program and you want to look up the `String` methods. If you are using BlueJ its very easy to see the

documentation for a class. As Figure 2.4 illustrates, you just need to click on *Help* and then click on *Java Class Libraries*. Choosing this results in an internet browser opening to a page where, on the left pane, you can find and click on the entry for `String` (or some other class) to view documentation which includes information about methods.

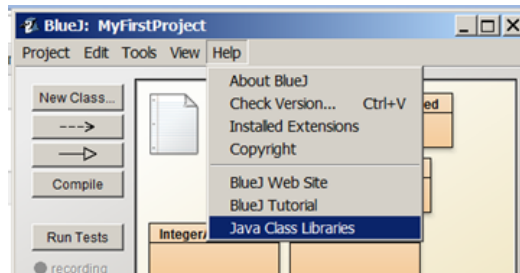


Figure 2.4: Getting documentation on Java classes

Several example programs follow that demonstrate how to use the various methods of the `String` class. To use a `String` method it is necessary that you reference the object and the method. For example, to obtain the length of `s` use the expression `s.length()`. Note the variable name is followed by a period which is followed by the method name and any arguments enclosed in parentheses. In object-oriented terminology we are asking the object `s` to execute its `length()` method.

### Example, obtaining the length of a string

In many applications it is necessary to examine a text string, character-by-character, to ensure it conforms to certain rules. For instance, when a user resets a password there may be restrictions that at least one character is in uppercase, that at least one character is alphabetic, etc. To do this processing where we examine the string character-by-character we need to know how long the string is. The `String` class has a method named `length` which returns, to the point where it is called, an integer value that is the length of the string. The following program shows the method being used; see line 9 in particular:

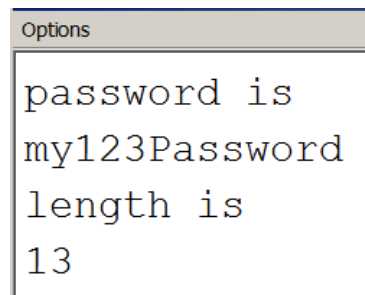
```
int passwordLength = password.length();
```

When this line executes the method is invoked and returns a value to where it was invoked. The value returned is assigned to `passwordLength`. Note the

name of the method is followed by parentheses even though no argument is passed ... this is necessary so the Java compiler knows that the code specifies a method name.

Listing 2.9: Finding the length of a string

```
1 public class UsingStringLength
2 {
3     public static void main(String[] args)
4     {
5         // variable password is of type String
6         // variable passwordLength is of type int
7         String password = "my123Password";
8         // use the length() method
9         int passwordLength = password.length();
10        // Display the string and its length
11        System.out.println("password is");
12        System.out.println(password);
13        System.out.println("length is");
14        System.out.println(passwordLength);
15    }
16 }
```



```
Options
password is
my123Password
length is
13
```

Figure 2.5: Showing the string and its length

### Example, getting the character at a specific position

In many applications where strings are being processed a specific character is expected in a specific position. For instance, a social insurance number can be coded as 3 digits, a hyphen, 3 digits, a hyphen followed by 3 more digits. The following program obtains the character at position 3. The

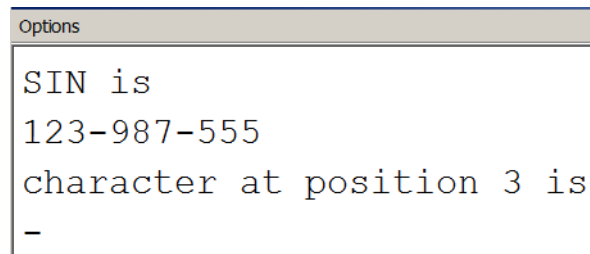
method `charAt(...)` returns the character at a specific position to the point where the method is invoked. The method `charAt(...)` must always have an argument value (the position) passed to it. For Java, positions within a string begin at 0 and so the first hyphen should be at position 3. Note in the program, at line 9, shown here:

```
char firstHyphen = sin.charAt(3);
```

how the method is invoked (a period separates the name of the string, `sin`, from the name of the method) and how the value 3 is passed to the method (in parentheses as an argument value).

Listing 2.10: Obtaining a character within the string at a specific position

```
1 public class UsingStringCharAt
2 {
3     public static void main(String[] args)
4     {
5         // variable password is of type String
6         // variable passwordLength is of type int
7         String sin = "123-987-555";
8         // use the charAt() method
9         char firstHyphen = sin.charAt(3);
10        // Display the string and
11        // the character in position 3
12        System.out.println("SIN is");
13        System.out.println(sin);
14        System.out.println("character at position 3
15                             is");
16        System.out.println(firstHyphen);
17    }
```



```
Options
SIN is
123-987-555
character at position 3 is
-
```

Figure 2.6: Showing the string and the character at position 3

### Example, determining if one string equals another string

To determine if one object is equal to another object you must use a method which, by convention, is named `equals`. The `String` class has a method `equals` and another method named `equalsIgnoreCase`. Both of these methods require an argument to be passed. If we want to compare two strings, say `s1` and `s2`, to see if they are equal we can use either of these two expressions:

```
s1.equals(s2)
s2.equals(s1)
```

Both must return the same `boolean` value. The following program initializes two strings and then displays the value when they are compared for equality. Note line 15, repeated here:

```
boolean equalsResult = s1.equals(s2);
```

On the right hand side of the assignment operator is the expression:

```
s1.equals(s2)
```

We say, in object-oriented terms, that we are asking the object `s1` to execute its `equals()` method with the argument `s2`. When line 15 executes the method is invoked, a value is returned to this point in the statement, and that result is assigned to the `boolean` variable `equalsResult`.

Listing 2.11: How to compare two strings

```
1 public class UsingStringEquals
2 {
3     public static void main(String[] args)
4     {
5         // variable password is of type String
6         // variable passwordLength is of type int
7         String s1 = "John A. Smith";
8         String s2 = "John A Smith";
9         // Display s1 and s2
10        // Display true or false according
11        // to whether they are equal or not
12        System.out.println("The strings");
13        System.out.println(s1);
14        System.out.println(s2);
15        boolean equalsResult = s1.equals(s2);
16        System.out.println(equalsResult);
17    }
18 }
```

```
Options
The strings
John A. Smith
John A Smith
false
```

Figure 2.7: Showing the result of using `equals()`

### Catenation operator `+`

We have seen the `+` operator before, but the operands were always numeric. The `+` operator can also be used to *add* (i.e. concatenate) strings. It is used frequently in statements that generate output. If at least one operand is a string then a result is formed by joining two strings. Joining two strings is called catenation.

When one operand is not a string then the equivalent string representing that non-string's value is generated, and then the catenation of two strings is carried out forming a new string as a result. For instance if you want to display a message "The value of X is " followed by the value of x you just code:

```
System.out.println("The value of x is "+x);
```

However, suppose you wanted to display a string and show the sum of two numbers. Consider:

```
int x = 10;
int y = 11;
System.out.println("the total is "+x+y);
```

Someone might expect the output from the above to be:

```
The total is 21
```

but it is not, rather the output is:

```
The total is 1011
```

The reason this happens is that the expression is evaluated from left to right. The first `+` is adding a string and a number producing the string "The total is 10". Hence the second `+` is adding a string and a number producing the string "The total is 1011".

**Exercises**

20. Evaluate the following Java expressions:

```
"x = "+100
"The remainder is "+ (21 % 10)
(21 % 10)+ " is the remainder"
"x = "+100+200
100 +"is the value of x"
100 + 200 +"is the value of x"
"" + 100 + 5
```

21. Write a program with 3 String variables: firstName, middleInitials, lastName. Assign values to these variables to represent your name. Print a line that shows your name displayed with the last name first, followed by a comma, followed by your first name, and then your middle initials. For example:  
Smith, John A

## 2.6 Output

We discuss two different ways to generate output from a program: using `System.out` and `JOptionPane`. We discuss the use of `System.out` in the next two sections; in the second of these we discuss how you can redirect the output which normally appears in the Terminal Window to a file.

In the third section we discuss the `JOptionPane` class and how that can be used to present information and data in the form of dialog boxes.

### 2.6.1 `System.out`

A simple way to generate output for the user is to use the `println(...)` and `print(...)` methods that belong to the pre-defined Java class named `System` and an object within `System` named `out`. The output generated is said to go to the standard output device. When you use this type of output with BlueJ you will see a window pop up named "Terminal Window" that contains the output produced by the program.

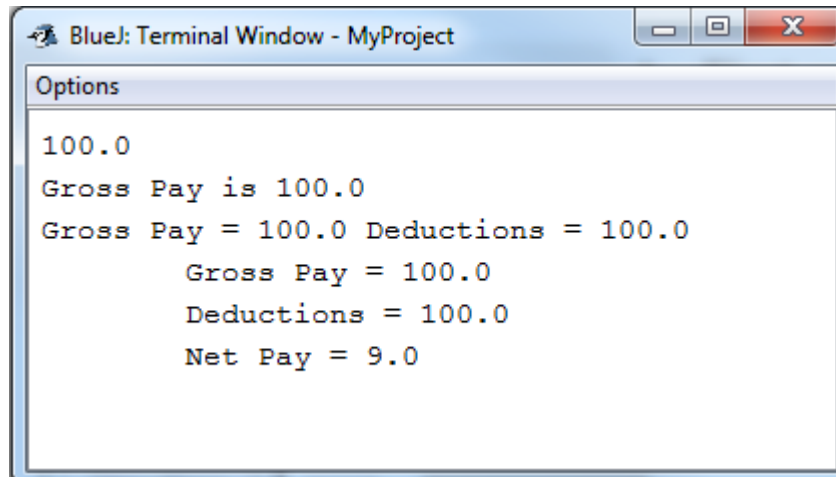
The following program listing illustrates ways of producing output. The `println(...)` and `print(...)` methods take one argument which is a text string. Often that text string is composed of multiple concatenations. Notice the last `println(...)` introduces special characters for *new line* (`\n`) and *tab* (`\t`). The special characters are not displayed, they are used to control the appearance of the output.

Listing 2.12: Using `println()`

```
1 public class UsingPrintln
2 {
3     public static void main(String[] args)
4     {
5         double grossPay, taxesPaid, taxRate,
6             netPay, deductions;
7         grossPay = 100.00;
8         deductions = 10.00;
9         taxRate = 0.10;
10        // Calculate taxes and net pay
11        taxesPaid = netPay = (grossPay -
            deductions) * taxRate;
        //
```



```
12         // Each time println() executes the output
           // will start on a new line
13         // Produce one line of output with one
           // double value
14         System.out.println(grossPay);
15         // Often a good idea is to label the output
           // so it is self-describing
16         // Produce one line of output with a label
           // and a value
17         System.out.println("Gross Pay is
           "+grossPay);
18         // Several items can be concatenated
19         // Note that one text string must appear on
           // one line
20         // but a long one can be formed over
           // multiple lines
21         System.out.println("Gross Pay = "+grossPay
22         +" Deductions = "+grossPay);
23         // You can force output to go onto more
           // than one line
24         // by embedding control characters in a
           // string
25         // '\n' is the new line character
26         // '\t' is the tab character
27         System.out.println("\tGross Pay = "+grossPay
28         +"\n\tDeductions = "+grossPay
29         +"\n\tNet Pay = "+netPay);
30     }
31 }
```

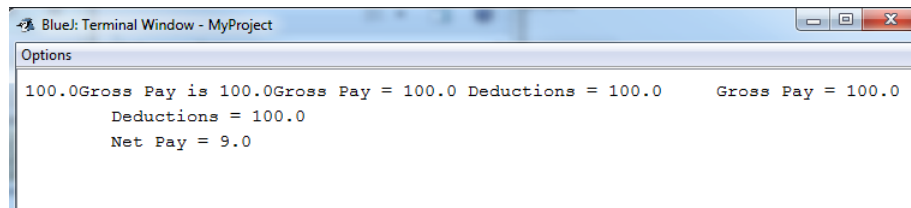


```

BlueJ: Terminal Window - MyProject
Options
100.0
Gross Pay is 100.0
Gross Pay = 100.0 Deductions = 100.0
    Gross Pay = 100.0
    Deductions = 100.0
    Net Pay = 9.0

```

The `println(...)` method causes the display to advance to a new line and then displays output. The `print(...)` method differs from `println(...)` in that it does not automatically advance to a new line when it displays output; instead, output begins at the point where the previous `print(...)` or `println(...)` left off. If we change all the `println(...)` to `print(...)` expressions for the previous example the output we get is:



```

BlueJ: Terminal Window - MyProject
Options
100.0Gross Pay is 100.0Gross Pay = 100.0 Deductions = 100.0    Gross Pay = 100.0
    Deductions = 100.0
    Net Pay = 9.0

```

## Exercises

22. Write a program to calculate the total of the provincial sales tax, the general sales tax, and the price of an item. Use the variables `totalPayable`, `pstPayable`, `gstPayable`, and `price` to represent the total payable, the provincial sales tax payable, the general sales tax payable, and the item's price. Use the formulas:

$$pstPayable = price * 0.05$$

$$gstPayable = price * 0.08$$

$$totalPayable = price + pstPayable + gstPayable$$

Test your program with `price = $50.00`. Your program must display `price`, `pstPayable`, `gstPayable`, and `totalPayable`. Similar to lines 27-29 in Listing 2.12 create output that is nicely aligned with values on separate lines.

### 2.6.2 Redirecting System.out

By default the `println(...)` and `print(...)` methods create output that is displayed on the standard output device (also called the *Console*) - with BlueJ we know this as the window named Terminal Window. The output is handled this way because the default value of `System.out` is a `PrintStream` object directed to standard output. If we redirect the output to a file we can reuse our knowledge of `println(...)` and `print(...)` to create files. The following program has a `main()` method that writes a line to a text file named `myfile.txt`. Note the 3 import statements in lines 1, 2, and 3. Further remarks follow the program listing.

Listing 2.13: Redirecting output to a file.

```
1 import java.io.File;
2 import java.io.PrintStream;
3 import java.io.FileOutputStream;
4 /**
5  * Redirect output to a file
6  */
7 public class RedirectOutputToFile
8 {
9     public static void main(String[] args)
10         throws Exception {
11         System.out.println("1. to standard output");
12         PrintStream standard = System.out;
13         File f = new File("myfile.txt");
14         FileOutputStream fs =new
15             FileOutputStream(f);
16         PrintStream ps = new PrintStream(fs);
17         System.setOut(ps);
18         System.out.println("2. to the other file");
19         ps.close();
20         System.setOut(standard);
21         System.out.println("3. to standard output");
22     }
```

Some comments:

- In line 13 we declare a file named `myfile.txt`. If this file already exists any existing lines are removed.

- Lines 14 and 15 create a new `PrintStream` that replaces the standard output in line 16.
- Line 17 results in a line of text being written to the file `myfile.txt`.
- Its important that a program closes a file (line 18) in order to release resources and to allow other programs or users to access the file.
- In line 19 the value of `System.out` is reset to its initial value.
- The output generated in line 20 goes to the Terminal Window.

If you run the above program you will find that a new file is created in your BlueJ project folder. You will not see it when you have your project open in BlueJ, but if you navigate to the project folder in your file system you will see the file named `myfile.txt`. You can open the file with a text editor.

### Exercises

23. Write a program that prompts the user for their first name, last name, and middle name. The program creates a file named `names.txt` where the names are on separate lines of the file.

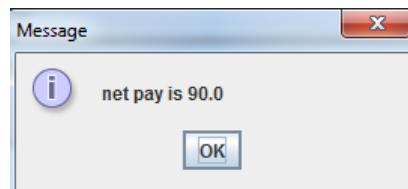
### 2.6.3 JOptionPane

In some situations a programmer may prefer to use `JOptionPane` message dialogs in order to provide the user a more *interactive* experience. The following program shows how to display some information to the user and where the program waits for the user to respond with the click of a button. When the pop-up window appears, the program is suspended until the user clicks the OK button. Note that line 1 is an `import` statement that directs the compiler to the location where it finds details regarding the `JOptionPane` class.

Listing 2.14: Using `println()`

```
1 import javax.swing.JOptionPane;
2 public class UsingDialogBox
3 {
4     public static void main(String[] args)
5     {
6         double netPay, grossPay, deductions;
7         grossPay = 100.00;
8         deductions = 10.00;
9         // Calculate net pay
10        netPay = grossPay - deductions;
11        JOptionPane.showMessageDialog(null, "net
           pay is "+netPay);
12    }
13 }
```

When line 11 executes the pop-up window becomes visible and the program waits for the user to press the OK button:



### Exercises

24. Write a program that prompts the user for their first name, last name, and middle name. The program then displays the names in a dialogue box.

## 2.7 Input

We examine two ways a programmer can arrange to get input from the user by using pre-defined Java classes: the `Scanner` class and the `JOptionPane` class.

### 2.7.1 The Scanner Class

A `Scanner` object can be used with the standard input stream which is named `System.in`. The typical statement used is:

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

`System` is a pre-defined Java class that has an object named `in`. Once a variable like `keyboard` is defined the programmer can use methods defined for a scanner object to get values (Java refers to these as *tokens*) the user has typed on the keyboard. Some of the most useful methods are listed below.

Useful Scanner methods	
<code>hasNext()</code>	returns true if the scanner has more tokens
<code>next()</code>	returns the next token
<code>nextLine()</code>	returns the next line
<code>nextInt()</code>	returns the next int in the input stream
<code>nextDouble()</code>	returns the next double in the input stream
<code>nextBoolean()</code>	returns the next boolean in the input stream

The program below shows one how to use `next()`, `nextDouble()`, and `nextInt()` to obtain a user's name, hours worked and rate of pay. Note that line 1 is an `import` statement that directs the compiler to the location where it find details of the `Scanner` class. The program uses pairs of statements; for example consider lines 12 and 13 repeated here:

```
System.out.println("\n\nEnter your name and press enter");
name = keyboard.next();
```

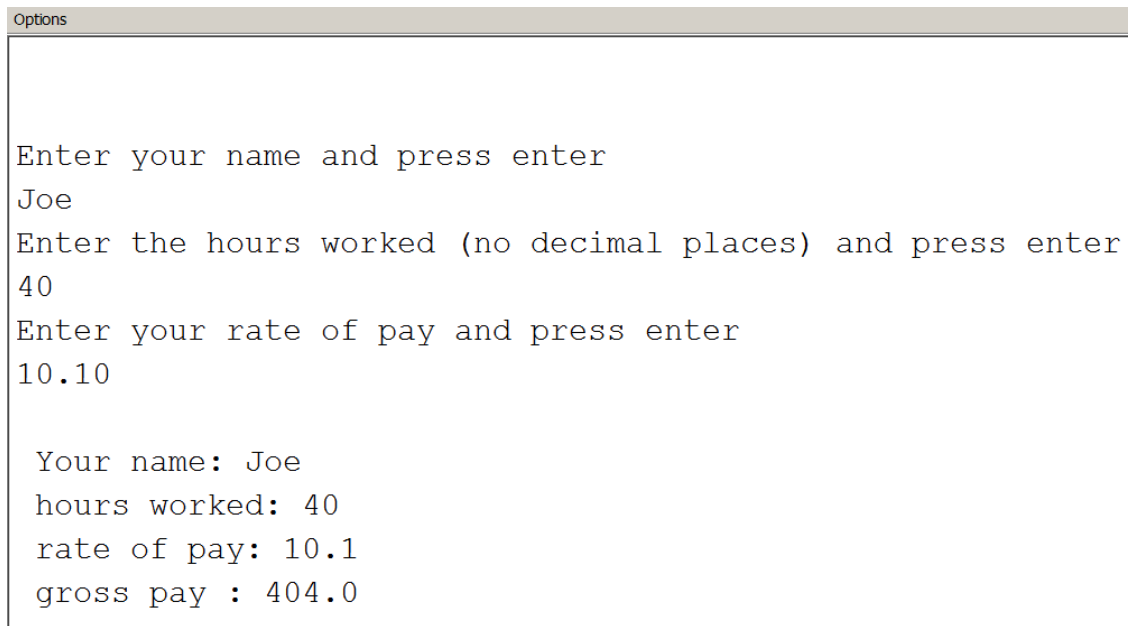
Line 12 prompts the user, and then in line 13 the user's input is obtained. The listing is followed with the contents of the Terminal Window for a sample run. This shows the output (prompts) from the program and the input provided by the user via the keyboard.

Listing 2.15: Using `JOptionPane` for input

```
1 import java.util.Scanner;
2 public class UsingScannerForInput
3 {
```

```
4 public static void main(String[] args)
5 {
6     double rateOfPay;
7     String name;
8     int hoursWorked;
9     // Declare a scanner object for the keyboard
10    Scanner keyboard = new Scanner(System.in);
11    // Prompt the user for a name
12    System.out.println("\n\nEnter your name (no
        spaces) and press enter");
13    name = keyboard.next();
14    // Prompt the user for hours worked
15    System.out.println("Enter the hours worked
        (no decimal places) and press enter");
16    hoursWorked = keyboard.nextInt();
17    // Prompt the user for the rate of pay
18    System.out.println("Enter your rate of pay
        and press enter");
19    rateOfPay = keyboard.nextDouble();
20
21    // Calculate gross pay and display all the
        information
22    double grossPay = hoursWorked * rateOfPay;
23    System.out.println("\n Your name: "+name
24        +"\n hours worked: "+hoursWorked
25        +"\n rate of pay: "+rateOfPay
26        +"\n gross pay : "+grossPay);
27 }
28 }
```





```
Options
Enter your name and press enter
Joe
Enter the hours worked (no decimal places) and press enter
40
Enter your rate of pay and press enter
10.10

Your name: Joe
hours worked: 40
rate of pay: 10.1
gross pay : 404.0
```

Figure 2.8: Terminal Window showing interaction with user

**Exercises**

25. Write a program that prompts the user for their birthday as the day (as an integer), followed by the month (as text), followed by the year (as an integer) with at least one space between the values. Use the Scanner methods `next()` and `nextInt()` to get these values. Then the program displays the birthday in the format *month day, year*. For example, if the user entered

1 January 1990

then the program would display

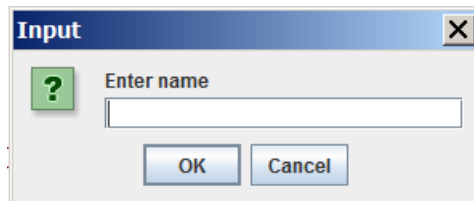
January 1, 1990.

### 2.7.2 The JOptionPane Class

To provide a user with a more interactive experience the programmer can use `JOptionPane` to prompt the user and to obtain text input from the user. One of the methods defined in `JOptionPane` is `showInputDialog(...)`. When this method executes the user is prompted to enter text. The text the user enters becomes the value of the method. Typically `showInputDialog(...)` is on the right-hand-side of an assignment statement; for example:

```
String name = JOptionPane.showInputDialog("Enter name");
```

When the above line executes the user sees the *pop-up* window:



The user then uses the keyboard to enter something in the white box in the pop-up, and then clicks the OK button. The text the user entered is the value returned by the method.

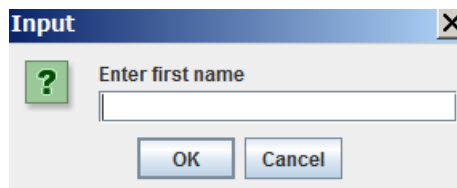
The following program uses two pop-ups to obtain values from the user; comments follow the listing.

Listing 2.16: Using `JOptionPane` for input

```
1  /**
2   * This Java class uses JOptionPane to obtain
3   * input from the user
4   */
5  import javax.swing.JOptionPane;
6  public class UsingJOptionPane
7  {
8      public static void main(String[] args){
9          String firstName =
10             JOptionPane.showInputDialog("Enter first
11             name");
12         String lastName =
13             JOptionPane.showInputDialog("Enter last
14             name");
```

```
11         System.out.println ("Your name is:  
12             "+firstName+" "+lastName);  
13     }
```

- Line 5 is required since we need to tell the Java compiler where it can find the `JOptionPane` class.
- When line 9 executes it causes a dialog box to be displayed to the user:



The user enters a value in the box and presses OK. Then control returns to the program and the value entered is assigned to `firstName`.

- A similar dialog box is displayed when line 10 executes.
- In line 11 the values obtained from the dialog boxes is displayed in BlueJ's terminal window.

## Exercises

26. Write a program that uses a dialogue box to prompt the user for a temperature in Celsius. Then the program uses a dialogue box to display the equivalent temperature in Fahrenheit.
27. Write a program that uses a dialogue box to prompt the user for a temperature in Fahrenheit. Then the program uses a dialogue box to display the equivalent temperature in Celsius.

## Chapter 3

# Control Structures

Programmers need 3 basic control structures when coding programs. These three things are: sequences, decisions, and loops. A sequence structure is one that comprises instructions that are to be executed sequentially one after the other. A decision structure allows for exactly one of a set of sequences to be executed. A loop structure comprises a sequence that is to be executed iteratively. Java has one sequence structure, two (some may say more) different decision structures, and several ways of coding loops.

### 3.1 *Compound statements*

Java statements delimited by curly braces form a *compound* statement. The opening brace, "{", appears first followed by Java statements and then the closing brace, "}" follows. Any time you include a "{" you must have a matching "}". Pairs like these must be used properly - a pair must never overlap with another pair, but as we will see one compound statement can contain another compound statement (see sections on nesting statements). An example is the following compound statement where the values of `x` and `y` are interchanged:

```
{
    temp = x;
    x = y;
    y = temp;
}
```

As you go through the examples in this text you will see many cases where compound statements are used.

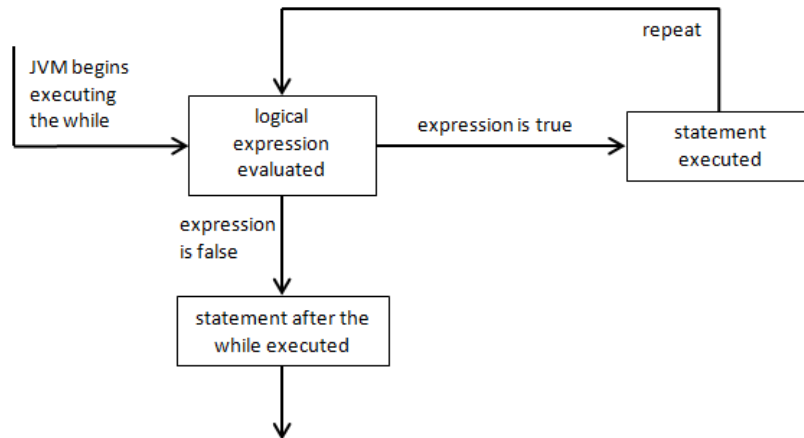
### 3.2 while

The JVM executes the statements in a program sequentially, one statement after another. However, the `while` statement can change this. A `while` statement contains a statement to be executed repeatedly as long as some logical expression is true. The statement executed repeatedly is often a compound statement. The general syntax is

```
while ( logical expression )  
    statement
```

The order of execution of Java statements can be visualized using a flow diagram:

How the JVM executes a while



A logical expression is an expression that evaluates to a boolean value, i.e. true or false. Java has several operators which evaluate to true and false including the relational and equality operators. Recall the relational operators are `<`, `<=`, `>=`, and `>`:

Relational operators		
operator	meaning	example
<code>&lt;</code>	less than	<code>count &lt; 100</code>
<code>&gt;</code>	greater than	<code>netPay &gt; 100</code>
<code>&lt;=</code>	less than or equal to	<code>netPay &lt;= grossPay</code>
<code>&gt;=</code>	greater than or equal to	<code>number &gt;= 0</code>

Recall the equality operators are `==` and `!=`.

Equality operators		
operator	meaning	example
<code>==</code>	equal to	<code>netPay == grossPay</code>
<code>!=</code>	not equal to	<code>netPay != grossPay</code>

Recall the logical operators *and*, *or*, and *not* where operands are `boolean` values and the result is a `boolean`:

Boolean operators		
operator	in Java	meaning
AND && evaluates to false otherwise.	<code>&amp;&amp;</code>	Evaluates to true if and only if both operands
OR && evaluates to false if both operands are false.	<code>  </code>	Evaluates to true if at least one operand is true
NOT	<code>!</code>	NOT: negates the operand.

Some sample boolean expressions follow where

```
boolean found = false; int i = 5; int j = 100;
```

expression	result
<code>j == 100</code>	true
<code>j != 100</code>	false
<code>found &amp;&amp; i &lt; 100</code>	false
<code>!found &amp;&amp; i &lt; 100</code>	true
<code>!found    i == -1</code>	true
<code>i &gt; 100 &amp;&amp; j &lt; 100</code>	false

**Example 1**

The following program prints numbers from 0 to 9. It does so by executing a compound statement repeatedly. The variable `count` is initialized in line 8 to the value 0 and as the `while` loop executes `count` takes on the values 1, 2, 3, etc. A detailed explanation of the program follows the program listing. As you examine the code note the use of indentation - for readability purposes it is common practice to indent the lines comprising a `while` loop.

Listing 3.1: Displaying numbers

```
1  /**
2   * Display the numbers from 0 to 9.
3   */
4  public class Numbers0To9
5  {
6      public static void main(String[] args)
7      {
8          int count = 0;
9          System.out.println("Numbers");
10         while ( count < 10 ){
11             System.out.println(count);
12             count = count + 1;
13         }
14         System.out.println("*****");
15     }
16 }
```

The JVM starts sequential execution with the statement in line 8 - the variable `count` is initialized to 0. The JVM then moves on to Line 9 which results in the printing of a heading for the output. Next, the JVM encounters the `while` loop in Line 10. Observe that lines 11 and 12 are part of a compound statement. This compound statement is executed for `count` equal to 0, 1, 2, and so on, up to `count` equal to 9; when `count` has the value 9 the compound statement is executed and `count` is assigned the value 10 in line 12. That's the last time the compound statement is executed since the logical expression evaluates to false - the JVM will move on to the statement following the `while` statement (line 14) where normal sequential execution resumes. The output follows:



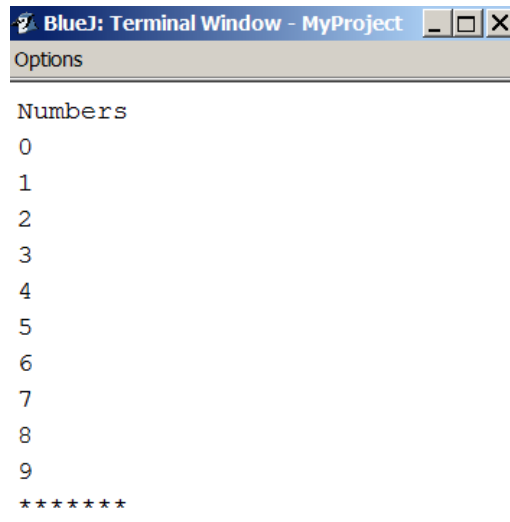


Figure 3.1: Output from Numbers0To9

**Example 2**

Consider another program which displays the digits of a positive number provided by the user. The program includes a scanner object in line 7 that is used to get input from the user via the keyboard. Lines 14 to 19 form a `while` loop where

- Line 15: the rightmost digit is obtained using the expression `number % 10`
- Line 16: the digit is displayed
- Line 17: the value of `number` is reduced by a factor of 10 using the expression `number / 10`
- Line 18: `number` is displayed

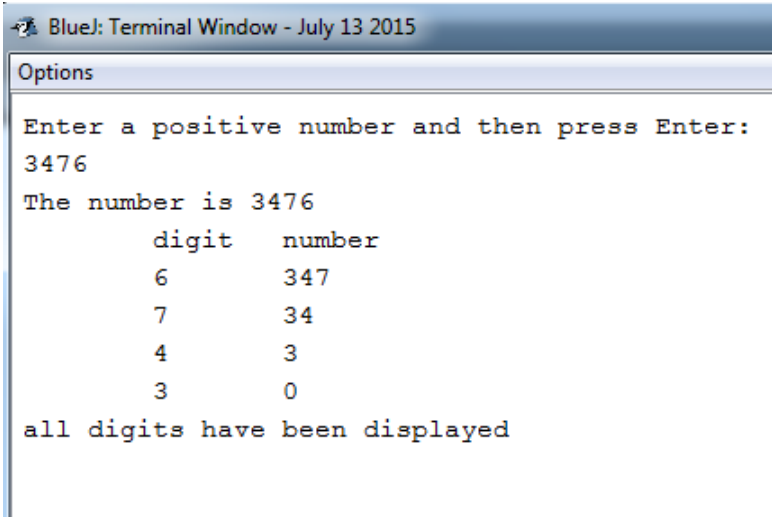
Eventually the value of `number` will be reduced to 0 and the loop terminates.

Listing 3.2: Display digits

```
1 import java.util.Scanner;
2 public class DisplayDigits
3 {
4     public static void main(String[] args)
5     {
6         // Arrange to use a scanner object for
7         // keyboard input
8         Scanner keyboard = new Scanner(System.in);
9         // Prompt the user for a positive number
10        System.out.println("Enter a positive number "
11        + "and then press Enter: ");
12        int number = keyboard.nextInt();
13        System.out.println("The number is "+number);
14        System.out.println("\tdigit\tnumber");
15        while (number > 0){
16            int digit = number % 10;
17            System.out.print("\t"+digit);
18            number = number / 10;
19            System.out.println("\t"+number);
20        }
21        System.out.println("end of list");
22    }
23 }
```

An example of output follows where the user entered the value 3476:

### Output from DisplayDigits



```
BlueJ: Terminal Window - July 13 2015
Options
Enter a positive number and then press Enter:
3476
The number is 3476
      digit  number
        6    347
        7    34
        4     3
        3     0
all digits have been displayed
```

### Nesting statements

The statement executed repeatedly can be any Java statement including another `while` (or any other statement discussed in this chapter).

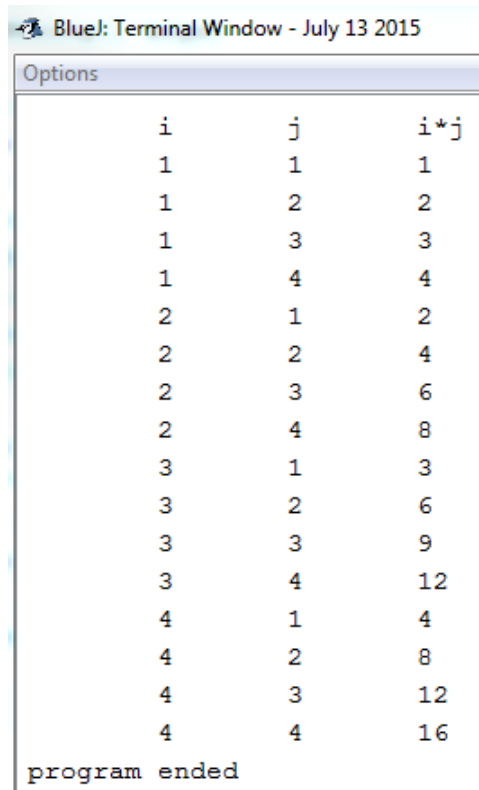
### Example 3

The program in Listing 3.3 displays the product  $i*j$  for  $i$  and  $j$  where both  $i$  and  $j$  take on values 1 through 4. The outer `while` (lines 15-23) executes 4 times, the first time with  $i$  as 1 and the next time with  $i$  as 2, then 3, and then 4. The inner `while` (lines 18-21) is executed entirely for each value of  $i$ ; that is, for each value of  $i$ , the variable  $j$  takes on the value 1, then 2, then 3, then 4. Note the indentation in the program: each line within the outer `while` is indented the same amount, and each line within the inner `while` is indented a further amount.

Listing 3.3: Nesting one `while` inside another while

```
1  /**
2   * Program with one while inside another while.
3   * The program prints i, j, and i*j
4   * where i varies from 1 to 4 and
5   * where j varies from 1 to 4
6   */
7  public class NestedWhiles
8  {
9      public static void main(String[] args)
10     {
11         int i, j;
12         System.out.println("\ti\tj\ti*j");
13         // i takes on values 1,2,3,4
14         i = 1;
15         while (i < 5){
16             j = 1;
17             // j takes on values 1,2,3,4
18             while (j < 5){
19                 System.out.println("\t"+i+"\t"+j+"\t"+(i*j));
20                 j = j + 1;
21             }
22             i = i + 1;
23         }
24         System.out.println("program ended");
```

```
25     }  
26 }
```



```
BlueJ: Terminal Window - July 13 2015  
Options  
i      j      i*j  
1      1      1  
1      2      2  
1      3      3  
1      4      4  
2      1      2  
2      2      4  
2      3      6  
2      4      8  
3      1      3  
3      2      6  
3      3      9  
3      4      12  
4      1      4  
4      2      8  
4      3      12  
4      4      16  
program ended
```

Figure 3.2: Output from NestedWhiles

**Exercises**

1. What happens when a user enters the value 0 when DisplayDigits is executed?
2. What happens when a user enters a negative value when DisplayDigits is executed?
3. What happens when a user enters something that is not an integer when DisplayDigits is executed?
4. Write a program that will sum the digits of a number. For example if the number is 124, then the sum of its digits is  $7 = 1+2+4$ .
5. Write a program that obtains integers from the user and displays their total. The program keeps getting integers until the user enters a value less than zero or greater than 100.
6. Write a program that will sum the integers from -100 to 100. Note the answer you expect is a sum equal to 0.
7. Write a program that converts from Celsius to Fahrenheit for Celsius values starting at -40 and going up +40 in increments of 1.
8. Write a program that converts from Fahrenheit to Celsius for Fahrenheit values starting at -40 and going up +40 in increments of 1.
9. Write a program to convert from Euro Dollars to US Dollars for Euros ranging from 100 to 1,000 in steps of 100. Prompt the user for the exchange rate for converting Euros to US dollars. At the time of writing the exchange rate was 1.12; that is, 1 Euro was worth 1.12 US dollars.
10. Consider the calculation of  $n$  factorial defined as:

$$n! = 1 \times 2 \times 3 \times \dots \times n \quad \text{where } n > 0$$

Use a `while` to calculate  $n!$  Prompt the user for the value of  $n$ .

11. How many times is the print statement in Listing 3.3 (line 19) executed?
12. Modify the program in Listing 3.3 so that  $i$  and  $j$  vary from 1 to 10.

### Autoincrement

Because statements that increment a variable's value, such as `i = i + 1;` are so common Java has a special unary operator `++` for this. The statement `i++;` has the same effect as the above assignment statement. `++` is a unary operator (takes one operand). The operand can be before or after the `++`. The difference relates to when the increment occurs which is only relevant in more complex expressions.

Java has a similar operator, `--`, which has the effect of decrementing the value of a variable, and so the following two statements are equivalent:

```
count = count - 1;
count--;
```

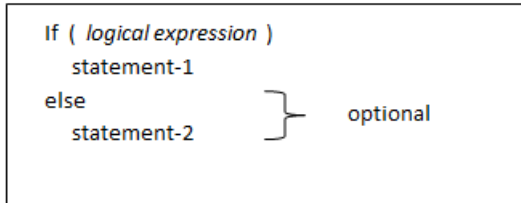
### Exercises

13. Modify the program in Listing 3.3 to use the `++` operator.
14. Use nested whiles to print a  $4 \times 4$  times-table. The times-table should appear as follows

	1	2	3	4
1	1	2	3	4
2	2	4	6	8
3	3	6	9	12
4	4	8	12	16

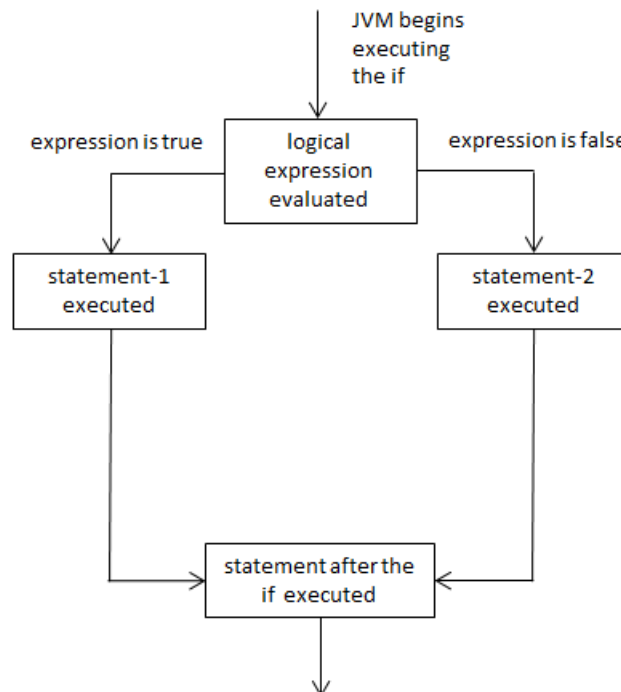
### 3.3 if

The structure of an `if` statement is shown below. The `else` and `statement-2` are optional - we say the if statement has an optional *else* clause. Statement-1 is sometimes referred to as the *then* clause.



When the JVM executes an if statement, the JVM will first evaluate the logical expression. If the expression is true then statement-1 is executed; if the expression is false then statement-2, if present, is executed. The if statement conditionally executes either statement-1 or statement-2. The JVM process can be visualized as:

#### How the JVM executes an if



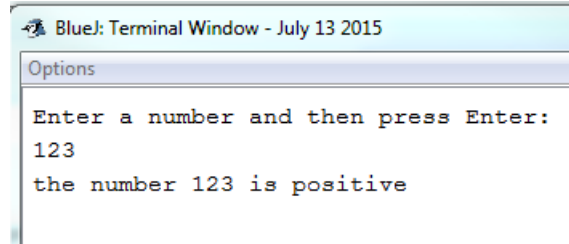


### Example 1

Suppose one needs to display one of two possible messages. To do this we can use an `if` with a then clause and an else clause. Consider the following program that displays one of two messages depending on the value of the expression `number > 0`. In lines 9 and 10 the user is prompted for a number; in line 14 the `if` determines to print `"positive"` or `"not positive"` according to the value of `number > 0`. Note that compound statements are used even though it was not necessary - some programmers always code compound statements. The output of a run where the user supplied the value 123 follows.

Listing 3.4: Using an if statement

```
1 import java.util.Scanner;
2 public class PositiveOrNot
3 {
4     public static void main(String[] args)
5     {
6         // Use a scanner object for keyboard input
7         Scanner keyboard = new Scanner(System.in);
8         // Prompt the user for a number
9         System.out.println("Enter a number "
10             +"and then press Enter: ");
11         int number = keyboard.nextInt();
12         System.out.print("the number "+number+" is
13             ");
14         // Display a message if number is positive
15         // or not
16         if (number > 0) {
17             System.out.println("positive");
18         }
19         else {
20             System.out.println("not positive");
21         }
22     }
23 }
```

**Using an if statement.**

```
Blue: Terminal Window - July 13 2015
Options
Enter a number and then press Enter:
123
the number 123 is positive
```

**Exercises**

15. Write a program that obtains a number from the user and displays whether the number is an even number or an odd number. Note the expression `number % 2` equals 0 if `number` is even and 1 if `number` is odd.
16. Write a program that obtains two numbers from the user and displays the larger of the two numbers.

## Nesting statements

The syntax of the `if` statement provides for the conditional execution of any Java statement, including other `if` statements, `whiles`, etc.

### Example 2

Suppose we need to handle monetary transactions and the program operates in a country where there are no pennies in circulation. In this case cash transactions will be rounded to the nearest nickel; electronic transactions are for the exact amount but there is a surcharge of 25 cents.

Consider the program in Listing 3.5 where the user is prompted to supply two things: the nature of the transaction (cash vs debit card) and the amount of the transaction. In the section on `doubles` we discussed the use of an integer data type for monetary transactions, and so the amount of a transaction is in pennies. In line 12 `int` variables are defined to hold the amount. In line 14 the user is prompted for the type and cost of a purchase.

The structure of the code involves the use of nested `if` statements. The outer `if` (lines 19 to 26) determines if payment is by cash or debit card. The *then* clause (lines 20-24) handles a cash payment and the *else* clause (lines 25-26) handles a debit card payment. For the case of a cash payment there is an inner `if` (lines 20 to 23) nested inside the *then* clause that rounds the cost up or down to the nearest nickel. The *else* clause of the outer adds the additional charge for using a debit card.

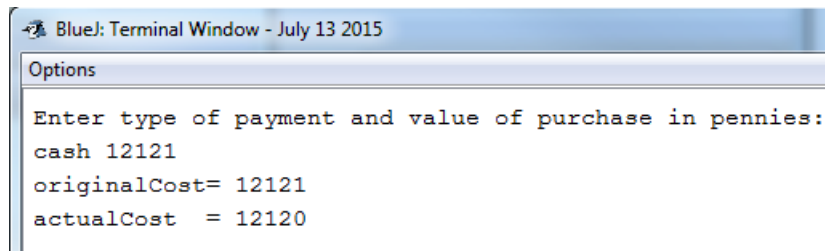
Following the listing is sample output showing the result for a cash transaction for \$121.21.

Listing 3.5: Using an if statement

```
1 import java.util.Scanner;
2 /**
3  * Determine value of payment to be received from
4  * customer
5  * based on whether or not it is cash payment.
6  * Cash payments are rounded off to the nearest
7  * nickel and
8  * debit card payments have a surcharge of 25 cents.
9  */
10 public class RoundCostUpDown
```

```
9 {
10     public static void main(String[] args)
11     {
12         int originalCost, actualCost;
13         String typePayment;
14         System.out.println("Enter type of payment
15             and "
16             +"value of purchase in pennies: ");
17         Scanner kb = new Scanner(System.in);
18         typePayment = kb.next();
19         originalCost = kb.nextInt();
20         if (typePayment.equals("cash")) {
21             if (originalCost % 5 < 3)
22                 actualCost = originalCost -
23                     originalCost%5;
24             else
25                 actualCost = originalCost + (5 -
26                     originalCost%5);
27         }
28         else
29             actualCost = originalCost + 25;
30         System.out.println(originalCost+"
31             "+actualCost);
32     }
33 }
```

### Using nested if statements.



```
Blue: Terminal Window - July 13 2015
Options
Enter type of payment and value of purchase in pennies:
cash 12121
originalCost= 12121
actualCost = 12120
```

**Example 3**

Sometimes the information we need to implement in a Java program is given by a table structure. For instance suppose we have the following table that is to be used to determine the appropriate numeric grade to be given for a specific letter grade. Consider the table:

letter grade	numeric grade
A	4
B	3
C	2
D	1
F	0

If a person is given a letter grade, its a simple matter for that person to find the grade in the letter grade column and look across to determine the numeric grade. However, it may not be obvious how to do this in a Java program. We will consider 3 different ways this might be coded, the last of which we would say is the preferred approach.

A first approach is to code an if statement for each line of the table where the logical expression relates to the letter grade value in the line. For instance the statement

```
if (letterGrade.equals("A"))
    numericGrade = 4.0;
```

will assign `numericGrade` the value 4.0 when the letter grade is "A". A program needs similar statements for the other letter grades. For example at most one of the following logical expressions will evaluate to `true`:

```
if (letterGrade.equals("A"))
    numericGrade = 4.0;
if (letterGrade.equals("B"))
    numericGrade = 3.0;
if (letterGrade.equals("C"))
    numericGrade = 2.0;
```

If you examine the program in Listing 3.6 you will see the variable `numericGrade` initialized to 0.0 and so there are just 4 `if` statements to catch "A", "B", "C" and "F".

Listing 3.6: Using an if statement

```
1 import java.util.Scanner;
2 /**
3  * Determine a numeric equivalent to a letter grade.
4  * Note the standard indentation of extra spaces.
5  */
6 public class LetterGradeToNumericGrade1
7 {
8     public static void main(String[] args)
9     {
10         String letterGrade;
11         // default value for numericGrade
12         //           corresponds to "F"
13         double numericGrade=0.0;
14         System.out.println("Please enter letter
15                             grade:");
16         Scanner kb = new Scanner(System.in);
17         letterGrade = kb.next();
18         if (letterGrade.equals("A"))
19             numericGrade = 4.0;
20         if (letterGrade.equals("B"))
21             numericGrade = 3.0;
22         if (letterGrade.equals("C"))
23             numericGrade = 2.0;
24         if (letterGrade.equals("D"))
25             numericGrade = 1.0;
26         System.out.println(letterGrade+" is
27                             equivalent to "+numericGrade);
28     }
29 }
```

When this first version executes every logical expression is evaluated. This can be avoided to some degree if we use nested `ifs` - we can avoid some unnecessary evaluations.

Consider the program in Listing 3.7. When you review this program note the nesting of the `ifs` - each *else* clause contains an `if`. Only a minimum number of logical expressions are evaluated. For instance if the letter grade is "B" only two logical expressions are evaluated.

Listing 3.7: Using an if statement

```
1 import java.util.Scanner;
2 /**
3  * Determine a numeric equivalent to a letter grade.
4  * Note the standard indentation of extra spaces
5  * for nested if statements.
6  */
7 public class LetterGradeToNumericGrade2
8 {
9     public static void main(String[] args)
10    {
11        String letterGrade;
12        double numericGrade;
13        System.out.println("Please enter letter
14                           grade:");
15        Scanner kb = new Scanner(System.in);
16        letterGrade = kb.next();
17        if (letterGrade.equals("A"))
18            numericGrade = 4.0;
19        else
20            if (letterGrade.equals("B"))
21                numericGrade = 3.0;
22            else
23                if (letterGrade.equals("C"))
24                    numericGrade = 2.0;
25                else
26                    if (letterGrade.equals("D"))
27                        numericGrade = 1.0;
28                    else
29                        numericGrade = 0.0;
30        System.out.println(letterGrade+" is
31                           equivalent to "+numericGrade);
32    }
33 }
```

The indentation you see in the above program is the standard way to show one control structure embedded in another control structure.

However, when there are nested `ifs` and when the logical expressions differ only in the value being tested: "A", "B", ..., a Java programmer can change



the indentation. Each of the cases being tested for are essentially the same and to stress that, a programmer would alter the indentation to that shown below in Listing 3.8. Then, the nested `ifs` are referred to as an *if else-if* structure.

Listing 3.8: Using an if statement

```
1 import java.util.Scanner;
2 /**
3  * Determine a numeric equivalent to a letter grade.
4  * Note how "else if" appears on one line
5  * and how they are aligned.
6  */
7 public class IfElseIfIndentation
8 {
9     public static void main(String[] args)
10    {
11        String letterGrade;
12        double numericGrade;
13        System.out.println("Please enter letter
14                           grade:");
15        Scanner kb = new Scanner(System.in);
16        letterGrade = kb.next();
17        if (letterGrade.equals("A"))
18            numericGrade = 4.0;
19        else if (letterGrade.equals("B"))
20            numericGrade = 3.0;
21        else if (letterGrade.equals("C"))
22            numericGrade = 2.0;
23        else if (letterGrade.equals("D"))
24            numericGrade = 1.0;
25        else
26            numericGrade = 0.0;
27        System.out.println(letterGrade+" is
28                           equivalent to "+numericGrade);
29    }
30 }
```

Later on in the section on the `switch` statement we will see yet another way to determine the pertinent line of the grade table.

**Exercises**

17. Consider how a numeric grade could be translated into a letter grade, as defined in this table:

range	grade
80-100	A
70-79	B
60-69	C
50-59	D
0-49	F

Given a mark, its a simple matter to figure out which range it falls into and determine the corresponding grade. Write a program which obtains a numeric value and translates that into a letter grade. Consider using statements of the form: `if ( mark > ... )`

18. Modify your program for the above question so that it validates the mark obtained from the user to ensure the value is in the range [0, 100].
19. Write a program that obtains 10 numbers from the user and then displays the largest of these numbers. Control the input using a `while` and nest an `if` inside the `while`.

### 3.4 for

The `for` statement is commonly used where there is a need for a statement to be executed a specific number of times. This type of looping construct is sometimes called a *counted* loop. The syntax of the `for` statement we consider here is

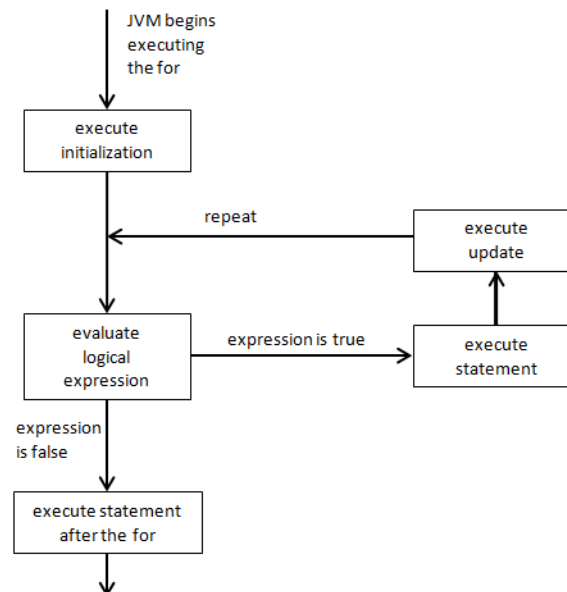
```
for ( initialization; logical expression; increment )
    statement
```

Contrasting the `for` syntax to the `while` syntax:

1. The keyword beginning the statement is **for**
2. Instead of a single logical expression inside parentheses there are three components separated by semi-colons.
  - (a) An initialization section
  - (b) A logical expression (just as the `while` has)
  - (c) An increment section

#### JVM and the for

The JVM executes a `for` as shown:



The `for` is heavily used in Java programming. We will use several examples to show its application.

### Example 1

Below we use a `for` to manage a loop that displays the numbers from 0 to 9. You should compare this to `Numbers0To9` in Section 3.2.

Listing 3.9: Using `for`

```
1 /**
2  * Display numbers 0 to 9 using a for
3  */
4 public class Numbers0To9WithFor
5 {
6     public static void main(String[] args)
7     {
8         System.out.println("Numbers");
9         for (int count=0; count < 10; count++ )
10            System.out.println(count);
11        System.out.println("*****");
12    }
13 }
```

Comparing the above program to `Numbers0To9` in section 3.2 we note:

1. The initialization component declares and initializes the variable *count*. Because it is declared inside the `for`, this variable *count* is known only here inside the `for`. This is known as its *scope*.
2. The second component is the logical expression - the loop executes as long as this evaluates to `true`
3. The update component is an autoincrement for *count*.

Notice how the lines that comprise the `for` statement are indented to enhance readability. This program displays the values 0, 1, ...9 and so it should be evident that *count* takes on values 0, 1, ...9, and that the `for` executes 9 times.

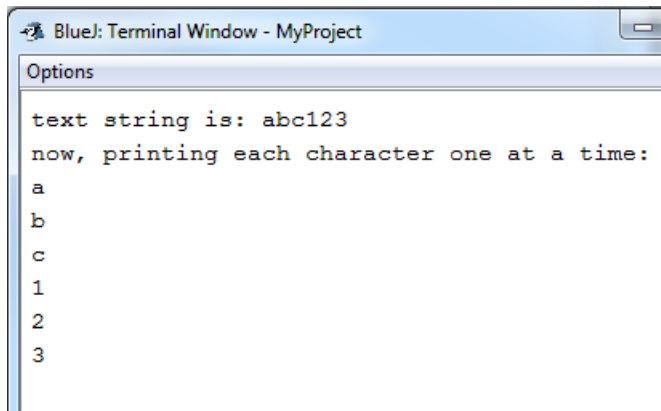
The `for` statement is the preferred programming idiom for a counted loop.

**Example 2**

A common situation where a counted loop arises in the processing of strings. A string of text comprises individual characters. The `String` method `length()` returns the length of a string, and the `charAt(...)` is used to access individual characters. The argument provided to `charAt(...)` is the index of a character within the string - the index of the first character is 0, and the index of the last character is `length()-1`. Consider the following program that displays the characters comprising a string one-by-one. To do this, the program has a `for` loop (lines 14-17) that executes once for each character in the string.

Listing 3.10: Display characters in a string one-by-one.

```
1  /**
2   * Displays a text string character-by-character.
3   * Get individual characters using the charAt(...)
4     method
5   */
6  public class GetIndividualCharacters
7  {
8     public static void main(String[] args){
9         // a string of characters
10        String text = "abc123";
11        // the length (number of characters) of the
12        string
13        int textLength = text.length();
14        System.out.println("text string is: "+text);
15        System.out.println("now, each character
16        one-by-one");
17        for (int i=0; i<textLength; i++){
18            char c = text.charAt(i);
19            System.out.println(c);
20        }
21    }
22 }
```

A terminal window titled "Blue: Terminal Window - MyProject" with an "Options" tab. The terminal displays the following text:

```
text string is: abc123
now, printing each character one at a time:
a
b
c
1
2
3
```

Table 3.1: Process individual characters of a string

### Nested statements

The `for` statement contains a statement to be repeated. This statement can be any Java statement. Consider the following example where an `if` statement appears inside a `for` statement.

### Example 3

This program examines a line of text and counts the number of times `'a'` appears. The program uses the `String` method `length()` to obtain the length of a text string and the method `charAt(...)` to obtain an individual character within a text string. The program has a `for` loop (lines 17-20) that executes once for each character in the text string; the `for` contains an `if` where the current character is compared to `'a'`.

Listing 3.11: Counting lower case alphabetic characters

```
1 import java.util.Scanner;
2 /**
3  * Count the number of lowercase 'a's
4  * in a line provided by the user.
5  */
6 public class CountLetters
7 {
8     public static void main(String[] args)
9     {
10         String text;
11         System.out.println("Enter text: ");
12         Scanner kb = new Scanner(System.in);
13         text = kb.nextLine();
14         int count = 0;
15         for (int i=0; i<text.length(); i++){
16             if (text.charAt(i) == 'a')
17                 count++;
18         }
19         System.out.println("The line contains
20                             "+count
21                             +" a\'s");
22     }
```



### Example 4

The program in Listing 3.12 below displays the product  $i*j$  for  $i$  and  $j$  where both  $i$  and  $j$  take on values 1 through 4. The output produced is the same as Example 3 in the section on the `while` statement.

The outer `for` (lines 10-13) executes 4 times, the first time with  $i$  as 1 and the next time with  $i$  as 2, then 3, and then 4. The inner `for` (lines 11-12) is executed entirely for each value of  $i$ ; that is, for each value of  $i$ , the variable  $j$  takes on the value 1, then 2, then 3, then 4.

Note the indentation in the program: each line within the outer `for` is indented the same amount, and each line within the inner `for` is indented a further amount.

Listing 3.12: Using nested for statements

```
1  /**
2   * Print values of i, j , and i*j where
3   * i varies from 1 to 4 in increments of 1, and
4   * where j varies from 1 to 4 in increments of 1.
5   */
6  public class NestedFor
7  {
8      public static void main(String[] args){
9          System.out.println("\ti\tj\ti*j");
10         for (int i=1; i<=4; i++){
11             for (int j=1; j<=4; j++)
12                 System.out.println("\t"+i+"\t"+j+"\t"+(i*j));
13         }
14         System.out.println("program ended");
15     }
16 }
```

Options		
i	j	i*j
1	1	1
1	2	2
1	3	3
1	4	4
2	1	2
2	2	4
2	3	6
2	4	8
3	1	3
3	2	6
3	3	9
3	4	12
4	1	4
4	2	8
4	3	12
4	4	16

program ended

Table 3.2: Display i, j, and i\*j

**Example 5**

In this example we create a  $5 \times 5$  times table. This table comprises rows and columns where the entry at the intersection the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column is the product of  $i$  and  $j$ ,  $i * j$ :

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

We develop this example in two steps. In the first step we simply produce the values that will appear as entries in the times table, and in the second step we will see it properly formatted with column and row headings.

**... Step 1, products for the times table**

This version produces all the values for the first row, then for the second row, etc. This will require an *outer for* controlling the row number and an *inner for* controlling the column number:

```

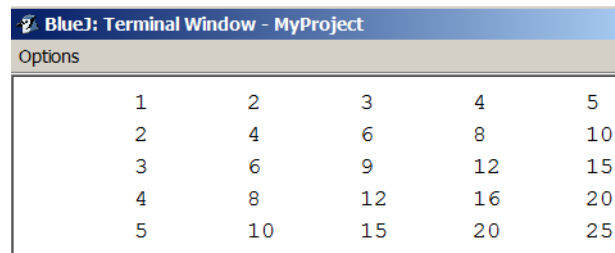
for (int i=1; i<=5; i++){
    for (int j=1; j<=5; j++){
        System.out.print(i*j);
        ...
    }
}

```

In the above the *outer for* uses the variable  $i$  to ensure the loop controlling rows executes 5 times, and the *inner for* uses the variable  $j$  to ensure this loop executes 5 times, once for each column. As the *inner loop* is executed completely for each value of  $i$ , the print statement in the *inner loop* executes a total of  $5 \times 5 = 25$  times. Now, consider the following program where nested *for*s are used to produce values for the times table.

Listing 3.13: Step 1 products for the times table.

```
1 /**
2  * Produce values for a 5x5 times table using
3  * nested for statements
4  */
5 public class TimesTableStep1
6 {
7     public static void main(String[] args)
8     {
9         // variable i represents row i
10        for (int i=1; i<=5; i++) {
11            // variable j represents row j
12            for (int j=1; j<=5; j++) {
13                // multiply i and j
14                // print(...) keeps all values for
15                // i on same line
16                System.out.print("\t"+i*j);
17            }
18            System.out.println();
19        }
20    }
21 }
```



BlueJ: Terminal Window - MyProject					
Options					
1	2	3	4	5	
2	4	6	8	10	
3	6	9	12	15	
4	8	12	16	20	
5	10	15	20	25	

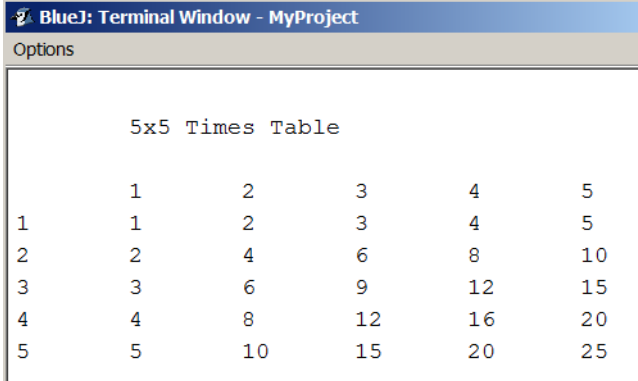
Table 3.3: Generating products.

**...Step 2, formatting the times table**

The completed program and its output are shown next. To reasonably format the times table we have added a title for the output (line 10), column headings (lines 12-14). Then, for each row of the table, a row heading is printed (line 19). Note the mixed use of the `print()` and `println()` methods.

Listing 3.14: Step 2 formatted times table.

```
1  /**
2   * 5x5 times table with column and row headings
3   * using nested for statements
4   */
5  public class TimesTableWithFors
6  {
7      public static void main(String[] args)
8      {
9          // heading and column headings
10         System.out.println("\n\t5x5 Times Table");
11         System.out.println();
12         for (int i=1; i<=5; i++)
13             System.out.print("\t"+i);
14         System.out.println();
15
16         // Compute and print rows of times table
17         for (int i=1; i<=5; i++){
18             // row heading and row contents
19             System.out.print(i);
20             for (int j=1; j<=5; j++)
21                 System.out.print("\t"+i*j);
22             System.out.println();
23         }
24     }
25 }
```



A terminal window titled "BlueJ: Terminal Window - MyProject" with an "Options" menu. The window displays a 5x5 multiplication table. The title bar is blue with white text. Below it is a grey bar with the word "Options". The main area is white with black text. The table is centered and has the title "5x5 Times Table" above it. The rows and columns are numbered 1 to 5. The values are the products of the row and column numbers.

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Table 3.4: Formatting the times table

## Exercises

20. Modify the program in Example 1 to display the value of `count` in line 11 outside the scope of the `for`. Does your program compile? What message do you get from the compiler?
21. Consider Example 1 again. The variable `count` is defined in the `for` statement and so the scope of `count` is the `for` statement. Modify the example properly in order to display the value of `count` in the print statement (line 11). To do this you must declare `count` before the `for` statement, as in:

```
int count;  
for (count=0; count < 10; count++)  
    ...
```

22. Write a program that prompts the user for 10 values, and then displays the sum, the average, the minimum, and the maximum of those 10 values.
23. Write a program that converts from Celsius to Fahrenheit for Celsius values starting at -40 and going up +40 in increments of 1.
24. Write a program that converts from Fahrenheit to Celsius for Fahrenheit values starting at -40 and going up +100 in increments of 1.
25. Write a program to convert from Euro Dollars to US Dollars for Euros ranging from 100 to 1,000 in steps of 100. Prompt the user for the exchange rate for converting Euros to US dollars. At the time of writing the exchange rate was 1.12; that is, 1 Euro was worth 1.12 US dollars.
26. Write a program that will sum the digits of a number. For example if the number is 124, then the sum of its digits is  $7 = 1+2+4$
27. Write a program that prompts the user for an identification number (e.g. student number, credit card number, etc.). The program must then display each digit of the number.
28. Consider the calculation of  $n$  factorial defined as:

$$n! = 1 \times 2 \times 3 \times \dots \times n \quad \text{where } n > 0$$

Write a program that will list in table form, the value of  $n$  and  $n!$  for  $n = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ . Display  $n$  and  $n!$ . The output will look similar to:

$n$	$n!$
1	1
2	2
3	6
4	24
...	

29. The fibonacci sequence is the following integer sequence:  
 $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$   
 We can define them more generally as:  
 $F_n = F_{n-1} + F_{n-2}$  for  $n > 1$   
 where  $F_1 = 1$  and  $F_0 = 0$   
 Write a program that prompts the user for  $n$  and then displays  $F_n$ .
30. Consider the formula where the value of  $f$  depends on  $x$ :  
 $f(x) = 3 + 5x - 7x^2 + 3x^3 + 6x^4$   
 For computational purposes we can re-express the formula as:  
 $f(x) = 3 + x(5 + x(-7 + x(3 + 6x)))$   
 which reduces the total number of calculations. Write a program to evaluate  $f(x)$  for  $x = 0, 1, 2, 3, \dots, 10$
31. Modify the Times Table example so that it produces an  $n \times n$  times table. Prompt the user for the value of  $n$ .
32. Write a program that reads an integer value and then displays that number of asterisks. For instance, if the value read is 11 then the output is:  
 \*\*\*\*\*
33. Write a program that reads an integer value representing the number of students majoring in Mathematics and then another value representing the number of students majoring in Statistics. The program then displays two lines of asterisks to illustrate the number of students majoring in those subject areas. For example if the values were 11 and 15 the output would be:  
 Mathematics \*\*\*\*\*  
 Statistics \*\*\*\*\*



34. Modify the previous program so that it reads 5 pairs of values, where each pair comprises a major (a text string) and the number of students in that major (an integer). For example if the input was

```
Mathematics 14
Statistics 15
English 25
French 15
Geology 10
```

the output would be:

```
Mathematics *****
Statistics *****
English *****
French *****
Geology *****
```

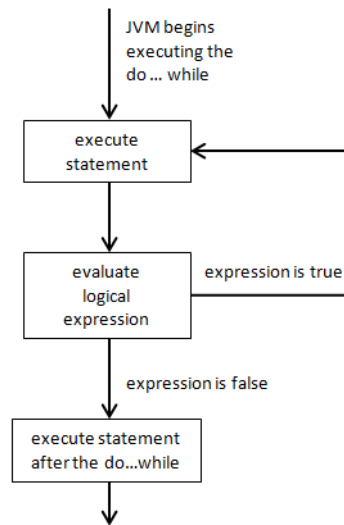
### 3.5 do ...while

The `do ...while` statement is useful when it is known the loop body must execute at least once. The syntax of the `do ...while` statement is

```
do statement while ( logical expression ) ;
```

#### JVM and the do ...while

The JVM executes a `do ...while` as shown:



In words we express the process followed by the JVM as:

1. execute the *statement*
2. evaluate the *logical expression*
3. if the expression is true then go back to step 1, otherwise carry on with the statement following the `do ...while`

When one compares this control structure to other Java control structures (compound, `if`, `for`, and `switch` statements) there is one difference that stands out: the semi-colon at the end. This semi-colon is necessary - if you remove it the compiler will not recognize your `do ...while` and the program will not be executable.

### Example 1

Let us consider a simple program to display the numbers from 0 to 9 using a `do ...while`. Lines 11 and 12 form the compound statement that is executed repeatedly. The process of executing the `do ...while` is straightforward:

1. execute lines 11 and 12
  - (a) display a number
  - (b) increment the value of `count`
2. evaluate `count < 10` in line 14, and if the expression is `true` then repeat these steps starting at step 1.

Note the semi-colon at the end of line 14.

Listing 3.15: Display numbers 0 ...9.

```
1 /**
2  * Display the numbers from 0 to 9.
3  */
4 public class Numbers0To9UsingDoWhile
5 {
6     public static void main(String[] args)
7     {
8         int count = 0;
9         System.out.println("Numbers");
10        do{
11            System.out.println(count);
12            count = count + 1;
13        }
14        while ( count < 10 );
15        System.out.println("*****");
16    }
17 }
```

**Example 2**

Consider a program someone can use to develop their addition skills. The program should behave as follows:

1. Give the user two numbers to add.
2. Evaluate the user's answer.
3. Prompt the user whether or not to repeat.
4. Go back to step 1 if the user wants to try another addition.

As well as including a `do ...while` this program makes use of three Java classes: `Random` (Chapter 4.1), `Scanner` (Chapter 4.3), and `String` (Section 2.5). Consider these points:

1. (line 14) The variable *generator* references an instance created from the `Random` class. The `Random` class has many methods that can be used to generate random values; in lines 20 and 21 there is the expression `generator.nextInt(10)+1` to obtain random values between 1 and 10.
2. The variable `kb` defined in line 13 references an instance of the `Scanner` class and is used to manage input from the standard input device, the keyboard.
3. The `String` variable *response* defined in line 15 is used to hold the user's response to the prompt `"To try again enter Y"`. The program uses two methods from the `String` class: `toLowerCase()` and `equals(...)`.
  - (a) (line 31) `toLowerCase()` converts the user's response to all lower case characters
  - (b) (line 33) `equals("y")` is used to compare the user's response in lower case to the lower case "y" and evaluates to `true` if the user's response had been "Y" or "y". If `true` then execution resumes at line 20, otherwise execution of the `do ...while` terminates and execution resumes at line 34.

Listing 3.16: Developing addition skills.

```
1 import java.util.Scanner;
2 import java.util.Random;
3 /**
4  * Give the user two random integers to add.
5  * Inform user if their answer is correct.
6  * Prompt the user to try again.
7  * The user always attempts one addition.
8  */
9 public class Additions
10 {
11     public static void main(String[] args)
12     {
13         Scanner kb = new Scanner(System.in);
14         Random generator = new Random();
15         String response;
16         System.out.println("Welcome. "
17             + "Try some additions:");
18         do
19         {
20             int n1 = generator.nextInt(10)+1;
21             int n2 = generator.nextInt(10)+1;
22             System.out.println(n1+"+"+n2+"=?");
23             int answer = kb.nextInt();
24             if (answer == n1+n2)
25                 System.out.println("Correct!");
26             else System.out.println(
27                 "Sorry that is not correct "
28                 + "... the sum is "+(n1+n2));
29             System.out.println(
30                 "To try again enter Y: ");
31             response = kb.next( );
32             response = response.toLowerCase();
33         }
34         while ( response.equals("y"));
35         System.out.println("Goodbye");
36     }
```

**Exercises**

35. Modify Example 1 so the program will display the sum of the numbers from 0 to 9.
36. Modify Example 2 so the user gets a report when the program ends: the number of correct and the number of incorrect answers.
37. Write a program that chooses a random number between 1 and 100, and then asks the user to guess what the number is. If a user guesses the number the program informs the user and stops, otherwise the program informs the user if they too high or too low, and the user is prompted to guess again. Use a `do ...while` to control the iteration.
38. One can simulate the tossing (rolling) of a six-sided die through the use of the `Random` class. In many games two dice are thrown and the player's turn depends on the total value of the two dice. The following code instantiates two dice that can be used in a game:

```
Random die1 = new Random();
Random die2 = new Random();
```

Now, if we want to roll the two dice and know the total thrown we could use:

```
int toss1 = die1.nextInt(6)+1;
int toss2 = die2.nextInt(6)+1;
total = toss1 + toss2;
```

In some games a player rolls the dice at least once. Suppose we want to simulate a player rolling the dice until "snake eyes" are thrown. Snake eyes is the term used to describe a throw where two one's appear. Write a program that uses a `do ...while` to simulate the rolling of two dice. The program must list the totals thrown until "snake-eyes" appear.

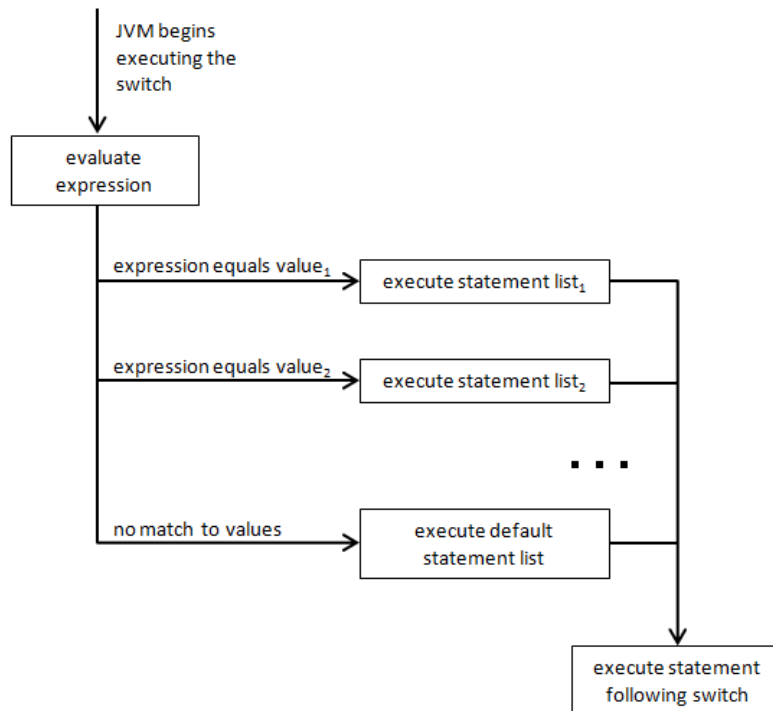
### 3.6 switch

The `switch` statement is a decision structure where one choice, of possibly many, different choices are made. The general structure of the `switch` statement is

```
switch ( expression ) {  
  case value1 : statement list1  
  case value2 : statement list2  
  case value3 : statement list3  
  ...  
  default : default statement list  
}
```

#### JVM and the switch

The JVM executes a `switch` as shown:



The `switch` statement contains a number of `case` groups where each group contains a list of statements. The `switch` statement has an expression which is used to determine where execution continues: If the value of the expression matches a value for a `case` group, then execution continues with the `case` group's statement list. If the expression does not match any of the specified values then the default statement list is executed, if one exists.

Each statement list has a well-defined starting point. The statements of a statement list are executed one-by-one until the JVM encounters a `break` statement. When a `break` is executed, the execution of the `switch` terminates and the statement following the `switch` is executed next (normal sequential execution of statements resumes). If a statement list does not have a `break` statement then the the next statement list will execute, and so on until either a `break` is encountered or the end of the switch is reached.

The default case group is optional: If the `switch` expression's value does not match a case value and if there is no default case group then the switch statement terminates - and the statement following the `switch` is executed next.

Normal usage of the `switch` is that each case group has a statement list and the last statement of the group is a `break`. However, there are times when it is useful for a statement group to be empty, and there are times when it is useful for a statement group to not have a `break` statement.

Note that when all statement lists end with a `break` statement it is possible to replace a `switch` statement with nested `if` statements where the logical expressions are of the form

```
expression.equals(value) for String expressions, and  
expression == value for other types.
```

The expression and matching values can only be of certain data types that include: `char`, `byte`, `short`, `int`, and the `String` type. Other included types are *enumeration* types and the *wrapper* classes: `Character`, `Byte`, `Short`, `Integer`.



### Example 1

Previously we considered translating a letter grade to a numeric grade using an `if`. In the program below each case group represents a line from the table:

letter grade	grade point
A	4
B	3
C	2
D	1
F	0

In this program a letter grade is obtained from the user. When the `switch` executes the expression `grade` is evaluated and compared to each case value. If the value of `grade` matches a value in some case, then the statement list for that case is executed. If the value of `grade` is not an "A", "B", "C" or "D" the program assumes it must be "F" and so the default case is executed and 0.0 is assigned to `nGrade`.

Listing 3.17: Translate grade to its numeric value.

```
1 import java.util.Scanner;
2 /**
3  * Letter grade translated to a numeric grade.
4  */
5 public class LetterGradeToNumericGradeWithSwitch
6 {
7     public static void main(String[] args)
8     {
9         String grade;
10        double nGrade;
11        System.out.println("Enter letter grade:");
12        Scanner kb = new Scanner(System.in);
13        grade = kb.next();
14        switch (grade) {
15            case "A": nGrade = 4.0;
16                    break;
17            case "B": nGrade = 3.0;
18                    break;
19            case "C": nGrade = 2.0;
20                    break;
```

```
21         case "D": nGrade = 1.0;
22             break;
23         default: nGrade = 0.0;
24     }
25     System.out.println(grade+" --> "+nGrade);
26 }
27 }
```

If this program did not have any `break` statements then every grade would be assigned the same numeric value: 0.0. Consider this code:

```
grade = kb.next();
switch (grade) {
    case "A": nGrade = 4.0;
    case "B": nGrade = 3.0;
    case "C": nGrade = 2.0;
    case "D": nGrade = 1.0;
    default: nGrade = 0.0;
}
System.out.println(grade+" --> "+nGrade);
```

Now suppose `grade` has the value `"A"` then every assignment statement executes and the last one executed is `nGrade = 0.0` so the numeric grade assigned is 0.0.

## Exercises

39. Write a program that prompts the user for a date in the `yyyy mm dd` format and then prints the date in the form *month name dd, yyyy*. For example, if the user provides `2000 01 01` then the program displays `January 1, 2000`.
40. Write a program where the user provides the name of a month and the program displays the number of days in that month. We will ignore the fact that February sometimes has 29 days (in a leap year). In a sense Java lets you combine cases by allowing empty statement lists. For example, April, June, September, and November each have 30 days and so you can write code such as:

```

case "April":
case "June":
case "September":
case "November":
    numDays = 30;
    break;

```

If the switch expression evaluates to any of

```
"April", "June", "September", "November"
```

then `numDays = 30` will be executed.

41. Suppose we need a program that accepts a month followed by a day and then reports the number of days left in the year. Again, we shall ignore the concept of leap year. For example, suppose the user entered:

October 30

As October has 31 days, November has 30 days, and December has 31 days, the number of days left is  $(31 - 30) + 30 + 31 = 62$ .

Incorporate the following type of `switch` where each statement group simply increments a variable and where there are no `break` statements (so statement lists are executed from the selected case until the end of the `switch`). Note the use of the combined assignment operator `+=`.

```

int numberOfDays = 0;
switch (month) {
    case "January":  numberOfDays += 31;
    case "February": numberOfDays += 28;
    case "March":    numberOfDays += 31;
    ...
}

```



## Chapter 4

# Classes in the Java Class Libraries

### 4.1 Random

The `Random` provides a capability to generate pseudorandom values. The term pseudorandom is used because the stream of values we can get are generated algorithmically - if one knows the initial value used and the algorithm, we can predict the sequence of *random* values. The interested reader who wants to learn more about random number generation should consult the book *The Art of Computer Programming Volume 2* [6]. In what follows we will use the word *random* but do remember the values obtained are pseudorandom.

The `Random` class provides methods the programmer can use to generate random values that include boolean, integer, and double types. The `Math` class also has a method `random()` that can be used to generate random double values between 0.0 (inclusive) and 1.0.

In order to generate random values a program must instantiate an object from the `Random` class. There are two constructors for this purpose: one that takes an argument (a seed or initial value) and one that does not (the *no-arg* constructor). The advantage to using a seed is that the stream of values is always the same and this can assist in debugging code. One cannot predict the values to be obtained if the no-arg constructor is used since it bases its' seed on the system time.

**Example 1**

We begin with a simple example to simulate rolling a six-sided die. Traditionally the values of the sides are 1, 2, 3, 4, 5, 6. `Random` has a method `nextInt(...)` that returns an `int` value between 0 and the argument provided. So, if `g` is a `Random` object, then to obtain random values as if one is rolling a six-sided die one uses: `g.nextInt(6)+1`. The following program simulates tossing a die 10 times.

Listing 4.1: Translate grade to its numeric value.

```

1 import java.util.Random;
2 /**
3  * Display 10 rolls of a 6-sided die.
4  */
5 public class RollDie
6 {
7     public static void main(String[] args)
8     {
9         System.out.print("\n\n10 rolls: ");
10        Random g = new Random();
11        for (int i=0; i<10; i++)
12            System.out.print(g.nextInt(6)+1+" ");
13    }
14 }

```

Four sample runs of `RollDie.java`:

RollDie run four times										
10 rolls:	5	5	3	5	6	6	3	6	6	3
10 rolls:	1	1	6	3	6	5	5	3	3	6
10 rolls:	1	4	6	5	4	2	2	5	1	1
10 rolls:	2	3	2	6	4	6	3	5	2	2

## Example 2

Consider the tossing of a coin where one side of the coin is considered a head and the other a tail. There are many approaches one could use, for example:

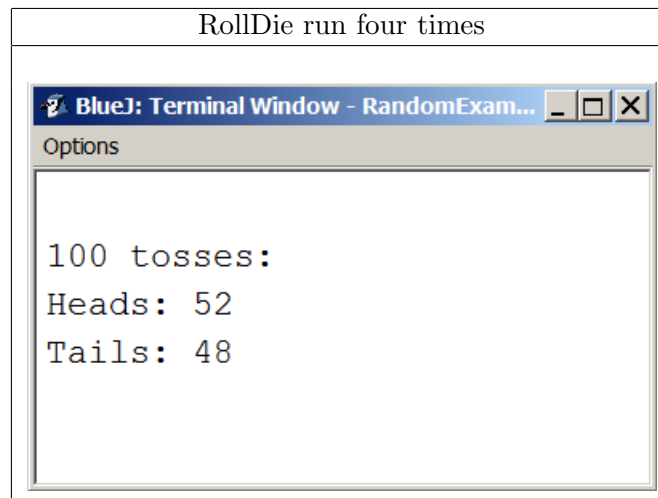
1. `nextInt(2)` generates 0 and 1
2. `nextRandom()` generates `true` and `false`
3. `nextInt()` generates integers - approximately half are negative, the other half positive (or half are even and half are odd).

The next program simulates tossing a coin 100 times and tabulating the number of occurrences for the two outcomes. In the long run we expect the number of heads and the number of tails to be equal, but that's not likely to occur on a single run.

Listing 4.2: Translate grade to its numeric value.

```
1 import java.util.Random;
2 /**
3  * Toss a coin 100 times and tabulate the
4  * number of heads and the number of tails.
5  */
6 public class TossCoin
7 {
8     public static void main(String[] args)
9     {
10         int heads = 0;
11         System.out.print("\n100 tosses: ");
12         Random g = new Random();
13         for (int i=0; i<100; i++)
14             if(g.nextBoolean())heads++;
15         System.out.println("\nHeads: "+heads
16                             +"\nTails: "+(100-heads));
17     }
18 }
```

A result from running TossCoin.java:



The image shows a Java Swing window titled "RollDie run four times". Inside the window is a smaller window titled "BlueJ: Terminal Window - RandomExam...". The terminal window has an "Options" menu and displays the following text:

```
100 tosses:  
Heads: 52  
Tails: 48
```



## Exercises

1. Modify Example 1 using a seed value when instantiating `g`. For example the line

```
Random g = new Random()
```

can be replaced by

```
Random g = new Random(101)
```

Run the program twice and notice the sequence of random numbers is the same both times. Using a seed can be useful if you are having difficulty debugging your program.

2. In the game of craps there are names given to various outcomes of rolling two dice. For example:

Names of rolls	
snakes eyes	two 1s
hard four	two 2s
yo-leven	6 and 5
natural	1 and 6, 2 and 5, 3 and 4

Write a program that will simulate throwing 2 dice until snake eyes occurs. The program must list each throw including the snake eyes.

3. In the standard game of *Pig* players take turns rolling a single die. In a turn a player repeatedly rolls a die according to:
  - If a player rolls a 1, the player scores nothing for that turn and it becomes the next player's turn.
  - If a player rolls any other number, that number is added to the player's turn total and the player's turn continues.
  - If a player chooses to "hold", the player's turn total is added to the player's total score, and it becomes the next player's turn.

Write a program to simulate the rolling of a single die until a 1 turns up. Your program must list each roll.

4. Consider the game of *Pig* again. Write a program to simulate a player's turn where the player's strategy is to continue rolling as long as the turn score is less than 25. That is, the player holds if the turn score is 25 or better. Of course, if a 1 is rolled, the player gets a turn score of 0. Your program must list each roll and at the end of the turn display the turn total.

## 4.2 Character

The `Character` class has many static methods that can be used. Because the methods are static the programmer does not instantiate an object. Instead, when using one of these class methods it must be prefixed with `Character.`, for example `Character.toLowerCase(ch)`. The following table lists some common static methods of the `Character` class:

Method	Description
<code>getNumericValue(...)</code>	Returns the int value that the specified character represents.
<code>isDigit(...)</code>	Determines if the specified character is a digit.
<code>isLetter(...)</code>	Determines if the specified character is a letter.
<code>isWhitespace(...)</code>	Determines if the specified character is white space
<code>toLowerCase(...)</code>	Converts the character argument to lowercase
<code>toUpperCase(...)</code>	Converts the character argument to uppercase

Three examples follow

1. Detecting the type of character
2. Getting the numerical value of a numeric character
3. Validating input

### Example 1

If the data you have is a string then the `String` method `charAt(...)` can be used to access a character at a specific index. When used in conjunction with a `for` statement the characters of a string can be accessed one-by-one. In the following program we access the characters of a string one-by-one and determine the type of each character using the `Character` methods `isDigit()` and `isLetter()`.

Listing 4.3: Types of characters.

```
1 import java.util.Scanner;
2 /**
3  * A string provided by the user is examined
4  * character by character to determine its type.
5  */
6 public class CharacterTypes
7 {
8     public static void main(String[] args)
9     {
10         Scanner kb = new Scanner(System.in);
11         System.out.print("Enter a line: ");
12         String line = kb.nextLine();
13         // characters are examined one-by-one
14         for (int i = 0; i < line.length(); i++){
15             char c = line.charAt(i);
16             if(Character.isLetter(c))
17                 System.out.println(i+"\t"+c
18                                     +"\t\tletter");
19             else if(Character.isDigit(c))
20                 System.out.println(i+"\t"+c
21                                     +"\t\tdigit");
22             else
23                 System.out.println(i+"\t"+c
24                                     +"\t\tother");
25         }
26     }
27 }
```

Below is the output from `CharacterTypes.java` for when the user provides the string `"A$12"`

```
Options
Enter a line: A$12
0      A      letter
1      $      other
2      1      digit
3      2      digit
```

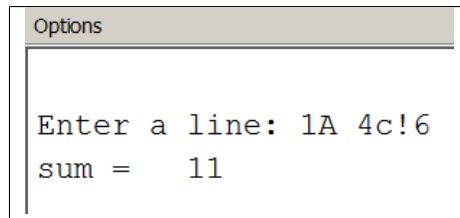
## Example 2

The `Character` method `getNumericValue()` can be used to obtain the decimal value of a character. This program examines the text provided by the user and sums the numeric values of the characters that are digits.

Listing 4.4: Types of characters.

```
1 import java.util.Scanner;
2 /**
3  * The sum of numeric characters is calculated.
4  */
5 public class SumNumericValues
6 {
7     public static void main(String[] args) {
8         Scanner kb = new Scanner(System.in);
9         System.out.print("\nEnter a line: ");
10        String line = kb.nextLine();
11        int sum = 0;
12        // characters are examined one-by-one
13        for (int i = 0; i < line.length(); i++){
14            char c = line.charAt(i);
15            if(Character.isDigit(c)){
16                sum += Character.getNumericValue(c);
17            }
18        }
19        System.out.println("sum = \t"+sum);
20    }
21 }
```

Below is the output from `SumNumericValues.java` for when the user provides the string `"1A 4c!6"` which contains the numeric characters 1, 4, 6.



```
Options
Enter a line: 1A 4c!6
sum = 11
```

### Example 3

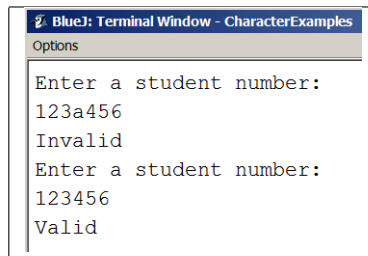
In many situations a user's input must be validated. Suppose a user is prompted for a student number that must comprise only digits. If the user enters invalid characters, and the program were to use the Scanner method `nextInt()`, then the program would crash. Instead the programmer must use the Scanner method `next()` and then analyze the characters to determine if the user entered a correctly formatted value.

Consider the following code that utilizes the Character method `isDigit(...)` to determine if a user has entered a numeric string (a valid student number).

Listing 4.5: Validation of input.

```
1 import java.util.Scanner;
2 /**
3  * A string provided by the user is examined
4  * to determine whether or not it is numeric.
5  */
6 public class ValidateStudentNumber
7 {
8     public static void main(String[] args)
9     {
10         Scanner kb = new Scanner(System.in);
11         System.out.println("Enter a number: ");
12         String number = kb.next();
13         // characters are examined one-by-one
14         boolean valid = true;
15         for (int i = 0; i < number.length(); i++){
16             char c = number.charAt(i);
17             if(! Character.isDigit(c)) valid =
18                 false;
19         }
20         if (valid) System.out.println("Valid");
21         else System.out.println("Invalid");
22     }
23 }
```

Below is the output from `ValidateStudentNumber.java` for two runs of the program.



```
Blue: Terminal Window - CharacterExamples
Options
Enter a student number:
123a456
Invalid
Enter a student number:
123456
Valid
```

## Exercises

5. Java allows char values to be used directly in arithmetic expressions. Modify Example 2 to just add the character instead of its numerical value using a statement such as  
`sum += c;`  
instead of  
`sum += Character.getNumericValue(c);`  
The sum in this case is the sum of the internal representations of those characters.
6. Modify Example 3 so it stops examining characters if it encounters a non-numeric character. Consider using a `for` that begins:  
`for (int i=0; valid && i<number.length(); i++)`
7. Write a program to validate a phone number where the number is expected to be a string of 10 digits. For example if the user entered 2343214567 the number would be valid, but if the user entered ADG3214567 the number would be invalid.
8. The standard US zip code is five digits. Write a program that prompts the user for a zip code and then determines if it is valid or not. To be valid the code must be five characters in length and all characters must be digits.
9. In 1983 the US Postal Service extended zip codes (ZIP+4) to include the five digits of the ZIP code, a hyphen, and four more digits that determine a more specific location within a given ZIP code. Write a program to validate a zip code entered by the user where the user might have entered a standard zip code (5 characters) or a zip+4 code

(10 characters including the dash separating the first 5 digits from the last 4 digits).

10. Sweden has a personal identity number (*personnummer*) that is issued by the Swedish Tax Agency. This identity number has 10 digits with a hyphen between the 6<sup>th</sup> and 7<sup>th</sup> digits, and is such that the 10<sup>th</sup> digit is a check digit. The check digit is calculated using the first 9 digits. A weighted sum of products is calculated as  $\sum(\text{digit}_i \times \text{weight}_i)$  where the weights are 2, 1, 2, 1, 2, 1, 2, 1, 2. However if a product is more than 9 it is replaced by the sum of its digits. The check digit must be equal to 10 minus the last digit (but note that if the last digit of the sum is zero, the check digit is 0). Write a program to verify the user has entered a valid personnummer:
- 10 digits with a dash between the 6<sup>th</sup> and 7<sup>th</sup> digits, and
  - the check digit is correctly based on the first 9 digits.

For example consider the personnummer 811228-9874. To verify the check digit (the last digit, the 4) is correct we need to follow the above procedure. The sum of the weighted products is:

$(8 \times 2) + (1 \times 1) + (1 \times 2) + (2 \times 1) + (2 \times 2) + (8 \times 1) + (9 \times 2) + (8 \times 1) + (7 \times 2)$   
which are:

$(16) + 1 + 2 + 2 + 4 + 8 + (18) + 8 + (14)$

and modifying where the product > 9:

$(1 + 6) + 1 + 2 + 2 + 4 + 8 + (1 + 8) + 8 + (1 + 4)$

we have:

$7 + 1 + 2 + 2 + 4 + 8 + 9 + 8 + 5 = 46$

And finally  $10 - 6 = 4$ . So, the personnummer above is valid.



## 4.3 Scanner

Previously we used a `Scanner` object to obtain data from the user via the standard input device, the keyboard. The input is considered to be a sequence of tokens where tokens are strings separated by delimiters which by default are whitespace. To Java, whitespace includes spaces, tabs, newline, and a few other characters. A `Scanner` object is said to parse the input stream making *tokens* available.

A programmer can specify exactly what constitutes a token. Consider that a program could be reading a file where tokens are separated by commas (e.g. a CSV file that is easily generated from Excel). The interested reader is referred to the Java documentation for more information on how to specify delimiter patterns. For our purposes we use the defaults for a scanner object and so tokens are strings where the strings are delimited by whitespace.

In this text we cover three usages for the `Scanner` class:

- As discussed earlier for handling input from standard input: `System.in`.
- Obtaining tokens from a string.
- Obtaining tokens from a file.

To use the methods in the `Scanner` class we must instantiate a `Scanner`, for example:

- `Scanner s = new Scanner("System.in");`
- `Scanner s = new Scanner(s); //s is of type String`
- `Scanner s = new Scanner(f); //f is of type File`

The `Scanner` class is in the `java.util` package and so programs need to include:

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

We will illustrate the use of a scanner object for reading a file and another for scanning a string.

## Declaring a Scanner for a text file

Recall that each class we create in BlueJ is stored as a file with the .java extension, and the compilation process creates another file with the .class extension. In a BlueJ project there is another file you will have noticed called Readme.txt. The .java files and the .txt file are both text files whereas the class file is a binary file. Text files are human-readable but a class file contains Java *bytecode* and to view and make sense of its contents would be quite difficult.

To read a text file we can declare a `Scanner` object which is associated with that file. Let us consider only files that are in our project. When the file is contained in our project we only need to name it as a string, as in:

```
Scanner f = new Scanner(new File("Readme.txt"));
```

Consider the following table of Scanner methods. We will use a number of these in the examples that follow.

method name	description
<code>next()</code>	Gets next token
<code>nextBoolean()</code>	Gets next token and converts it to an boolean
<code>nextInt()</code>	Gets next token and converts it to an int
<code>nextDouble()</code>	Gets next token and converts it to an double
<code>hasNext()</code>	Returns true if there is at least one more token available, false otherwise
<code>hasNextBoolean(...)</code>	Returns true if there is at least one more token available and that token is of type boolean, false otherwise
<code>hasNextInt()</code>	Returns true if there is at least one more token available and that token is of type int, false otherwise
<code>hasNextDouble()</code>	Returns true if there is at least one more token available and that token is of type double, false otherwise
<code>hasNextLine()</code>	Returns true if there is at least one more line available, false otherwise
<code>nextLine()</code>	Returns an entire line, up to the next end-of-line character and returns the line as a String (the end-of-line character is <i>consumed</i> but it is not part of the return value).

Note that the above methods can result in errors that cause a program to fail. For instance, if a program uses `nextInt()`, but the next token is a character string, then an exception will occur. If a program executes `next()`, but the input stream is empty, then an exception will occur.

### Example 1. Reading Readme.txt

Consider the following program that reads the file `Readme.txt` and displays its lines including line numbers. Note the program has three import statements to direct the compiler to definitions for `Scanner`, `File` and `FileNotFoundException`.

The `File` class itself is quite complex but for our purposes we are just naming the file and instantiating a `File` object. Errors can arise when a program processes a file - the obvious one is trying to read a file that does not exist. The Java file `FileNotFoundException` is associated with that condition. Note the program also contains a `throws` clause - Java requires this and for our purposes here we are declaring that we know this situation might arise.

Two `Scanner` methods used here are:

1. `hasNext()` which returns `true` or `false` according to whether or not there are more tokens to be retrieved.
2. `nextLine()` which retrieves the next line (of course this may retrieve several tokens embedded in one string).

Listing 4.6: Displaying contents of Readme.txt.

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4 /**
5  * Display contents of Readme.txt with line numbers
6  */
7 public class DisplayReadme
8 {
9     public static void main(String[] args)
10    throws FileNotFoundException
11    {
12        Scanner f = new Scanner(
13            new File("Readme.txt"));
14        int i=1;
15        System.out.println(
16            "<<<< File Readme.txt >>>>");
17        while (f.hasNext()){
18            String line = f.nextLine();
19            System.out.println((i++)+" "+line);
20        }
21        System.out.println(
22            "<<<< end of listing >>>>");
23    }
24 }
```

When BlueJ creates Readme.txt it initializes the file with default contents. The programmer can edit this file to store relevant information about the project. If the file has not been edited then it has certain contents by default which are:

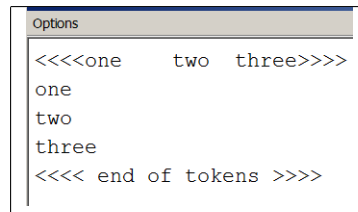
```
Options
<<<< File Readme.txt >>>>
1 -----
2 This is the project README file. Here, you should describe your project.
3 Tell the reader (someone who does not know anything about this project)
4 all he/she needs to know. The comments should usually include at least:
5 -----
6
7 PROJECT TITLE:
8 PURPOSE OF PROJECT:
9 VERSION or DATE:
10 HOW TO START THIS PROJECT:
11 AUTHORS:
12 USER INSTRUCTIONS:
<<<< end of listing >>>>
```

**Example 2. Scanning a string**

You can define a `Scanner` object to process a string with `Scanner` methods. The method `hasNext()` returns `true` if there is another token in the string, and the method `next()` will return the next token in the string. Consider the following program that scans a string obtaining its tokens one-by-one.

Listing 4.7: Display tokens in a string.

```
1 import java.util.Scanner;
2 /**
3  * Display tokens in a string
4  */
5 public class ScanString
6 {
7     public static void main(String[] args)
8     {
9         String sample = "one    two \tthree";
10        Scanner s = new Scanner(sample);
11        System.out.println(
12            "<<<<"+sample+">>>>");
13        while (s.hasNext()){
14            String token = s.next();
15            System.out.println(token);
16        }
17        System.out.println(
18            "<<<< end of tokens >>>>");
19    }
20 }
```



```
Options
<<<<one    two \tthree>>>>
one
two
three
<<<< end of tokens >>>>
```

**Exercises**

11. Modify Example 1 to use `next()` instead of `nextLine()`. Display each token on a separate line.
12. Modify Example 2 to get the value for the variable `sample` from the user.
13. Write a program that prompts the user for 10 `boolean` values. Display the number of `true` values entered by the user.

## 4.4 Math

The class `Math` contains fields for  $\pi$  and  $e$  and methods for performing basic numeric operations including exponential, logarithm, square root, and trigonometric functions.

The methods in the `Math` class are static methods and so one does not instantiate an instance. To use a method you must prefix the method name by `Math..` For example to get the absolute value of an `int` `x` the programmer just codes

```
int y = Math.abs(x);
```

or to determine the circumference of a circle of radius `r`:

```
double circumference = 2.0 * Math.PI * r.
```

Next we list a number of the `Math` method and then we present an example.



The Math Class		
Fields		
field	type	description
E	double	$e$
PI	double	$\pi$
Static Methods		
method	type	description
sin	double	sine of an angle (in radians). e.g. <code>Math.sin(2.5)</code>
cos	double	cosine of an angle (in radians). e.g. <code>Math.cos(2.5)</code>
tan	double	tangent of an angle (in radians). e.g. <code>Math.tan(2.5)</code>
toRadians	double	converts an angle in degrees to an angle in radians. e.g. <code>Math.toRadians(180.0)</code>
toDegrees	double	converts an angle in radians to an angle in degrees. e.g. <code>Math.toDegrees(3.14)</code>
exp	double	Euler's number $e$ raised to a power. e.g. <code>Math.exp(2.5)</code>
log	double	natural logarithm (base $e$ ). e.g. <code>Math.log(2.5)</code>
log10	double	base 10 logarithm. e.g. <code>Math.log10(2.5)</code>
pow	double	Returns the value of the first argument raised to the power of the second argument. e.g. <code>Math.pow(x, y)</code>
random	double	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. e.g. <code>Math.random()</code>
abs	int	returns the absolute value of an int. e.g. <code>Math.abs(i)</code>
abs	double	returns the absolute value of a double. e.g. <code>Math.abs(x)</code>
max	int	returns the larger of two int values. e.g. <code>Math.max(5, 2)</code>
max	double	returns the larger of two double values. e.g. <code>Math.max(5.6, 2.0)</code>
min	int	returns the smaller of two int values. e.g. <code>Math.min(5, 20)</code>
min	double	returns the smaller of two double values. e.g. <code>Math.min(5.3, 20.7)</code>
hypot	double	returns the $\sqrt{x^2 + y^2}$ . e.g. <code>Math.hypot(2.5, 3.3)</code>

**Example 1**

The following program prompts the user for three `int` values and reports the largest of the three.

Listing 4.8: Use `Math.max()` to find largest of 3 values.

```
1 import java.util.Scanner;
2 /**
3  * Prompt for 3 int values and report the largest
4  */
5 public class FindMax
6 {
7     public static void main(String[] args){
8         Scanner kb = new Scanner(System.in);
9         System.out.println(
10             "Please enter 3 int values");
11         int i = kb.nextInt();
12         int j = kb.nextInt();
13         int k = kb.nextInt();
14         int mx = Math.max(i, Math.max(j,k));
15         System.out.println("largest is "+mx);
16     }
17 }
```

**Exercises**

14. Write a program that accepts `int` values until the user enters a zero. The program must display the smallest value greater than zero.
15. Write a program that obtains the radius of a circle from the user and calculates the area of the circle.
16. Write a program that obtains the radius of a sphere from the user and calculates the volume of the sphere.
17. Write a program that obtains the x and y values of a right-angled triangle from the user and calculates the length of the hypotenuse.

## 4.5 Integer

The `Integer` class has many static fields and methods that can be used. Because these are static the programmer does not instantiate an object. Instead, when using one of these class fields or methods it must be prefixed with `Integer.`, for example `Integer.MAX_VALUE`. The following lists some common static fields and methods of the `Integer` class:

Field	Description
<code>MAX_VALUE</code>	A constant holding the maximum value an int can have, $2^{31}-1$ .
<code>MIN_VALUE</code>	A constant holding the minimum value an int can have, $-2^{31}$ .

Method	Description
<code>max()</code>	Returns the greater of two int values as if by calling <code>Math.max</code> . E.g <code>Integer.max(34, 55)</code>
<code>min()</code>	Returns the smaller of two int values as if by calling <code>Math.min</code> . E.g <code>Integer.min(34, 55)</code>
<code>parseInt()</code>	Parses the string argument as a signed decimal integer. E.g. <code>parseInt(" 23 ")</code>

An example follows that demonstrates how to *extract* an integer value embedded in a string.

**Example 1**

Suppose input values to a program are provided in a *CSV* style. CSV stands for comma-separated-values and is a format that has been used in computing systems for moving data from one system to another system. Suppose the data is available in the following manner: each line has an item name, a comma, and a quantity, with no embedded spaces. For example:

```
item,quantity  
monitor,45  
laptop,55  
tablet,50  
desktop,40
```

Using a `Scanner` object a program could use the `Scanner` method `nextLine()` to get a line having three things: an item name, a comma, and an integer. The program can find the location of the comma and know that what follows in the string is a quantity. The quantity can be converted to an integer using the `parseInt()` method. The program below is designed to obtain 4 lines of such information from a user.

Listing 4.9: Using `parseInt()` to get a decimal value.

```
1 import java.util.Scanner;
2 /**
3  * Four lines are read where each line contains
4  * an item name, a comma, and a quantity
5  * with no embedded spaces.
6  */
7 public class TotalQuantity
8 {
9     public static void main(String[] args)
10    {
11        Scanner kb = new Scanner(System.in);
12        int totalQty = 0;
13        for (int i = 0; i < 4; i++){
14            System.out.print("Enter next line: ");
15            String line = kb.nextLine();
16            int commaAt = line.indexOf(",");
17            String qtyAsString =
18                line.substring(commaAt+1);
19            int qty = Integer.parseInt(qtyAsString);
20            totalQty += qty;
21        }
22        System.out.println("total = "+totalQty);
23    }
```

When the program is run with the 4 lines mentioned above we have the output:

```
Options
Enter next line: monitor,45
Enter next line: laptop,55
Enter next line: desktop,22
Enter next line: tablet,55
total = 177
```

**Exercises**

18. Modify Example 1 to find the item for which the quantity on hand is the largest.
19. Write a program that accepts one line that holds an unknown number of integers in a CSV format. The program must print each value on a separate line and then display the largest and smallest of the values.

## Chapter 5

# ArrayLists

There are several techniques for handling collections of data. In this chapter we introduce the `ArrayList`. An `ArrayList` can be visualized as a linear list of objects at index positions 0, 1, ...

The `ArrayList` is a data structure that grows and shrinks gracefully as objects are added and removed. This is a distinct contrast to the array structure covered in the next chapter (with an array once you have defined its size the size cannot be changed).

An `ArrayList` holds a collection of objects whereas arrays can be collections of either a primitive data type or objects. If you wanted to use an `ArrayList` to hold data of a primitive data type you would need to use a *wrapper* class (e.g. `Integer`, `Double`, `Boolean`, `Character`) where wrapper objects contain data of a primitive data type. At this point in your study of Java you have at least used strings that are instances of `String` (objects of type `String`), and so our examples will deal with `ArrayLists` of type `String`.

We illustrate `ArrayLists` using these examples:

1. Basic operations on an `ArrayList`
2. Preventing duplicate entries in an `ArrayList`
3. Creating an `ArrayList` from an array
4. A non-typesafe `ArrayList`.

Below we list some important methods that are defined for `ArrayList`.

Method	Description plus examples using: <code>ArrayList&lt;String&gt; people = new ArrayList()</code>
<code>add(...)</code>	Can be used to either a) append a given element to the end of a list, or, b) if a position is specified insert the given element at the specified position (following elements are shifted <i>down</i> ). <code>people.add("Jaime");</code> <code>people.add(4, "Jaime") ;</code>
<code>clear()</code>	Removes all elements from a list. <code>people.clear();</code>
<code>contains(...)</code>	Returns true if this list contains the specified element. <code>boolean found = people.contains("Jaime");</code>
<code>get(...)</code>	Returns the element at a specified position in this list. <code>String person = people.get(4);</code>
<code>indexOf(...)</code>	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. <code>int pos = people.indexOf("Jaime");</code>
<code>isEmpty()</code>	Returns true if the list has no elements. <code>boolean empty = people.isEmpty();</code>
<code>remove(...)</code>	Can be used to remove either a) the element at a specified position in this list, or b) the first element matching a given object; returns the deleted element and shifts other elements <i>up</i> . <code>String removed = people.remove(4);</code> <code>String removed = people.remove("Jaime");</code>
<code>set(...)</code>	Replaces an element with another element; returns the previous element. <code>String previous = people.set(4, "Jaime");</code>
<code>size()</code>	Returns the number of elements in this list. <code>int numElts = people.size();</code>



## The enhanced for

There is a variation on the `for` called the *enhanced for* that can be used when a program iterates from the first element through to the last element of an `ArrayList` and does not change any values. Using `collection` to represent the `ArrayList` and `type` to represent the type of elements in the collection, the syntax is

```
for ( type variable : collection )
    statement
```

A `for` statement to iterate through an `ArrayList` names of Strings is:

```
for (String s : names)
    System.out.print(s+"");
```

## Example 1

It is considered a good programming practice to specify the data type for the elements of an `ArrayList`. The way to express this is to declare the type inside a pair of angle brackets: `<>`. By specifying this a program cannot accidentally add a different type of object to the `ArrayList`. We say this makes the `ArrayList` *typesafe*. In this example we use the basic `add()` and `remove()` methods to add 4 elements and remove 1 element, and then we use an enhanced `for` to display the elements one by one.

Listing 5.1: Basic operations on an `ArrayList`.

```
1 import java.util.ArrayList;
2 /**
3  * Create an ArrayList from an array of strings
4  */
5 public class BasicOperationsOnArrayList
6 {
7     public static void main(String[] args){
8         // new, empty ArrayList of people
9         ArrayList<String> people=new ArrayList();
10        // add some names
11        people.add("Joe");
12        people.add("Jasper");
13        people.add("Dick");
14        people.add("Abigail");
```

```
15         // remove a name
16         people.remove("Dick");
17         // display the names in people
18         for (String p: people)
19             System.out.print(p+ " ");
20     }
21 }
```

The output, of course, does not include "Dick":

```
Options
Joe  Jasper  Abigail
```

## Example 2

The `contains()` method returns `true` when a given element exists in an `ArrayList`. The program below uses `contains()` and prevents duplicate elements. The user is prompted for names to add to the list - the process stops when the user enters the word `stop`.

Listing 5.2: Prevent duplicate elements.

```

1 import java.util.Scanner;
2 import java.util.ArrayList;
3 /**
4  * Prevent duplicate elements in ArrayList
5  */
6 public class PreventDuplicatesInArrayList
7 {
8     public static void main(String[] args){
9         ArrayList<String> people=new ArrayList();
10        // add some names
11        Scanner kb = new Scanner(System.in);
12        System.out.println("enter names followed"
13            +" by the word stop: ");
14        String name = kb.next();
15        while (!name.equals("stop")) {
16            if (!people.contains(name))
17                people.add(name);
18            name = kb.next();
19        }
20        // display the names in people
21        for (String p: people)
22            System.out.print(p+" ");
23    }
24 }

```

The following shows the prompt to the user, the user's response:

```

Joe Joe Jasper Abigail Abigail Jasper stop

```

and the output generated:

```
Options
-----
enter names followed by the word stop:
Joe Joe Jasper Abigail Abigail Jasper stop
Joe Jasper Abigail
```

### Example 3

This example is included to show how `ArrayLists` are specified in some legacy code. When `ArrayLists` were added to the Java language they were not typesafe - the declaration of an `ArrayList` did not include a type specification, for example:

```
ArrayList name = new ArrayList();
```

This declaration has no angle brackets and so no type specification, and so its possible to add any kind of object to the `ArrayList`. Currently, the recommended practice is always to include a type in the declaration so the program is more robust - certain errors at runtime cannot occur. This example is included only for demonstration purposes and is not recommended practice. The output follows the code listing.

Listing 5.3: Do not declare an `ArrayList` this way.

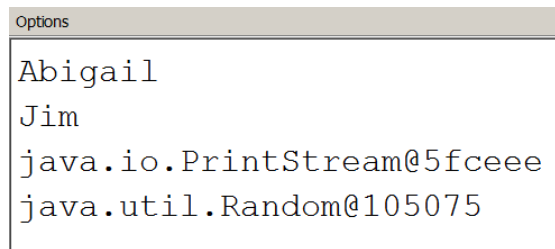
```
1 import java.util.ArrayList;
2 import java.util.Scanner;
3 import java.util.Random;
4 /**
5  * Declaring an ArrayList that is not typesafe.
6  * Not a recommended practice, but something
7  * you might see in legacy code.
8  */
9 public class OldStyleArrayList
10 {
11     public static void main(String[] args){
12         // No type specification for people
13         // Hence, any old object will do
14         ArrayList people=new ArrayList();
15         // Add some elements
16         people.add("Abigail");
17         people.add("Jim");
18         // these two adds are for demonstration
19         people.add(System.out);
```

```

20     people.add(new Random());
21     // display the people
22     for (Object p: people)
23         System.out.println(p);
24 }
25 }

```

The output below shows two `String` objects, a `System.out` object, and a `Scanner` object - all of which were added to the `ArrayList`.



```

Options
Abigail
Jim
java.io.PrintStream@5fcee
java.util.Random@105075

```

## Exercises

1. There is a class called `Collections` which can be used in a program if one includes the import statement:
 

```
import java.util.Collections;
```

 The `ArrayList` is part of the Java Collections framework and there is a method `sort(...)` in `Collections` that can be used to sort an `ArrayList`. For instance, to sort the `ArrayList` named `people` you use the statement:
 

```
Collections.sort(people);
```

 Modify Example 2 so that the list of names appears in alphabetical order.
2. Write a program to analyze text. Each word (token) found is stored in an `ArrayList`. Your program must read the file `Readme.txt`. Display the list of words.
3. Modify the previous program so that duplicate words are not stored in the `ArrayList`.
4. Determine the punctuation used in `Readme.txt`. Remove all punctuation from the tokens and store the words in lowercase in the `ArrayList`.

5. Modify Example 3 to make the `ArrayList` `people` typesafe. What happens now when you compile the program?

## Chapter 6

# One-Dimensional Arrays

There are many situations where we deal with a collection of information. Some examples are:

1. names of students in a class
2. courses offered by a department
3. temperatures for the last month
4. employees in a company

The above cases all have one thing in common: in each case there can be more than one value. For instance, there would be several students in a class and for each student there is a name, for example: "John", "Mary", "Lee", etc. In Java, one way of handling a collection like this is to use a *data structure* called an *array*. The array is declared similar to other variables and then an integer (called an index) is used to refer to its elements individually. So, `studentName` can be the name of the collection and `studentName[0]`, `studentName[1]`, `studentName[2]`, etc. is the way we refer to elements of the collection. As we will eventually see there are other ways of handling these sorts of things - arrays are just one technique a programmer can draw upon. To declare an array of names where each element of the array can be a `String` value we use:

```
String[] studentName;
```

The square braces `[]` are used to indicate a one-dimensional array. Its called one-dimensional because one index value is used to refer to an individual element of the array. In Java index values begin at 0 and go up to the length of the array -1. We can declare arrays of any type, for example:

declaration	sample purpose
<code>String[] studentName;</code>	an array of names
<code>int[] mark;</code>	an array of marks
<code>double[] temperature;</code>	an array of temperatures
<code>boolean[] answer;</code>	an array of true/false answers
<code>char[] letter;</code>	an array of multiple choice answers

The above are examples of how to declare an array. Before the array can be used the programmer must also declare its size. Once the programmer declares the size it cannot be made larger - this is one of the drawbacks to using arrays and why sometimes another technique will be chosen. To declare an array that can hold, say, 100 names we use:

```
String[] studentName;  
studentName = new String[100];
```

or, we can combine the above into one line:

```
String[] studentName = new String[100];
```

or, if an `int` variable holds the length we can write:

```
int arraylength = 100;  
String[] studentName = new String[arraylength];
```

Every array has an `int` field named `length` that is a part of it; the value stored is the length of the array. So, for `studentName` above the value stored in `studentName.length` is 100. This field is very useful; for instance if we need to display all the names in `studentName` we can use the code:

```
for (int i=0; i<studentName.length; i++)  
    System.out.println(studentName[i]);
```

The `length` field is *immutable* which means it cannot be altered once it is set. This means that once you have declared an array to be a certain length you cannot change its length.



## 6.1 Initializing arrays

Because arrays can have multiple values there is a different syntax used when its necessary to set initial values. For instance, suppose we need an array to hold the number of days in each month. We can declare and initialize as:

```
int[] daysInMonth =
    {31,28,31,30,31,30,31,31,30,31,30,31};
```

The Java syntax for initializing an array is to enclose a comma-separated list of values between the pair { }. Initializing arrays this way also sets the length of the array. The value of `daysInMonth.length` is 12.

### Example 1

Consider the following program where `daysInMonth` is initialized and displayed.

Listing 6.1: Initializing and displaying an array.

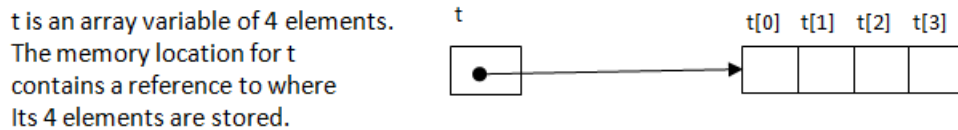
```
1 /**
2  * Display number of days in each month
3  */
4 public class MonthLengths
5 {
6     public static void main(String[] args){
7         int[] daysInMonth =
8             {31,28,31,30,31,30,31,31,30,31,30,31};
9         System.out.println("Days for each of "
10            +daysInMonth.length+" months ");
11         for (int i = 0; i< daysInMonth.length; i++)
12             System.out.print(daysInMonth[i]+" ");
13     }
14 }
```

The output:

Options
Days for each of 12 months 31 28 31 30 31 30 31 31 30 31 30 31

## 6.2 Storage of arrays and copying arrays

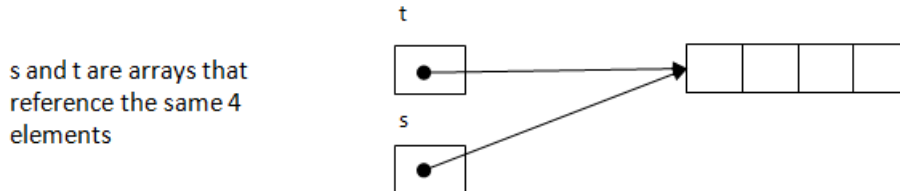
Arrays are objects in Java and so the memory location for the array variable contains a reference to the actual storage locations holding the array's values. For instance the memory allocations for an array can be visualized as:



Now suppose we need to make a copy of the array. If we just use:

```
s = t; //s and t are arrays of same type
```

what we end up with is two storage locations for `s` and `t` that reference the same 4 elements. We haven't created a copy, rather we have two array variables that reference the same 4 elements:



If we need a real copy of the array `t` then we require a loop to accomplish this:

```
// s and t are of the same type
for (int i=0; i<t.length; i++) s[i] = t[i];
```

You can re-instantiate an array variable. New locations are assigned to the array (see below) and the old ones are reclaimed for reuse according to an internal Java garbage collection procedure.

**public interface Comparable < T >**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`).

**All Known Implementing Classes:** ... `String`, ...

**Method Detail**

`int compareTo (T o)`      Compares this object with the specified object, `o`, for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

### 6.3 The enhanced for

There is a variation on the `for` called the *enhanced for* that can be used when a program iterates from the first element through to the last element of an array and does not change any values. The syntax is

```
for ( type variable : array )
    statement
```

The `for` statement in the previous example can be rewritten:

```
for (int days : daysInMonth)
    System.out.print(days+" ");
```

#### Example 2

Consider the following program where `temperature` is assigned values obtained from a user and then the average temperature is displayed. The assignments must be done using a `for` whereas the calculation of the sum can be done with a *enhanced for*.

Listing 6.2: Initializing an array from input.

```
1 import java.util.Scanner;
2 /**
3  * Display average of 7 values
4  */
5 public class AverageTemperature
6 {
7     public static void main(String[] args){
8         Scanner kb = new Scanner(System.in);
9         double[] temperature = new double[7];
10        System.out.println("Enter 7 temperatures:");
11        for (int i=0; i<7; i++)
12            temperature[i] = kb.nextDouble();
13        double sum = 0.0;
14        for (double t:temperature) sum +=t;
15        System.out.println("average= "+sum/7.0);
16    }
17 }
```

**When to use the enhanced for**

The *enhanced for* helps to express a programming idiom succinctly as no loop counter is required. However, there are many cases where the *enhanced for* cannot be used:

1. iterate backwards through the array elements
2. access elements of same-numbered elements of more than one array
3. partially filled arrays (discussed later)
4. assigning new values to array elements.

## 6.4 Passing string values into main()

In all of our main methods we have specified a `String` array named `args`:

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

In the above line `args` is declared to be an array of `String`. The variable `args` is used to pass values (that are strings) into a method. When you have used BlueJ to execute the `main()` method you have the opportunity to pass an array of strings to the program.

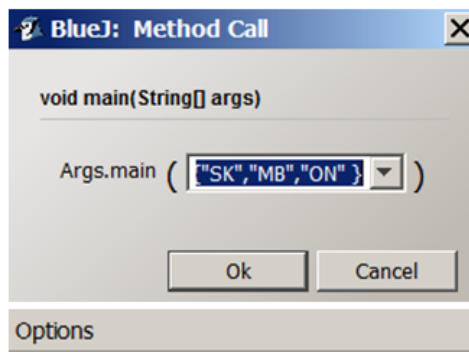
### Example 3

The following program just lists the strings passed into the program.

Listing 6.3: String values passed into `main()`.

```
1  /**
2   * Print the values passed into the program
3   */
4  public class Args
5  {
6      public static void main(String[] args){
7          System.out.println("The elements of args:");
8          for (String s: args) System.out.print(" "+s);
9      }
10 }
```

The following shows a user executing `main()` and passing in 3 strings with the resulting output from the program:



```
The elements of args:
SK MB ON
```

## 6.5 Parallel arrays

There are times when two or more arrays have exactly the same number of elements and where array elements at the same index relate to one another in some meaningful way. For example suppose we have one array of student names and another of student numbers. If the arrays represent information for the same set of students then we would want to arrange that the  $i^{\text{th}}$  element of the name array and the  $i^{\text{th}}$  element of the number array are for the same student, say the  $i^{\text{th}}$  student.

### Example 4

Consider the following example where two arrays hold information for 5 students: one array of names and the other an array of student numbers. For simplicity we initialize the arrays inline. The program prompts the user for a student number and displays the student's name. In order to get the name of the student the program goes through all the elements of `number` and when it finds a number matching the input, it displays the corresponding name in the other array.

Listing 6.4: Finding information in parallel arrays.

```
1 import java.util.Scanner;
2 /**
3  * Student information is in two arrays.
4  * Find student number and report name.
5  */
6 public class StudentInfo
7 {
8     public static void main(String[] args){
9         String[] name =
10             {"Joe", "Linda", "Mary", "Peter", "Lee"};
11         int[] number = {123, 222, 345, 567, 890};
12         Scanner kb = new Scanner(System.in);
13         System.out.println("Enter student number:
14         ");
15         int toFind = kb.nextInt();
16         for (int i=0; i<number.length; i++)
17             if (toFind==number[i])
18                 System.out.println(name[i]);
19     }
```

18 }

This program performs what is usually called a *search* operation: scanning an array looking for a specific element. The program as it was written always iterates through the whole `number` array; normally a programmer would stop the iteration once the element has been found - that is left as an exercise.



## 6.6 Partially filled arrays

In our examples so far the arrays are completely full - every element has a value. In general we do not expect this to always be the case and so, for some applications, we keep track of how many locations are actually filled.

### Example 5

Suppose we need to calculate the average monthly sales. Since there are 12 months we use an array of length 12. We want a user to use the program at any time of year and so there may be fewer than 12 values. The program prompts the user and requests the last value entered to be -1 (a stopper value). The program keeps track of how many elements are filled. Consider the following program and the points discussed after the listing:

Listing 6.5: Average sales for up to 12 months.

```
1 import java.util.Scanner;
2 /**
3  * From monthly sales calculate monthly average.
4  */
5 public class MonthlySales
6 {
7     public static void main(String[] args){
8         double[] sales = new double[12];
9         Scanner kb = new Scanner(System.in);
10        System.out.println("Enter monthly sales"
11            +" enter -1 after last value");
12        int numberMonths=0;
13        double aSale = kb.nextDouble(); //1st month
14        while(aSale != -1) {
15            sales[numberMonths++] = aSale;
16            aSale = kb.nextDouble();
17        }
18        double sum = 0;
19        for (int i=0; i<numberMonths; i++)
20            sum+=sales[i];
21        if (numberMonths>0) System.out.println(
22            "average = "+sum/numberMonths);
23    }
24 }
```

The program exhibits some important features:

1. The `sales` array is of length 12 and the variable `numberMonths` keeps track of how many months of data the user provides.
2. Prior to the `while`, in line 13, the first sales amount is obtained
3. the `while` tests the value of the last sales amount obtained.
4. In the body of the `while` the previously obtained sales amount is placed into the array, and the next value is obtained.
5. Lines 19 and 20 accumulate the total sales
6. Testing for no months of data in line 21 prevents the program from crashing if the user entered -1 as the first value (division by zero).

## Arrays and ArrayLists

In some cases you may want to use the functionality of the `ArrayList` class but for whatever reason the data you are working with is in an array. It is easy to create an `ArrayList` from an array as shown in the program below.

Listing 6.6: Initializing an `ArrayList` from an array.

```
1 import java.util.ArrayList;
2 /**
3  * Create an ArrayList from an array of strings
4  */
5 public class ArrayListFromArray
6 {
7     public static void main(String[] args){
8         // An array that will be used to provide
9         //   initial data for an ArrayList
10        String[] name={"Joe","Jasper","Abigail"};
11        ArrayList<String> nameAL =
12            new ArrayList(name.length);
13        // The add() method is used to append
14        //   an element to the ArrayList
15        for (String n: name) nameAL.add(n);
16        // Printing an ArrayList results in each
17        //   of its elements being displayed as
18        //   in a comma-separated list.
19        System.out.println(nameAL);
20    }
21 }
```

Line 15 is an enhanced `for` where each element of the array is added to the `ArrayList`. Line 19 prints the `ArrayList`. Note the output below and how the `ArrayList` is displayed as a comma-separated-values list embedded in square brackets [ ] - this is the default display for an `ArrayList`.

Options

```
[Joe, Jasper, Abigail]
```

## 6.7 Array utilities in Java class libraries

Arrays are often used in programming and there are many important algorithms that are used. For instance, copying an array was discussed previously. The `System` class contains a method `arraycopy()` that can be used to copy a portion of one array to another. The method takes 5 arguments (in this order): name of the source array, starting element position in the source, the destination array, the starting element position in the destination, and the total number of elements to copy. For instance to copy all elements of the array `t` to the array `s` we could use:

```
System.arraycopy(t, 0, s, 0, t.length);
```

There is a Java library class named `java.util.Arrays` that has additional methods which include:

1. `equals()`: Returns true if two arrays are equal to one another. The arrays are equal if they have the same number of elements and if corresponding elements are equal.
2. `sort()`: Rearranges the elements of an array so they are in ascending sequence.
3. `binarySearch()`: Returns the index of an element if it was found in a sorted array. Binary search is a type of search technique that takes advantage of the fact that an array is sorted. The general idea is to continually bisect the array looking for the required element. The process examines the middle element and determines if the required element is above or below the middle element; then the process continues on that subset of the array where the required element may be present. The process continues until the required value is found or there is nothing left to examine.
4. `fill()`: Assigns a specified value to every element of an array.

### Example 6

The interested student is referred to the Java Class Library documentation for complete information regarding Arrays. Here, we demonstrate how one can sort an array and then search the array for a specific entry. Consider that we have an array of names. To simplify we shall initialize the array in the code. The program prompts the user for a name, performs a search, and then responds accordingly. Following the listing there are some remarks.

Listing 6.7: Initializing and displaying an array.

```
1 import java.util.Arrays;
2 import java.util.Scanner;
3 /**
4  * An array of names is sorted and then
5  * searched for a specific name.
6  */
7 public class SortAndSearch
8 {
9     public static void main(String[] args){
10         String[] name =
11             {"Joe", "Linda", "Mary", "Peter", "Lee", "Patricia"};
12         Arrays.sort(name);
13         Scanner kb = new Scanner(System.in);
14         System.out.println("Enter a name: ");
15         String toFind = kb.next();
16         int foundAt =
17             Arrays.binarySearch(name, toFind);
18         if (foundAt >= 0)
19             System.out.println(
20                 "Found in position "+foundAt);
21         else System.out.println("Not Found ");
22     }
23 }
```

Note the following points regarding SortAndSearch above:

1. The Arrays class is imported in line 1.
2. The `sort()` method is invoked in line 12. As a result the entries of name have been rearranged and are now sorted alphabetically.
3. In line 17 `binarySearch()` is used to search for the name entered by

the user. If the value is not negative then that is the index where the name was found.

## Exercises

1. Modify Example 1 to include a parallel array for the names of months. On 12 lines, one per month, display each month and its number of days.
2. Modify Example 2 to determine the minimum and the maximum of the 7 temperatures. Note that this is similar to Exercise 1 in the Section on the `for` statement, but in this case the elements are stored in an array.
3. Modify Example 3 so that it sorts the strings before they are displayed.
4. Modify lines 14-16 in Example 4 so that the loop stops if the number is found.
5. Modify Example 5 so that it displays the name of the month when sales were their largest.
6. Write a program to determine someone's score on a multiple-choice test having 12 questions. The program has two char arrays: `correctAnswers[]` and `studentAnswers[]`. The array `correctAnswers` holds the correct answers to the test. Use the following for correct answers:  
a b c d a b c d a b c d  
  
The student's answers will be provided by the user of the program. These must be stored in the array `studentAnswers[]`. After the student answers have been obtained the program must determine the student's score: the number of questions the student answered correctly.  
  
For example if the student answers are:  
a a a b b b c c c d d d  
then the score for this student is 4.
7. Write a program to analyze text such that each word (token) found is stored in an array. Use the file `Readme.txt`. Sort the array and display its contents.





## Chapter 7

# Introduction to Methods

Every Java application has at least one class that has a method named `main`. If you execute a Java class the JVM automatically looks for a method named `main` and passes control to its first statement.

In practice most classes comprise many methods. By using methods you can write modular code where an algorithm is broken down into its component parts. For an example of the modular approach we discuss the implementation of the Sieve of Eratosthenes which is an algorithm for finding prime numbers. This algorithm is attributed to a Greek mathematician, Eratosthenes of Cyrene, who lived more than 2,000 years ago.

Using a sieve of size  $n$  we can find all the prime numbers between 1 and  $n$ . Recall that a prime number is an integer greater than 1 that has no positive divisors other than 1 and itself. For instance, 7 is a prime number since the only integers that evenly divide 7 are 1 and 7 itself; the number 7 can only be expressed as the product of two numbers:  $1 \times 7$ . Any integer that is not prime can be expressed as the product of two integers other than 1 and itself. For example, 8 is not a prime number since it can be expressed as a product of 2 and 4. The integer 8 is divisible by 1, 2, 4, and 8; we say the integer 8 is a *composite* integer.

Next we discuss the basic algorithm and then its implementation in Java. There are many references to the Sieve of Eratosthenes on the web . At the time of writing an article could be found in Wikipedia that illustrates a sieve for finding all primes less than 120.

## 7.1 Example - Sieve of Eratosthenes

To use the sieve we first construct a list of positive integers ranging up to a *limit*. The algorithm is iterative; it begins by crossing off all multiples of 2. Then the list is scanned for the next integer following 2 that is not crossed off (this will be 3) and its multiples are crossed off. This process of finding the next non crossed off integer and crossing off its multiples continues until no more can be found. In the implementation we represent the list by an array list of integers where the  $i^{\text{th}}$  entry is simply  $i$ , and we cross off an integer by setting it to 0.

### Sieve of Eratosthenes:

1.  $\text{next} \leftarrow 2$  // begin crossing-off multiples starting with 2
2. for  $\text{next}$  ranging from 2 to  $\text{limit}$  in increments of 1
  - (a) if  $\text{list}_{\text{next}}$  is not crossed-off // cross-off all multiples of next
    - i.  $p \leftarrow 2 * \text{next}$
    - ii. while  $p \leq \text{limit}$ 
      - A.  $\text{list}_p \leftarrow 0$  // cross-off this multiple
      - B.  $p \leftarrow p + \text{next}$  // next multiple

Lets look at applying the algorithm for  $\text{limit}=25$

- The algorithm requires the list:  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25  
Note the list starts with 0 1 in order that  $\text{list}_i = i$ .
- When step 2 executes for the first time ( $\text{next}$  is 2) the list becomes  
0 1 2 3 0 5 0 7 0 9 0 11 0 13 0 15 0 17 0 19 0 21 0 23 0 25  
where all the even numbers larger than 2 are replaced by 0.
- When step 2 executes again ( $\text{next}$  is 3) the list becomes  
0 1 2 3 0 5 0 7 0 0 0 11 0 13 0 0 0 17 0 19 0 0 0 23 0 25  
where all the numbers that are multiples of 3 are replaced by 0.
- This process continues and the list eventually becomes:  
0 1 2 3 0 5 0 7 0 0 0 11 0 13 0 0 0 17 0 19 0 0 0 23 0 0
- From the above, starting at 2, we easily see primes less than or equal to 25 are: 2 3 5 7 11 13 17 19 23

In Listing 7.1 you can see our first implementation of the algorithm. Following the listing we discuss the implementation in more detail.

Listing 7.1: Sieve of Eratosthenes

```
import javax.swing.JOptionPane;
import java.util.ArrayList;
public class Sieve1 {
    public static void main(String[] args){
        // get limit from user
        String limitAsString =
            JOptionPane.showInputDialog("Enter upper
            limit:");
        int limit = Integer.parseInt(limitAsString);
        // set up the list of integers
        ArrayList<Integer> list = new ArrayList<>();
        for (int i=0; i<=limit; i++){
            list.add(i);
        }
        // apply the sieve technique
        for (int next=2; next<limit; next++){
            if (list.get(next)!=0){
                // cross off multiple of next
                int p = 2*next;
                while (p <= limit){
                    // cross off this element
                    list.set(p, 0);
                    p+=next;
                }
            }
        }
        // display primes
        ArrayList<Integer> result = new
            ArrayList<>();
        for (int i=2; i<limit; i++){
            if (list.get(i)!=0) result.add(i);
        }
        JOptionPane.showMessageDialog(null,"primes
        < "+limit+" are "+result);
    }
}
```

In Listing 7.1 you should note:

1. As always we have coded the algorithm using a single method named `main`.
2. We have used `JOptionPane` to obtain the limit for the sieve from the user.
3. The `ArrayList` named `list` is initialized so that its elements start with 0 1. The code initializes `list` so the  $i^{\text{th}}$  position is  $i$ :

```
for (int i=0; i<=limit; i++){  
    list.add(i);  
}
```

4. An entry in the list is crossed off (set to zero) using the statement:  
`list.set(p, 0);`

## 7.2 Using Methods

A programmer will often use many methods when coding an algorithm. This is done to improve readability and to provide reusable pieces of code. We will, over the next two examples, re-write the Sieve to better indicate the overall structure of the program:

- get limit from user
- apply the sieve
- display the results

A book we recommend to Java programmers is *Clean Code: A Handbook of Agile Software Craftsmanship* [7] by Robert C Martin. For an example using methods to create a self-documenting and readable program the book uses the Sieve of Eratosthenes. In that example Robert Martin codes the algorithm using more methods than we show here in this chapter.

Section 7.2.1 uses a *value-returning* method and Section 7.2.2 introduces a non value-returning method (i.e. a `void` method) and the use of *parameters*.

### 7.2.1 Value-returning methods

A *value-returning* method is one that returns a value to the point where the method was invoked. Let us begin by placing the code for getting the limit from the user in a method. For this we will use a value-returning method that begins:

```
public static int getLimitFromUser()
```

There are two things you should note about the above method header:

1. Instead of `void` this method specifies `int`. In this way we are stating the method will return an `int` value.
2. The method is named `getLimitFromUser`.

To illustrate the use of this value-returning method we have re-coded the Sieve as shown in Listing 7.2. In this listing you should note:

1. There are two methods, one named `main` and the other named `getLimitFromUser`.
2. The method `getLimitFromUser` contains the code to prompt the user for the `limit`.
3. The last statement in `getLimitFromUser` is a `return` statement this `return` is executed when `getLimitFromUser` ends causing a value to be returned to the point where the method was invoked.
4. The first statement in `main` is  

```
int limit = getLimitFromUser();
```

This statement calls `getLimitFromUser` which executes and when it completes it returns an `int` value which is assigned to `limit`.

Listing 7.2: Sieve of Eratosthenes with value-returning method `getLimitFromUser`

```
import javax.swing.JOptionPane;
import java.util.ArrayList;

public class Sieve2
{
    public static void main(String[] args){
        int limit = getLimitFromUser();
        // set up the list of integers
        ArrayList<Integer> list = new ArrayList<>();
        for (int i=0; i<=limit; i++){
            list.add(i);
        }
        // apply the sieve technique
        for (int next=2; next<limit; next++){
            if (list.get(next)!=0){
                // cross off multiple of next
                int p = 2*next;
                while (p <= limit){
                    // cross off this element
                    list.set(p, 0);
                    p+=next;
                }
            }
        }
        // display primes
        ArrayList<Integer> result = new
            ArrayList<>();
        for (int i=2; i<limit; i++){
            if (list.get(i)!=0) result.add(i);
        }
        JOptionPane.showMessageDialog(null, "primes
            < "+limit+" are "+result);
    }
    //
    // the method getLimitFromUser follows
    // on next page
}
```

```
public static int getLimitFromUser(){
    String limitAsString =
        JOptionPane.showInputDialog("Enter upper
        limit:");
    int number =
        Integer.parseInt(limitAsString);
    return number;
}
}
```

We say that Java has two types of methods:

1. `void` methods that do not return a value.
2. Value-returning methods have their type declared in the method header.
3. A `return` statement must specify a value of the type defined in the header.



### 7.2.2 Parameters

A method can be defined with or without parameters. We have seen, but never made use of, the parameter specified for a main method. By convention the parameter for a main method is an array of `Strings`; arrays are covered elsewhere in these notes.

In our next version of the Sieve we have coded two more methods that begin with the headers:

```
public static ArrayList<Integer> applySieve( int
    upperlimit)

public static void displayResults(
    ArrayList<Integer> listOfPrimes)
```

The above headers indicate:

1. The method `applySieve` has one parameter named `upperlimit` of type `int`. When `applySieve` is called we say the calling method must pass in an `int` value.
2. The method `applySieve` declares its type to be `ArrayList<Integer>` and so it must return a value which is an `ArrayList` of `Integers`.
3. The method `displayResults` has one parameter named `listOfPrimes` which is an `ArrayList` of `Integers`. When `displayResults` is called we say the calling method must pass in an `ArrayList` of `Integers`.
4. The method header for `displayResults` uses the keyword `void` and so `displayResults` must not return a value.

In Listing 7.3 you will see a class that has 4 methods. The `main` method is very simple:

```
public static void main(String[] args){
    int limit = getLimitFromUser();
    ArrayList<Integer> result = applySieve(limit);
    displayResults(result);
}
```

One can say it clearly states what it does: get a limit from a user, apply the sieve to get a result and then display that result. Note each statement in `main` invokes another method. In Listing 3 you should observe:

- `getLimitFromUser` is the same as in Listing 7.2. This method does one simple thing: it interacts with the user to get a limit for the Sieve of Eratosthenes.
- `displayResults` is a void method, and so it does not return any value.
- `displayResults` has a parameter that is used by a calling method to pass an `ArrayList` into the method. The method uses the name `listOfPrimes` to refer to that `ArrayList`. `displayResults` does one simple thing: it displays the list of primes.
- `applySieve` is value-returning. The header specifies the type that it returns to a calling method. The last statement in the method returns the `ArrayList` containing the list of primes.
- `applySieve` has a parameter that is used by a calling method to pass an `int` into the method. This method uses the name `upperLimit` to refer to that `int` value.

The hope with Listing 7.3 is that, overall, the program is easier to understand. Someone reading this code should easily grasp its structure and order of execution. Beginning at `main`, the reader knows the other methods are called in sequence. The main method makes it clear how the program is organized.

Listing 7.3: Sieve of Eratosthenes using 4 methods including main

```
import javax.swing.JOptionPane;
import java.util.ArrayList;

public class Sieve3
{
    public static void main(String[] args){
        int limit = getLimitFromUser();
        ArrayList<Integer> result =
            applySieve(limit);
        displayResults(result);
    }

    public static void
    displayResults(ArrayList<Integer>
    listOfPrimes){
        JOptionPane.showMessageDialog(null,"list of
        primes "+listOfPrimes);
    }

    public static int getLimitFromUser(){
        String limitAsString =
            JOptionPane.showInputDialog("Enter upper
            limit:");
        int number =
            Integer.parseInt(limitAsString);
        return number;
    }

    public static ArrayList<Integer> applySieve(int
    upperlimit){
        // set up the list of integers
        ArrayList<Integer> list = new ArrayList<>();
        for (int i=0; i<=upperlimit; i++){
            list.add(i);
        }
        // apply the sieve technique
        for (int next=2; next<upperlimit; next++){
```

```
        if (list.get(next)!=0){
            // cross off multiple of next
            int p = 2*next;
            while (p <= upperlimit){
                // cross off this element
                list.set(p, 0);
                p+=next;
            }
        }
    }
    // the list of primes to return
    ArrayList<Integer> result = new
        ArrayList<>();
    for (int i=2; i<upperlimit; i++){
        if (list.get(i)!=0) result.add(i);
    }
    return result;
}
}
```

## Summary

This chapter is an introduction to using methods in Java applications. The chapter on designing classes will complete this topic. Do note that as you have been learning Java you have been introduced to many methods developed by other programmers. Examples:

- the `Math` class has a method `max` which is a value-returning method with two parameters. To use `max` we write code such as `Math.max(i, j)` where `i` and `j` are passed in to its parameters.
- The `Random` class has a value-returning method used to get a random value. To use `nextInt` we write code such as `g.nextInt(6)` where the value `6` is passed in to its parameter.
- The `System` class has a method `println` that has one parameter of type `String`; note that `println` does not return a value.

Methods are used by programmers to produce readable code. Methods can be used to break complex software into its component parts. Well designed components can be read and understood without having to know everything about the software as a whole. Robert Martin's purpose in writing *Clean Code* is to help its readers become better programmers. Each method must be given a name that indicates exactly the purpose of the method. To some programmers this way of naming methods along with good names for variables replaces the need for most comments.

Methods can be categorized as *value-returning* or `void`. A value-returning method declares its type in the header and must have a `return` statement that returns a value of the requisite type.

Methods may be designed with no parameters, or with parameters of specific types. A method with more than one parameter separates their declarations with commas. For instance:

```
public static void methodA (int a, double x, String aName)
```

When the above method is called the calling method must include three values in the order indicated; for example:

```
methodA (25, 26.3, "George");
```

**Exercises**

1. Consider the Sieve examples in this chapter. Write another version that uses methods to break up the complexity of `applySieve`.
2. Consider Example 2 in Section 4.2. In this example a scanner object is used to get a line from the user, then the characters are examined one-by-one, and then the sum of the numeric value of those characters is displayed. Rewrite the program so the `main` method is:

```
public static void main(String[] args){
    String line = getLineFromUser();
    int s = analyzeLine( line );
    displaySum( s );
}
```

3. Consider Example 3 in Section 4.2. In this example the user is prompted for a student number, the student number is analyzed for validity and a message `valid` or `invalid` is displayed. Rewrite the program so the `main` method is:

```
public static void main(String[] args){
    String number = getStudentNumber();
    boolean valid = analyzeNumber( number );
    displayValidity( valid );
}
```

## Chapter 8

# Designing Java Classes

Up to this point our programming has involved the coding of a single class but often we have demonstrated the use of one or more classes defined in the Java class libraries. These pre-defined classes include: `Character`, `Integer`, `JOptionPane`, `Math`, `Random`, `Scanner`, `String`, and `System`. In this chapter we will see how to create programs where we can code several classes ourselves. A class comprises *fields*, *constructors*, and *methods* which make up the subject matter of this chapter. But before we get into those, we discuss a few points regarding our use of these pre-defined classes.

The `Math` class contains various mathematical constants and utility methods for common mathematical operations. For instance, the `Math` class has a field named `PI`. In a program we can have a statement such as

```
double circumference = 2.0 * Math.PI * r;
```

where we reference `PI` using the name of the class, `Math`. In particular we did not create an object of type `Math`. It's a similar case for the method `max()` in the `Math` class. To find the maximum of two numbers we can use the expression `Math.max(num1, num2)`. `PI` is a static field and `max()` is a static method; these types of fields and methods can be accessed without the need for an object.

*Math class  
has a static field*

So to use the facilities of the `Math` class we did not need an object, but there are other times when it was necessary for us to create an object. For example consider the `Scanner` class and the instantiation of an object:

```
Scanner scanner = new Scanner(source);
```

In order to get the next token from `scanner` we use expressions like `scanner.next()`. The object (also called an *instance*) named `scanner` must

*has a static method*

*instance method*

keep track of its source and its current position on that source. When a scanner object executes the `next()` method, the method obtains the next token on the source, advances its position on that source, and returns the token.

*value-returning  
method*

With respect to methods we have seen another difference: sometimes we have used a method that returned a value and other times we invoked a method knowing that something will happen. When we simulate the tossing of a six-sided die we use:

```
int throw = generator.nextInt(6)+1;
```

where we understand `nextInt(6)` returns a value that is used in the arithmetic expression `generator.nextInt(6)+1`. The method `nextInt()` is an example of a *value-returning* method.

When we display something in the user's terminal window we use a statement like:

```
System.out.println("Enter a number: ");
```

*void method*

where we understand that the string "Enter a number: " will be displayed, and where there is no value returned that we make any use of. The method `println()` is an example of a *void* method (one that does not return a value).

In the rest of this chapter we:

- present an example from an educational setting where multiple classes are useful;
- explore the use of fields, methods, and constructors;
- discuss *modifiers* named `public` and `private` used to control access to classes, fields, methods, and constructors;
- discuss the concept of *overloading*;
- describe the implementation of *associations* between classes;
- describe the passing of values to parameters;
- describe variable length parameter lists;
- discuss the use of methods to simplify the logic of a program.

The above list is lengthy and indicates this chapter covers several new Java programming concepts.



## 8.1 Using Multiple Classes

So far we have always used a single class for our examples and exercises. However it is very common for Java-based systems to involve several classes where each class encompasses the requirements (fields, constructors, and methods) of a significant concept. For example, if you were developing a system for your educational institution you would need to implement things having to do with students, subject areas, courses, instructors, etc. What should happen is that you design separate classes for each of these concepts: a class for student, a class for subject, a class for course, and so on:

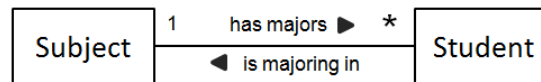
```
public class Student{
...
public class Subject{
...
public class Course{
...
public class Instructor{
...

```

Before writing any code we recommend drawing a *Class Diagram* to illustrate the concepts you are thinking about. To simplify we present a diagram in Figure 8.1 that shows just a subset of the concepts mentioned above: two classes and a relationship. The relationship is based on a business rule for the institution: a student declares a major in one subject, and that for a subject there can be many students majoring in that subject.

*Class Diagram*

Figure 8.1: A simple Class Diagram.



Each class is shown in its own compartment and a relationship is shown as a named line. The small filled arrow shape gives the viewer an indication of how to *read* the diagram. The 1 and the \* are called *multiplicities* that stand for *one* and *many* respectively, and are used to express how many of one object can be related to another object through the relationship. In this example the association line represents a two-part business rule:

- a student *is majoring in* one subject
- a subject *has many majors* who are students

As we continue we will be developing `Student` and `Subject`, and in Section 8.11 you will find complete listings of these two classes.

## 8.2 Fields

We have learned to define variables when we needed to keep track of information. Suppose we were developing a class to represent the concept of *student*. We would need variables for student identifier, first name, last name, gender, active, etc. Formally these variables are referred to as *fields* and previously we would have defined these similar to:

```
public class Student() {
    int id;
    String firstName;
    String lastName;
    char gender;
    boolean active;
```

*private fields*

However, it is generally recommended that one define the above fields including another *modifier* named `private`. By specifying `private` we make it impossible for these fields to be referenced from outside the class. We will say more about this idea later. Now we have:

```
public class Student() {
    private int id;
    private String firstName;
    private String lastName;
    private char gender;
    private boolean active;
```

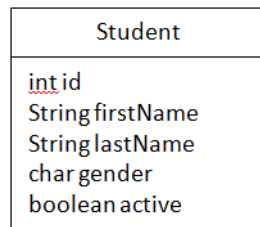
*Class shown with  
2 compartments*

The above is Java code that defines part of a class named `Student`. A less formal way to illustrate the above is to use a class diagram. Now we show two compartments, one for the name of the class, and one for the names of the fields and types, as shown in Figure 8.2.

*Object Diagram*

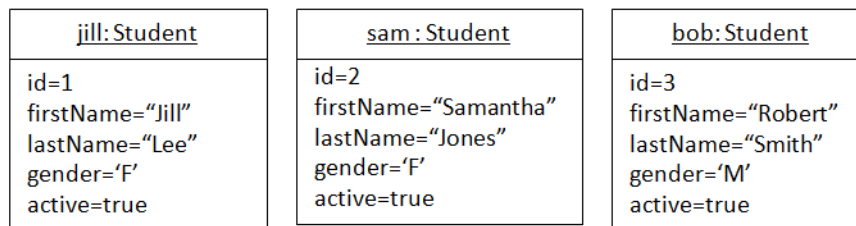
`Student` objects can be created where each object would have its own copy of these fields, and so each student object could have different values stored in its fields. Figure 8.3 is an **Object Diagram** where each object is represented by two compartments. In the upper compartment we show a name for an object (e.g. `jill`) and the class of the object (e.g. `Student`). By convention this information is underlined. In the lower compartment we show values

Figure 8.2: Class diagram with two compartments



for fields. The diagram makes it clear there are three students and each student object has its own fields to hold values for id, first name, etc.

Figure 8.3: Object Diagram with 3 student objects.



### Instance and class fields

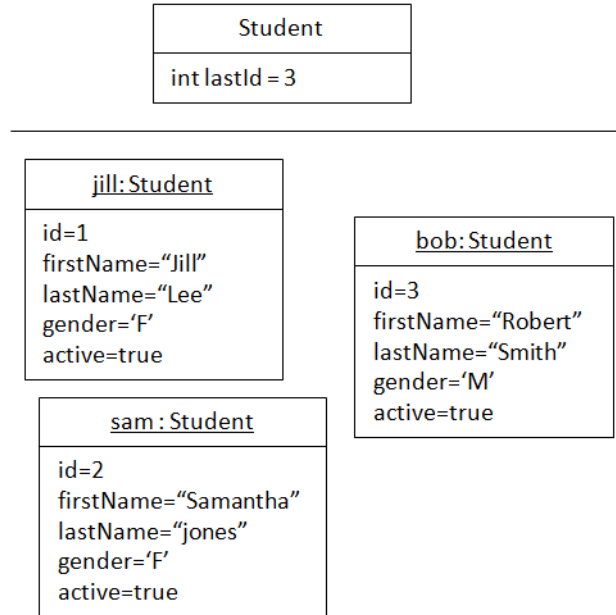
If we define a field in a class and specify it as static that field is referred to as a **class** field. Only one copy of a class field exists regardless of the number of objects that exist. Suppose we are developing a system where we generate student identifiers sequentially. Consider a field named `lastId` where we store the `id` used for the last student object created. Consider the following for our `Student` class:

```
public class Student {
    // class fields
    private static int lastId; // a static field
    // instance fields
    private int id;
    private String firstName;
    private String lastName;
    private char gender;
    private boolean active;
}
```

*static field  
is a class field*

In Figure 8.4 we show the Student class and 3 student objects. Note that we show the `lastId` field in the Student class with its *current* value. Later we will discuss methods and how the value of `lastId` is used and updated.

Figure 8.4: Static field in class and instance fields in objects.



## 8.3 Methods

A method is a named collection of Java statements. Optionally a method can have a parameter list to provide for data to be passed in and can be designed to return a value back to the point where it is invoked. Methods are an important tool when we write Java programs. We use methods for two purposes.

1. To eliminate code redundancy: If we find that we are repeating the same code as elsewhere in our program we write that code as a method and call it from wherever it's needed. Now, if this code ever needs to change there is only place where the change needs to be performed.
2. To create readable programs: Programs are subject to change. Change happens if errors are discovered in existing code, or, if the business rules change. In real systems it has been found that more time and effort is spent maintaining a program that what was required to create it in the first place. Whenever code must be maintained the program must be understood first before changes are made - the more readable your code is, the easier and more reliably the code can be modified.

When we are designing a program and we decide that certain functionality must be placed into a method we determine

1. whether or not the method returns a value
2. if the method is a class method
3. if the method is an instance method
4. whether or not the method can be accessed from other classes.

### Value-returning methods

A method can be designed to return a value of any primitive type, array, or object type. We have seen many examples of these including `max()` and `min()` in the `Math` class, and the `nextInt()` and `nextBoolean()` of the `Random` class.

In our educational example that we are developing we have made the fields of `Student` private. Instead of giving other classes direct access to fields the convention is to provide methods for this purpose. One reason to do things this way is to *hide* the implementation of fields which then makes it

easier to change an implementation later. So, for each field of the Student class we can design a group of methods that are called *getters* (sometimes called *accessors*). For each of these we specify the data type they return and the last statement in a getter method is a `return` statement. Consider the following code in the Student class:

```
public class Student {
    private static int lastId;
    private int id;
    private String firstName;
    private String lastName;
    private char gender;

    public static int getLastId(){
        return lastId;
    }
    public int getId(){
        return id;
    }
    public String getFirstName(){
        return firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public char getGender(){
        return gender;
    }
    public boolean isActive(){
        return active;
    }
}
```

Observe how methods are named above. Because they are getters, each method name (but one) begins with "get" prepended to a field name where the first character of the field name is capitalized. Since `active` is `boolean` its getter begins with *is*. As for variable names, we use "camel case" for method names.

*camel case*

Consider `getLastId()` above. It is defined as static since the `lastId()` field is static. To invoke this method we use:

*calling a static method*

```
int lastOne = Student.getLastId();
```

Calling the other methods above only make sense in the context of some object, and so these methods are examples of *instance* methods. If `joe` is the name we are using to reference a student object then we can write meaningful statements to get, for example, the gender of this student:

*calling an  
instance method*

```
Student joe = new Student();
...
char g = joe.getGender();
```

Two other standard value-returning methods normally provided for a class are the `toString()` and `equals()` methods.

When an object is being displayed via `println()`; it is the `toString()` method that determines what will be displayed. The `toString()` method returns a string that represents an object. As an example, we will use the following `toString()` in our system:

*toString()*

```
public String toString(){
    return id+" "+firstName+" "+lastName;
}
```

If the statement

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

was executed then the output to the terminal window would be:

```
1 Jill Lee
```

When objects are compared for equality a programmer normally uses the `equals()` method. Previously we have used the `equals()` method to compare two strings for equality. The idea that two strings are equal is simple: the two strings are either the same string, or they are of the same length comprising the same characters. In a program to compare two names, say `name1` and `name2`, we can use either:

*equals()*

```
name1.equals(name2)
```

or

```
name2.equals(name1)
```

For more complex objects like student objects it is not always so simple to decide on the equality test. A student object has several fields that could enter into consideration. At this time we will take a simple viewpoint on this and consider two student objects the same if they have the same value for the student identifier. Consider the method:

```
public boolean equals(Student s){
    return id == s.id;
}
```

Note how this method has a parameter `s` of type `Student`. The `equals` method is invoked for a particular student object and that object is being compared to the object `s`. If some code were to check two student objects (say `jill` and `sam`) to see if the objects have the same content the programmer could code either

```
if (jill.equals(sam))
```

or

```
if (sam.equals(jill))
```

If instead one uses `jill == sam` it is memory addresses (and not content) that are being compared for equality. We use `==` only to determine if two object references are to the same object, as in:

```
Student s1, s2;
...
// test to see if s1 and s2 refer to the
// same student object
if (s1 == s2) ...
```



### void methods

Sometimes we need methods just to complete some task and for which there is no value that needs to be returned. One of the conventions in java programming is to provide *setter* methods that are used only to set, or change, the value of a field. Consider this partial listing of `Student`:

```
public class Student {
    // constructors ...
    // field declarations ...
    // getters ...
    // setters:

    // but no setter for the id field
    //     public void setId(int newId){
    //         id = newId;
    //     }

    public void setFirstName(String newFirstName){
        firstName = newFirstName;
    }
    public void setLastName(String newLastName){
        lastName = newLastName;
    }
    public void setGender(char newGender){
        gender = newGender;
    }
    public void setActive(boolean newActive){
        active = newActive;
    }
}
```

In the above we have commented out a setter that could have been included for `id`. A reason for not having a setter for `id` is the idea that once a student is assigned a unique identifier that identifier would never change.

Observe how methods are named above. Because they are setters, each method name begins with "set" prepended to a field name where the first character of the field name is capitalized. As mentioned before, we use "camel case" for method names.

*naming setters*

*camel case*

## 8.4 Constructors

*no-arg*

Constructors are used to instantiate an object. If you do not explicitly code a constructor, then the Java compiler creates a *no-arg* constructor for you. A no-arg constructor is a constructor that takes no arguments - the parameter list is empty. A constructor is invoked any time you use the `new` operator, for example:

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

A constructor is similar to a method in that it is a block of code that may have a parameter list, but there are two major differences:

1. A constructor returns an object but the code for a constructor does not include any `return` statements.
2. Although a constructor returns an object created from a specific class, the constructor does not explicitly declare the type.

Its common for a class to have multiple constructors that will differ from one another in their parameter lists (see overloading, Section 8.6). To illustrate we introduce two constructors for the `Student` class:

1. A no-arg constructor: This constructor has no values passed in to it and so it creates a student object with a generated value for the student identifier but has to make up values for the other fields. Presumably those fields will eventually be filled in with calls to setter methods.
2. A constructor with four parameters. This constructor assigns a generated value to the student identifier and the other fields are set according to the caller's supplied values.

The first part of our `Student` class is now (note that the code for `nextId()` is shown on page 199):

```
1 /**
2  * A student.
3  */
4 public class Student {
5     // class fields
6     private static int lastId;
7     // instance fields
8     private int id;
9     private String firstName;
```

```

10     private String lastName;
11     private char gender;
12     private boolean active;
13     // first constructor, no arguments
14     public Student(){
15         id = nextId();
16         // default values for a student:
17         firstName = "unknown";
18         lastName = "unknown";
19         gender = '?';
20         active = false;
21     }
22     // second constructor, four arguments
23     public Student (String firstName, String
24         lastName, char gender, boolean active){
25         id = nextId();
26         //
27         // when parameters and fields have the same
28         // name they are distinguished this way:
29         // A field name alone refers to the
30         // parameter;
31         // A field name prefixed with "this."
32         // refers to an object's fields.
33         this.firstName = firstName;
34         this.lastName = lastName;
35         this.gender = gender;
36         this.active = active;
37     }
38     // other methods ...

```

The second constructor (lines 23-35) illustrates a style of coding that is quite common for constructors and methods. In the parameter list of the constructor the parameters have the exact same name as the corresponding fields. Formally, in the Java language, we say the field is *shadowed* by the parameter. To reference the field within the method you must use the `this` keyword. When the name is used alone (e.g. `firstName`) in a constructor it is a reference to the parameter. When a name is prefixed with `"this."` it is a reference to the object's field. Consider line 31:

```
this.firstName = firstName;
```

The right hand side of the assignment statement is a reference to a param-

*shadowing*

*this keyword*

eter and the left hand side of the assignment statement is a reference to the the object's field. And so this assignment statement assigns a value passed in via the parameter to an object's field.

We illustrate the class diagram for the Student class with three compartments, where the third shows constructors and methods. Note that we have left out some details such as the field types and parameter types. We have included plus and minus signs to show whether a method or constructor is public('+') or private('-').

Figure 8.5: Student class with 3 compartments.

Name of class →	Student
Fields →	id firstName lastName gender active
Constructors →	+Student () +Student (firstName, lastName, gender, active)
Methods →	-nextId () +getId () +getLastId () +getFirstName () +getLastName () +getGender () +isActive () +getMajor () +setLastId (newLastId) +setFirstName (newFirstName) +setLastName (newLastName) +setGender (newGender) +setActive (newActive) +setMajor (newMajor) +toString () +equals (s)

## Using constructors

A constructor is invoked any time a program executes a `new` operator. Consider the code below where we invoke each of the above constructors. For the no-arg constructor we follow up with setters to fill out the student object.

Listing 8.1: Using constructors.

```
1  /**
2   * Create two student objects
3   * using the two constructors
4   */
5  public class UseConstructors
6  {
7      public static void main(String[] args){
8          // first, with the no-arg constructor
9          Student jill = new Student();
10         // use setters to complete the student
11         // object
12         jill.setFirstName("Jill");
13         jill.setLastName("Lee");
14         jill.setGender('F');
15         jill.setActive(true);
16         // now with the other constructor
17         Student sam = new
18         Student("Samantha", "Jones", 'F', true);
19         // display the students
20         System.out.println(jill);
21         System.out.println(sam);
22     }
```

The output from `UseConstructors` is shown below. We can see from the output that when a student object is displayed the `toString()` method was used.

```
Options
1 Jill Lee
2 Samantha Jones
```

## 8.5 Visibility Specifications: Public, Private

In our introduction to Java we are only concerned with the modifiers `public` and `private` which can be used on classes, fields, methods, and constructors.

### Classes

When complex systems are developed Java classes will be organized into packages. We have used several packages (i.e. the Java class libraries) where each library is a package containing related classes. For example,

*java.lang*

1. the package `java.lang` contains fundamental classes such as `Character`, `Double`, `Integer`, `Math`, and `String`;

*java.util*

2. the package `java.util` contains utility classes such as `Arrays`, `ArrayList`, `Collections`, `Random`, and `Scanner`.

The packages we have been using are all designated as public meaning that any other class can use them. If there is a need to provide different accessibility then other modifiers such as `private` and `protected` can be used. Please see Volume II of these notes for further information.

For introductory programming we will only use `public` for our classes. With BlueJ all of our classes will be in one default `package`. For more information with respect to BlueJ consult the BlueJ documentation.

### Fields

*private*

If a field is designated private the field can only be accessed from within the class where it is defined. This is considered good practice because the class has control over the implementation of the field and exposes the field only through methods. This idea of keeping the implementation aspects hidden is called *Information Hiding*.

*Information Hiding*

As an example, suppose a three-dimensional point (x,y,z) is represented in a program with three floating point variables. Suppose the representation must change to be an array of size three, then that change would only affect this class and no others. The methods that were in place to provide information about x, y, and z can continue to serve (with minor modification) the needs of other classes. So, information hiding is a way of controlling the

scope of changes in a system.

When a field is designated `public` the field can be accessed from any other class. What can happen then is that other classes become dependent on the data type that is used. However, there are situations where `public` is appropriate - consider the utility class `Math`. `Math` has two public fields `PI` and `E` for  $\pi$  and  $e$ .

*public*

## Methods

When a method is designated `public` then any other class can invoke that method. For getter and setter methods this is the usual practice.

*public*

When a method is designated as `private` the method can only be called from within the class where it is defined. In our `Student` example we use a private method that controls the value of the field `lastId`. Consider the constructor below and the utility method `nextId()`. This utility method is only called from a constructor and nothing outside the class can call it. In this way the value of `lastId` can never be changed except when a `Student` object is created.

*private*

```
public Student(){
    id = nextId();
}
private int nextId(){
    // increment lastId and return the new value
    // to be used for the new student.
    return ++lastId;
}
```

## Constructors

When a constructor is designated `public` then any other class can use that constructor to create an object. This is the usual case for constructors.

*public*

When a constructor is designated as `private` the constructor can only be called from within the class where it is defined. One class we have used that has a private constructor is `Math`. `Math` is a utility class made up of static fields and static methods, and so there would be no benefit to ever having

*private*

a `Math` object. The private constructor prevents anyone from instantiating a `Math` object.

If you try to instantiate an instance of `Math` your program will not compile; the message you receive back from the compiler is "Math() has private access in java.lang.Math".

## 8.6 Overloading

The Java language allows you to define more than one method with the same method name as long as the parameter lists are different. If two or more methods have the same name we say the name is *overloaded*. The same is true for constructors - a class can have more than one constructor as long as their parameters lists are different. We have seen overloading in practice with the previous example where two constructors were coded for the `Student` class.

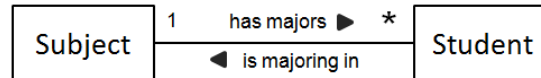
*overloaded*



## 8.7 Associations

Recall the class diagram from Figure 8.1 reproduced below.

Figure 8.6: A simple Class Diagram.



To implement this system of classes we need to define a Subject class. Later we discuss the implementation of the association. To keep our example simple Subject is defined to have two fields - see below.

```

/**
 * Subject area in which a student
 * may declare a major.
 */
public class Subject
{
    // fields
    private String code;
    private String name;
    //constructor
    public Subject(String code, String name)
    {
        this.code = code;
        this.name = name;
    }
    // getters
    public String getCode(){
        return code;
    }
    public String getName(){
        return name;
    }
    // setters
    public void setCode(String code){
        this.code = code;
    }
    public void setName(String name){
        this.name = name;
    }
}

```

```

    }
    // toString()
    public String toString(){
        return code+" (" +name+" )";
    }
    // equals(...)
    public boolean equals(Subject other){
        return this.code == other.code;
    }
}

```

### Implementing the association

*2 sides of an  
association*

When we implement an association we must consider that the association can be viewed from either side of the association. So in this case we can say there is a *student side* and a *subject side*. Such an association is sometimes called a *one-to-many* association because for each student there is only one subject area, and for each subject area there can be many related students.

*student side*

Let us consider the student side, *is majoring in*, first. We have stated that a student chooses one major and so a simple field could be used for this purpose. We add a field named `major` and so the fields in `Student` now become:

```

private static int lastId;
private int id;
private String firstName;
private String lastName;
private char gender;
private boolean active;
private Subject major;

```

Notice that the last field `major`, of type `Subject`, is the field that implements the association for a student object. For each student object there is field that can be set to the student's major subject. Following the convention of getters and setters we add the following two methods to `Student`:

*"1" side: field of type Student*

```
// getter
public Subject getMajor(){
    return major;
}
// setter
public void setMajor(Subject newMajor){
    major = newMajor;
}
```

To show how we can make use of the above field, consider the code segment below where a student `sam` and a subject area `math` are instantiated, and then we add `math` as `sam`'s major.

```
public static void main(String[] args{
    Subject math = new
        Subject("Math", "Mathematics");
    Student sam = new
        Student("Samantha", "Jones", 'F', true);
    sam.setMajor(math);
}
```

Now, we consider the other side of the association ... the *subject side*. To implement the subject side of the association, *has majors*, we need to add a field to the `Subject` class. Since many students could have the same major a good name for the field is `majors` and a good choice for type is `ArrayList<Student>` as this allows for any number of students.

*subject area side*

*"many" side: field of type ArrayList*

We show selected lines of the `Subject` class listed below. In line 5 we declare an `ArrayList` of `Students`, and in the constructor (lines 8-13) an empty `ArrayList<Student>` is created. There is a getter for `majors` in lines 18-20 that returns `majors`, and a setter in lines 24-26 where there is a parameter of type `ArrayList<Student>`. In addition to the getters and setters we include one more method `addMajor( )`, in lines 28-30, that allows one to add a student to the existing list of majoring students.

```
1 public class Subject
2 {
3     // fields
4     ...
5     ArrayList<Student> majors;
6
7     //constructor
8     public Subject(String code, String name)
9     {
10         this.code = code;
11         this.name = name;
12         majors = new ArrayList();
13     }
14
15     // getters
16     ...
17
18     public ArrayList<Student> getMajors(){
19         return majors;
20     }
21     // setters
22     ...
23
24     public void setMajors(ArrayList<Student>
25         majors){
26         this.majors = majors;
27     }
28     // add a student to those majoring in this area
29     public void addMajor(Student newMajor){
30         majors.add(newMajor);
31     }
32     ...
33 }
```

Now reconsider the case where there is a student `sam` and a subject area `math`, and where `sam` declares `math` as the major. There are two actions we must perform relating to the two *sides* of an association. Not only do we add `math` as the major for `sam`, but we add `sam` to the list of majoring students for `math`. Consider the following program where lines 13 and 14 perform the *transaction* for declaring a major.

*transaction has  
2 actions*

Listing 8.2: Sam declares a Math major

```
1
2 /**
3  * Create a student Sam and a subject area Math
4  * and then code the action of
5  * Sam declaring a major in Math
6  */
7 public class SamDeclaresMathMajor
8 {
9     public static void main(String[] args){
10         Subject math = new
11             Subject("Math", "Mathematics");
12         Student sam = new
13             Student("Samantha", "Jones", 'F', true);
14         // two actions for the "declare major"
15         // transaction
16         sam.setMajor(math);
17         math.addMajor(sam);
18         System.out.println("Math majors = "
19             +math.getMajors());
20     }
21 }
```

## 8.8 Reusing code

*eliminate  
duplicate code*

Consider a case where we want to declare a major for several students. We can choose to duplicate lines 13 and 14 in Listing 8.2 several times, or, we can create a method that contains the equivalent of lines 13 and 14, and invoke that method as many times as necessary. In this way we can eliminate code redundancy and can make the overall program easier to comprehend. And if the code for declaring a major has to change, the change affects the small method `declareMajor()`. Consider the following example where we call `declareMajor()` several times:

Listing 8.3: Invoking a method several times.

```
1  /**
2   * Instead of duplicating code, use a method to
3   * perform the same action three times.
4   */
5  public class DeclareSubjectAreaForMultipleStudents
6  {
7      public static void main(String[] args){
8          Subject math = new
9              Subject("Math", "Mathematics");
10         Student jill = new
11             Student("Jill", "Lee", 'F', true);
12         Student sam = new
13             Student("Samantha", "Jones", 'F', true);
14         Student bob = new
15             Student("Robert", "Smith", 'M', true);
16         // Each student is majoring in Math
17         declareMajors(jill, math);
18         declareMajors(sam, math);
19         declareMajors(bob, math);
20         System.out.println("Math majors = "
21             +math.getMajors());
22     }
23     public static void declareMajors(Student s,
24         Subject m){
25         // student s declares a major in m
26         s.setMajor(m);
27         m.addMajor(s);
28     }
29 }
```

24 }

The output below shows the students who have declared their major to be Mathematics.

```
Options
Math majors = [1 Jill Lee, 2 Samantha Jones, 3 Robert Smith]
```

## 8.9 Parameter lists and arguments

Constructors and methods can be defined with a parameter list that comprises one or more variables, and these variables can be primitive types, classes, arrays, ArrayLists, etc. When one method (the caller) invokes another method (the called method) control transfers from the caller to the called method, but before this occurs the values of the arguments are copied to corresponding parameters in the called method. It is required that the caller's arguments match in number and type with the called method's parameters.

*copying of arguments  
into parameters*

When the called method completes and control returns back to the caller the opposite does not occur - parameter values are not copied back into the arguments. The caller's fields are not altered unless an argument is an object **and** the called method has modified the object. If the called method alters the object then the caller's argument (object) is changed.

*no copying on return*

Recall that a variable that represents an object holds a reference to (the memory address of) the object. So when an object is passed it is the reference that is copied to the parameter. When the called method accesses the object through the parameter it is accessing the object and not some copy of the object.

*passing objects*

Consider the program below and the method `removeMajors()` that has two parameters: the first is `n`, an `int` which is a primitive data type, but the second one is an object - an `ArrayList` named `x`. The method removes `n` elements from `x` decrementing `n` in the process. From the output you will see the arguments before and after the call. The `ArrayList` is changed, but the `int` variable is not changed.



Listing 8.4: Objects can be modified through a method call.

```
1 import java.util.ArrayList;
2 /**
3  *
4  * Arrange for 3 students to major in Math.
5  * The method removeMajors() has 2 parameters.
6  * removeMajors() alters both parameters.
7  * In the caller only the ArrayList changes,
8  * the "int" is not changed.
9  *
10 */
11 public class ObjectModifiedByCalledMethod
12 {
13     public static void main(String[] args){
14         Subject math = new
15             Subject("Math","Mathematics");
16         Student jill = new
17             Student("Jill","Lee",'F',true);
18         Student sam = new
19             Student("Samantha","Jones",'F',true);
20         Student bob = new
21             Student("Robert","Smith",'M',true);
22         // Each student is majoring in Math
23         declareMajors(jill, math);
24         declareMajors(sam, math);
25         declareMajors(bob, math);
26         // the majors
27         ArrayList<Student> majors =
28             math.getMajors();
29         // n is the number of Math majors
30         int n = majors.size();
31         System.out.println(n+" majors: "+majors);
32         removeMajors(n, majors);
33         // n is not changed
34         // majors has been changed
35         System.out.println(n+" majors: "+majors);
36     }
37 }
```

```
34     public static void removeMajors
35         (int n, ArrayList<Student> x)
36     {
37         while (n>0){
38             n--;           // decrement n
39             x.remove(n); // remove the nth. element
40         }
41     }
42
43     public static void declareMajors
44         (Student s, Subject m)
45     {
46         // s declares a major in m
47         s.setMajor(m);
48         m.addMajor(s);
49     }
50 }
```

The output below shows the arguments, an `int` and an `ArrayList`, before and after the call.

---

Options

```
3 majors: [1 Jill Lee, 2 Samantha Jones, 3 Robert Smith]
3 majors: []
```

## 8.10 Varargs: a variable number of arguments

A method can be defined such that the last parameter can accept multiple values. Between the parameter type and the parameter name for the last parameter we include an ellipsis (...). In the method the last parameter is then an array.

*varargs: denoted by ...*

Recall the last program, Listing 8.4, where each student declares a Mathematics major. That program invokes `declareMajors()` 3 times, once per student. In the program listing below we have changed the method in such a way that all students who declare the same major can be handled in a single call to the method.

In line 13 of the calling method there are three `Student` variables passed to `declareMajors()`:

```
declareMajors(math, jill, sam, bob);
```

In lines 18-19 we have declared a parameter list as:

```
public static void declareMajors(
    Subject m, Student ... s)
```

where the last parameter is defined as "`Student ... s`".

In lines 22-25 you can see the elements of `s` are accessed using an enhanced `for` statement.

```
for (Student student: s){
    student.setMajor(m);
    m.addMajor(student);
}
```

Listing 8.5: Multiple argument values for the last parameter.

```

1  /**
2   * 3 students declare Math - one call
3   */
4  public class VarargParameterForStudentObjects
5  {
6      public static void main(String[] args){
7          Subject math = new
8              Subject("Math", "Mathematics");
9          Student jill = new
10             Student("Jill", "Lee", 'F', true);
11         Student sam = new
12             Student("Samantha", "Jones", 'F', true);
13         Student bob = new
14             Student("Robert", "Smith", 'M', true);
15         // Each student is majoring in Math
16         // A single call to declareMajors
17         declareMajors(math, jill, sam, bob);
18         System.out.println("Math majors = "
19             +math.getMajors());
20     }
21     // varargs used for last parameter
22     public static void declareMajors(
23         Subject m, Student ... s){
24         // Set each student to have a major in m.
25         // s is an array of Student
26         for (Student student: s){
27             student.setMajor(m);
28             m.addMajor(student);
29         }
30     }
31 }

```

When the program is run we get the same output as before - showing three Mathematics majors:

```

Options
Math majors = [1 Jill Lee, 2 Samantha Jones, 3 Robert Smith]

```

## 8.11 Code listings: Student, Subject

Listing 8.6: The Student class.

```
1  /**
2   * A student.
3   */
4  public class Student {
5      // class fields
6      private static int lastId;
7      // instance fields
8      private int id;
9      private String firstName;
10     private String lastName;
11     private char gender;
12     private boolean active;
13     private Subject major;
14     // first constructor, no arguments
15     public Student(){
16         id = nextId();
17         // default values for a student:
18         firstName = "unknown";
19         lastName = "unknown";
20         gender = '?';
21         active = false;
22     }
23     // second constructor, four arguments
24     public Student (String firstName, String
25         lastName, char gender, boolean active){
26         id = nextId();
27         //
28         // when parameters and fields have the same
29         // name they are distinguished this way:
30         // a field name alone refers to the
31         // parameter
32         // a field name prefixed with "this."
33         // refers to an object's fields.
34         this.firstName = firstName;
35         this.lastName = lastName;
36         this.gender = gender;
```

```
35         this.active = active;
36     }
37
38     private int nextId(){
39         // increment lastId and return the new value
40         // to be used for the new student.
41         return ++lastId;
42     }
43
44     public int getId(){
45         return id;
46     }
47
48     public static int getLastId(){
49         return lastId;
50     }
51
52     public String getFirstName(){
53         return firstName;
54     }
55
56     public String getLastName(){
57         return lastName;
58     }
59
60     public char getGender(){
61         return gender;
62     }
63
64     public boolean isActive(){
65         return active;
66     }
67
68     public Subject getMajor(){
69         return major;
70     }
71
72     public void setLastId(int newLastId){
73         lastId = newLastId;
74     }
```

```
75
76     // no setter for the student's id field
77     //     public void setId(int newId){
78     //         id = newId;
79     //     }
80
81     public void setFirstName(String newFirstName){
82         firstName = newFirstName;
83     }
84
85     public void setLastName(String newLastName){
86         lastName = newLastName;
87     }
88
89     public void setGender(char newGender){
90         gender = newGender;
91     }
92
93     public void setActive(boolean newActive){
94         active = newActive;
95     }
96
97     public void setMajor(Subject newMajor){
98         major = newMajor;
99     }
100
101     public String toString(){
102         return id+" "+firstName+" "+lastName;
103     }
104
105     public boolean equals(Student s){
106         return id == s.id;
107     }
108 }
```

Listing 8.7: The Subject class.

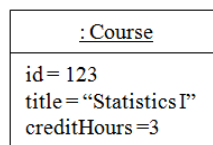
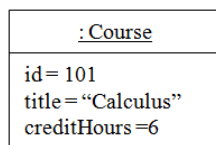
```
1 import java.util.ArrayList;
2 /**
3  * Subject area in which a
4  * student may declare a major.
5  */
6 public class Subject
7 {
8     // fields
9     private String code;
10    private String name;
11    private ArrayList<Student> majors;
12    //constructor
13    public Subject(String code, String name)
14    {
15        this.code = code;
16        this.name = name;
17        majors = new ArrayList();
18    }
19    // getters
20    public String getCode(){
21        return code;
22    }
23
24    public String getName(){
25        return name;
26    }
27
28    public ArrayList<Student> getMajors(){
29        return majors;
30    }
31    // setters
32    public void setCode(String code){
33        this.code = code;
34    }
35
36    public void setName(String name){
37        this.name = name;
38    }
```



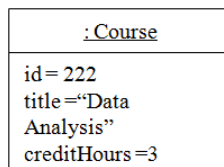
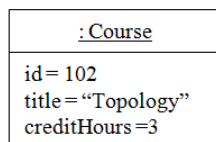
```
39
40     public void setMajors(ArrayList<Student>
        majors){
41         this.majors = majors;
42     }
43     // add a student to those majoring in this area
44     public void addMajor(Student newMajor){
45         majors.add(newMajor);
46     }
47     // toString()
48     public String toString(){
49         return code+" (" +name+" )";
50     }
51     // equals(...)
52     public boolean equals(Subject other){
53         return this.code == other.code;
54     }
55 }
```

## Exercises

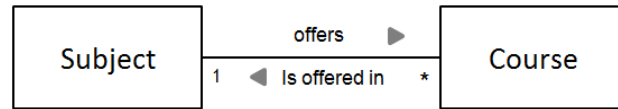
1. Create each of the three students shown in the object diagram of Figure 8.3 and the subject areas Computer Science and Mathematics. Write the code to implement the transactions:
  - Samantha declares a major in Computer Science;
  - Robert declares a major in Computer Science;
  - Jill declares a major in Mathematics.
2. Suppose we must create a way for a student to change their major. This can be done by adding another method to Subject that would remove a student from the list of students majoring in a subject area. Write this method and then write the code necessary for the following transactions:
  - Samantha declares a major in Computer Science;
  - Robert declares a major in Computer Science;
  - Jill declares a major in Mathematics.
  - Robert re-declares his major to be Mathematics.
3. Add a new method to the Student class that returns a student's full name.
4. In Listing 8.5 there are three `Student` objects passed to `declareMajors()`. Because the last argument in `declareMajors()` is an array you can pass an array instead of the three `Student` arguments. Place jill, sam, and bob in a `Student` array and call the method using two arguments: the subject and the `Student` array.
5. Develop a `Course` class and then use a program to instantiate the following objects:



Note the objects are unnamed but the top compartment shows the class the object belongs to.



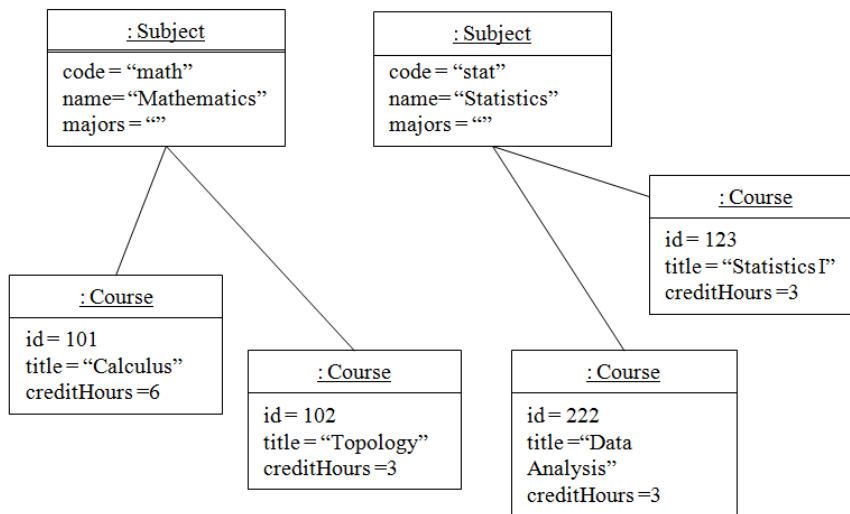
6. Implement the following business rule:



- A subject offers several courses;
- A course is offered in one subject.

Write a program that creates two subject areas and four courses as shown in the following object diagram. Note the lines that connect a subject to a course - these link lines show which courses are offered by which subject area.

In this object diagram each object is unnamed. Note however, the top compartment shows the class the object belongs to.



Then, for each subject area display the courses it offers.



## Chapter 9

# A Brief Introduction to Graphical User Interfaces

The way a user interacts with a computing system is termed the user interface (UI). Designing and developing a UI is critical to the acceptance and success of a computing system. A UI is the user's view of the system and its design should not be taken lightly. Many educational institutions have a course devoted to this very subject.//

When learning to program we tend to keep the UI quite simple. This has been the case for all of our programs. One could consider our discussion on `JOptionPane` to be a small exception as it is the one case where the user can use a mouse to interact with the program; in all other cases the user's interaction required using the keyboard. `JOptionPane` provides only a limited amount of non-keyboard interaction.

In this chapter we introduce the graphical user interface (GUI), pronounced *gooey* or spelled out as G-U-I, where a user interacts with a Java system via the keyboard, mouse, or other peripherals, and graphical components such as buttons, text fields, radio buttons, etc.

It is very likely the student has used many GUIs. Some common examples where we use GUIs:

1. an automated teller machine
2. a airline check-in system
3. a cell phone or tablet computer

## 4. an email system

In this chapter we examine some of the elements that go into a GUI and how those elements are managed in Java. As you will see a GUI involves a lot of coding, but we will facilitate our coding by using the Simple GUI Builder (a BlueJ extension). If the Simple GUI Builder extension is not installed on your computer you should visit the BlueJ web page and follow the instructions for download and installation of the extension.

The typical GUI is presented as a window on the user's monitor. A window is called a container comprising other components. There are many such components the GUI programmer must master; we will consider a few of the most common components and give you a sense of basic GUI programming:

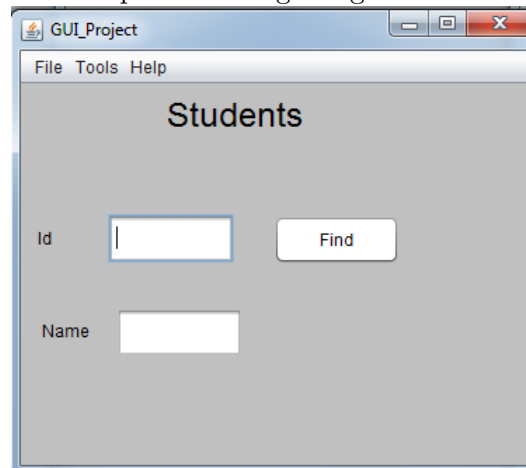
1. Labels. These are textual components that provide descriptive information for the user. Typically they provide a title for a window or describe what appears in another component. They are visual components the user see but does not interact with.
2. Text Fields. The primary purpose of a text field is to provide an element in a window where the user can enter data. These components must have meaningful names as they will be referenced in code through set and get type methods.
3. Buttons. These components may show a picture or text indicating what happens when they are clicked with a mouse. These will need meaningful names as they are reference directly in code. The action to be performed is a method that executes when the user clicks the button, and this needs to be coded by the programmer.

There are of course many more components including radio buttons, check boxes, combo boxes, panels, borders, etc. The interested reader is referred to other articles and texts such as those found at <http://docs.oracle.com/javase/tutorial>

## 9.1 Brief Introduction to Simple GUI Builder

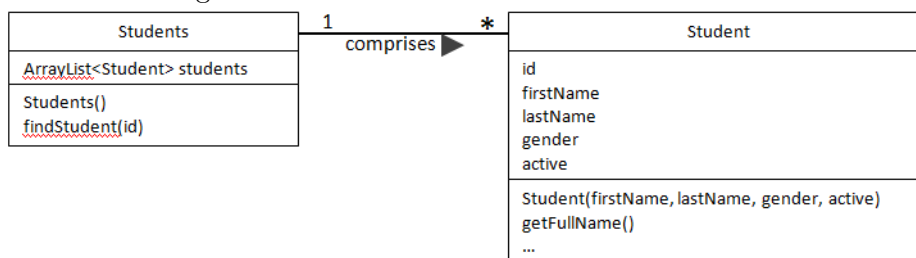
The sample GUI (see Figure 9.1) we will build in this section permits a user to enter a student identifier, click the Find button, and in response the system will display the student's full name.

Figure 9.1: Sample GUI for getting student information.



Our example will utilize two classes, `Students` and `Student`, that were presented in the previous chapter and illustrated in Figure 9.2.

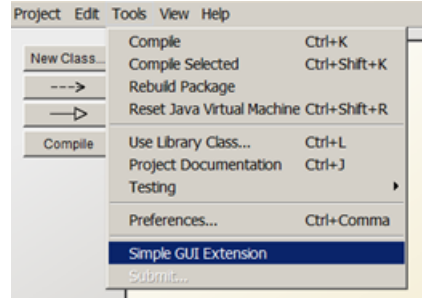
Figure 9.2: `Students` is a collection of students.



To work through this example you need to start a BlueJ project and include `Students` and `Student` classes. To start building a GUI you must select the Tools menu and then select the Simple GUI Extension (see Figure 9.3).

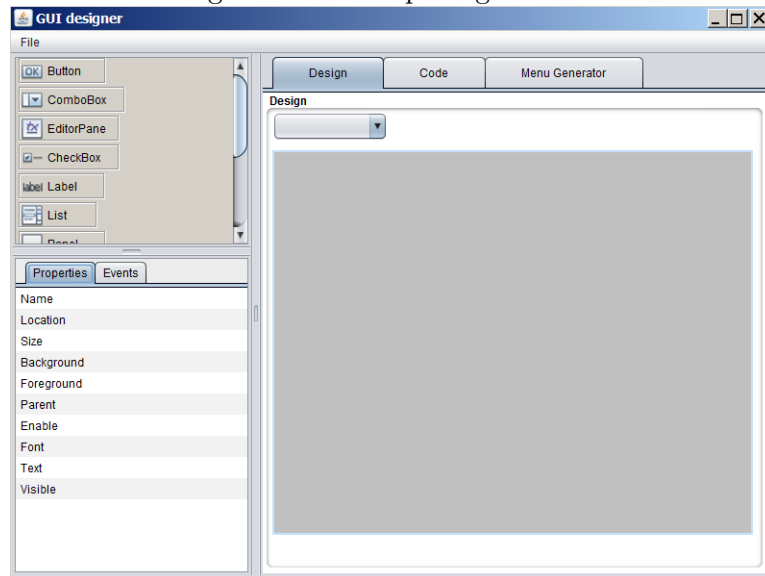
When the Simple GUI Extension is selected a window opens where you can

Figure 9.3: Starting the Simple GUI Builder.



design the GUI, examine generated Java code, and create a menu for your GUI (menus are ignored in this brief introduction). The window appears as shown in Figure 9.4 where, of the three tabs, the Design tab is selected showing a large blank area where you *draw* the GUI using drag and drop features of the builder.

Figure 9.4: The opening window.



At this point if you switch to the Code tab you will be able to see generated code similar to Listing 9.1. In the listing displayed below we have removed lines referring to a menu in order to focus on specific aspects. Comments



follow the listing.

Listing 9.1: The initial code generated.

```
1  /**
2  * Several imports for GUI related classes
3  */
4  import javax.swing.UIManager.LookAndFeelInfo;
5  import java.awt.*;
6  import java.awt.event.ActionEvent;
7  import java.awt.event.ActionListener;
8  import java.awt.event.KeyAdapter;
9  import java.awt.event.KeyEvent;
10 import java.awt.event.MouseAdapter;
11 import java.awt.event.MouseEvent;
12 import java.awt.event.MouseWheelEvent;
13 import java.awt.event.MouseWheelListener;
14 import javax.swing.border.Border;
15 import javax.swing.*;
16
17 // The class extends the Java class named JFrame
18 public class initialGUICodeListing extends JFrame {
19
20     // When you GUI has a menu, you need a JMenuBar
21     private JMenuBar menuBar;
22
23     // This constructor builds your GUI
24     public initialGUICodeListing(){
25         // Title and size for the window
26         this.setTitle("GUI_project");
27         this.setSize(500,400);
28         //
29         generateMenu();
30         this.setJMenuBar(menuBar);
31
32         // Your GUI needs a JPanel to hold
33         // the labels, etc
34         JPanel contentPane = new
35             JPanel(null);
36         contentPane.setPreferredSize(new
37             Dimension(500,400));
```

```

35         contentPane.setBackground(new
36             Color(192,192,192));
37
38         // The JFrame is a container to
39         // which the panel is added
40         this.add(contentPane);
41
42         // some lines removed ...
43
44         // The JFrame must be visible to
45         // the user
46         this.setVisible(true);
47     }
48
49     //method for generate menu
50     public void generateMenu(){
51         menuBar = new JMenuBar();
52         // some lines removed ...
53     }
54
55     // the main() method creates a running GUI
56     public static void main(String [] args){
57         System.setProperty("swing.defaultlaf",
58             "com.sun.java.swing.plaf.nimbus.NimbusLookAndFee
59             javax.swing.SwingUtilities.invokeLater(new
60                 Runnable() {
61                     public void run() {
62                         new GUI_project();
63                     }
64                 });
65     }
66 }

```

- lines 4 to 15 are import statements for Java class libraries and classes that may be used in the GUI.
- line 18 indicates the class *extends* `JFrame`. `JFrame` is a Java class that provides the basic container for the window.

- The constructor begins in line 24 and builds the GUI object that will be displayed to the user.
- lines 26 and 27 set the title and size of the window
- We are ignoring the menu built in line 30
- Several lines (startin at line 33) refer to a `JPanel` that serves as the container holding labels, text fields and the button of our interface.
- Line 52 is the start of the `main()` method that instructs the JVM to *create* and *run* the GUI object created in the constructor.

To briefly summarize the above we can say the GUI is a visible `JFrame` object that contains a `JPanel` object. Now, we want to complete the GUI with the components that will be in the `JPanel`. Consider Figure 9.4 again. In the upper left hand corner contains the controls for buttons, combo boxes, check boxes, labels, text fields, etc. that can be added to a GUI. These controls can be dragged and dropped on the panel; once selected they can be resized and repositioned via the mouse. Below the controls you see properties for the selected control where you can modify the name, location, size, text, etc. It is useful to assign a meaningful name so you can find it in the code. For some controls such as a button you should specify meaningful text to appear on the control and an event handler (a method to execute when, say, a user clicks a button control).

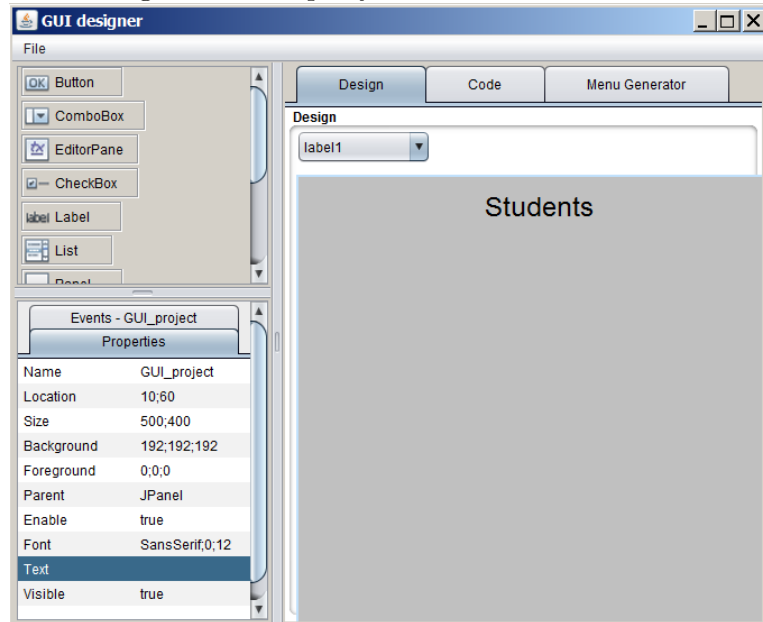
The Java code to create the above GUI can be done fairly quickly using the Simple GUI Builder. This simple GUI has three labels, two text fields, and a button. We will outline the steps to create the labels, then the text boxes, and lastly the button.

### Step 1. Add three labels

To add a label click the Label control in the upper left of the tool's window and hold the mouse button down. Next, drag the cursor to where the label is to appear on the panel. Later if necessary, you can move the label by clicking and dragging.

a) Add a label at the top center; you will see the properties for the label in the lower left. Enter its text property as *Students* and set its font size to be *24*. You should now see properties and the design as in Figure 9.5.

Figure 9.5: Property values for the 3 labels.



- b) Add a label on the left as per Figure 9.1; set its text property to Id;
  - c) Add a label on the left as per Figure 9.1; set its text property to Name
- The properties for each of the above labels is shown in Figure 9.6

## Step 2. Add two text fields

To add a text field click the Text Field control in the upper left of the tool's window and hold the mouse button down. Next, drag the cursor to where the text field is to appear on the panel.

We are adding two text fields that are used to hold values for a student's identifier and for their full name. Proceed as follows:

- a) Add a text field to the right of the Id label as per Figure 9.1. Delete the current value of the Text property. Set the Name property to textfieldId.
  - a) Add a text field to the right of the name label as per Figure 9.1. Delete the current value of the Text property. Set the Name property to textfieldName
- The property values for these text fields should be as follows:

Figure 9.6: Property values for the 2 text fields.

Properties	Events - textfieldId	Properties	Events - textfieldName
Name	textfieldId	Name	textfieldName
Location	68;92	Location	68;160
Size	90;35	Size	90;35
Background	255;255;255	Background	255;255;255
Foreground	0;0;0	Foreground	0;0;0
Parent	JLayeredPane	Parent	JLayeredPane
Enable	true	Enable	true
Font	SansSerif;0;12	Font	SansSerif;0;12
Text		Text	
Visible	true	Visible	true

### Step 3. Add a button

To add a button to the UI you must click the Button control in the upper left of the tool's window and hold the mouse button down. Next drag the cursor on to the panel where the button is to appear. In the case of our simple GUI where we want one button as per Figure 9.1:

- Position the button to the right of `textfieldId`.
- Set its Name property to `buttonFindStudent`.
- Set the Text property to `Find`.

Figure 9.7: Property values for the button.

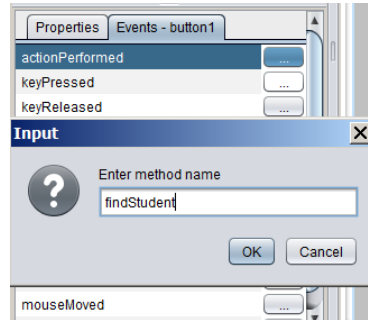
Properties	Events - buttonFindStudent
Name	buttonFindStudent
Location	181;95
Size	90;35
Background	214;217;223
Foreground	0;0;0
Parent	JLayeredPane
Enable	true
Font	SansSerif;0;12
Text	Find
Visible	true

One aspect of a button is that, when clicked, some action must be performed, and this action must be defined as a Java method. For this button you must:

- Click on the Events tab (beside "Properties")
- Click the button beside "actionPerformed" and enter the method name `findStudent` in the pop-up window.

You should now see an Events tab as shown in Figure 9.8.

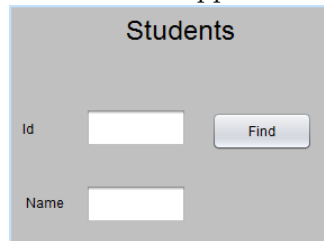
Figure 9.8: The method to execute when the button is clicked.



#### Step 4. Create the GUI class

To create the GUI class you can click Save in the GUI Designer's File menu. It will be named GUI\_Project and once compiled you can execute the `main()` method. What you should see is a window similar to Figure 9.9. Since we have not completed the event handler method `findStudent()` for the Find button nothing happens when you click Find.

Figure 9.9: The GUI appears to the user.



#### Step 5. Modify the code (for Students)

Modify the generated code in GUI\_Project as shown below. Note that when this is done you cannot go back to the GUI tool - if you do, you will lose these modifications. The GUI Designer is considered a *one-way* tool - you can only generate code; you cannot generate a design from the code.

a) We need a variable that references the collection of students. Locate the fields and add the declaration

```
private Students students;
```

b) We need to create the `students` object. Locate the constructor and add the statement

```
this.students = new Students();
```

c) Examine the `main()` method. This method instantiates a `GUI_Project` and in the above two points we have arranged for the `GUI_Project` to contain a list of students.

### Step 6. Modify the code (provide an action)

Modify generated code in `GUI_Project` to perform a meaningful action when the button is clicked.

a) Locate the `findStudent()` method

b) The generated code contains a comment `// TODO` which you can replace with:

```
1     int id =
        Integer.parseInt(textfieldId.getText());
2     Student s = students.findStudent(id);
3     if (s == null)
4         textfieldName.setText("NOT FOUND");
5     else
6         textfieldName.setText(s.getFullName());
```

In line 1 the value entered by the user in a text field is converted to an `int`, and then in line 2 that value is passed to the `findStudent()` method of the `students` object. In line 3 a check is made to determine if a corresponding student was found. If the student was not found then the message `NOT FOUND` is displayed in the GUI; otherwise the student's full name is obtained and displayed in the GUI.

### Step 7. Run/experiment with your GUI

Note that when you run `main()` the GUI shows on your monitor, and in the System tray at the bottom of your monitor you see the Java icon for your GUI:

Figure 9.10: The GUI appears to the user.



## Summary

When the above actions were performed a class named GUI\_project was created; its code, and that of Students and Student are shown in the listings that follow. To summarize the GUI created here:

- We used a BlueJ extension called the Simple GUI Designer.
- Using drag and drop features we designed a GUI with a labels, text fields, and a button.
- Controls such as text fields and buttons require meaningful names as they are always referenced in code.
- A button is different as there is always an action to be coded that is invoked when the button is pressed or clicked.
- The code involved for GUIs is complex and lengthy. A tool like the Simple GUI Designer is very useful in practice.

### 9.1.1 Listings

Listings for the GUI.

#### The GUI\_project Class

Listing 9.2: The generated class GUI\_project with modifications.

```

1  /**
2  *Text genereted by Simple GUI Extension for BlueJ
3  */
4  import javax.swing.UIManager.LookAndFeelInfo;
5  import java.awt.*;
6  import java.awt.event.ActionEvent;
7  import java.awt.event.ActionListener;
8  import java.awt.event.KeyAdapter;
9  import java.awt.event.KeyEvent;
10 import java.awt.event.MouseAdapter;
11 import java.awt.event.MouseEvent;
12 import java.awt.event.MouseWheelEvent;
13 import java.awt.event.MouseWheelListener;
14 import javax.swing.border.Border;
15 import javax.swing.*;

```



```
16
17
18 public class GUI_Project extends JFrame {
19     private Students students;
20     private JMenuBar menuBar;
21     private JButton buttonFindStudent;
22     private JLabel label1;
23     private JLabel label2;
24     private JLabel label3;
25     private JTextField textfieldName;
26     private JTextField textfieldId;
27
28     //Constructor
29     public GUI_Project(){
30
31         this.students = new Students();
32         this.setTitle("GUI_Project");
33         this.setSize(500,400);
34         //menu generate method
35         generateMenu();
36         this.setJMenuBar(menuBar);
37
38         //pane with null layout
39         JPanel contentPane = new JPanel(null);
40         contentPane.setPreferredSize(new
41             Dimension(500,400));
42         contentPane.setBackground(new
43             Color(192,192,192));
44
45         buttonFindStudent = new JButton();
46         buttonFindStudent.setBounds(180,94,90,35);
47         buttonFindStudent.setBackground(new
48             Color(214,217,223));
49         buttonFindStudent.setForeground(new
50             Color(0,0,0));
51         buttonFindStudent.setEnabled(true);
52         buttonFindStudent.setFont(new
53             Font("sansserif",0,12));
54         buttonFindStudent.setText("Find");
```

```

51         buttonFindStudent.setVisible(true);
52         //Set action for button click
53         //Call defined method
54         buttonFindStudent.addActionListener(new
           ActionListener() {
55             public void actionPerformed(ActionEvent
           evt) {
56                 findStudent(evt);
57             }
58         });
59
60
61         label1 = new JLabel();
62         label1.setBounds(104,12,300,20);
63         label1.setBackground(new
           Color(214,217,223));
64         label1.setForeground(new Color(0,0,0));
65         label1.setEnabled(true);
66         label1.setFont(new Font("SansSerif",0,24));
67         label1.setText("Students");
68         label1.setVisible(true);
69
70         label2 = new JLabel();
71         label2.setBounds(12,93,90,35);
72         label2.setBackground(new
           Color(214,217,223));
73         label2.setForeground(new Color(0,0,0));
74         label2.setEnabled(true);
75         label2.setFont(new Font("sansserif",0,12));
76         label2.setText("Id");
77         label2.setVisible(true);
78
79         label3 = new JLabel();
80         label3.setBounds(15,159,90,35);
81         label3.setBackground(new
           Color(214,217,223));
82         label3.setForeground(new Color(0,0,0));
83         label3.setEnabled(true);
84         label3.setFont(new Font("sansserif",0,12));
85         label3.setText("Name");

```

```
86         label3.setVisible(true);
87
88         textfieldName = new JTextField();
89         textfieldName.setBounds(68,160,90,35);
90         textfieldName.setBackground(new
91             Color(255,255,255));
91         textfieldName.setForeground(new
92             Color(0,0,0));
92         textfieldName.setEnabled(true);
93         textfieldName.setFont(new
94             Font("sansserif",0,12));
94         textfieldName.setText("");
95         textfieldName.setVisible(true);
96
97         textfieldId = new JTextField();
98         textfieldId.setBounds(62,93,90,35);
99         textfieldId.setBackground(new
100             Color(255,255,255));
100         textfieldId.setForeground(new Color(0,0,0));
101         textfieldId.setEnabled(true);
102         textfieldId.setFont(new
103             Font("sansserif",0,12));
103         textfieldId.setText("");
104         textfieldId.setVisible(true);
105
106         //adding components to contentPane panel
107         contentPane.add(buttonFindStudent);
108         contentPane.add(label1);
109         contentPane.add(label2);
110         contentPane.add(label3);
111         contentPane.add(textfieldName);
112         contentPane.add(textfieldId);
113
114         //adding panel to JFrame and setting of
115             window position and close operation
115         this.add(contentPane);
116         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
117         this.setLocationRelativeTo(null);
118         this.pack();
119         this.setVisible(true);
```

```
120     }
121
122     //Method actionPerformed for buttonFindStudent
123     private void findStudent (ActionEvent evt) {
124         int id =
125             Integer.parseInt(textfieldId.getText());
126         Student s = students.findStudent(id);
127         if (s == null)
128             textfieldName.setText("NOT FOUND");
129         else
130             textfieldName.setText(s.getFullName());
131     }
132
133     //method for generate menu
134     public void generateMenu(){
135         menuBar = new JMenuBar();
136
137         JMenu file = new JMenu("File");
138         JMenu tools = new JMenu("Tools");
139         JMenu help = new JMenu("Help");
140
141         JMenuItem open = new JMenuItem("Open  ");
142         JMenuItem save = new JMenuItem("Save  ");
143         JMenuItem exit = new JMenuItem("Exit  ");
144         JMenuItem preferences = new
145             JMenuItem("Preferences  ");
146         JMenuItem about = new JMenuItem("About  ");
147
148         file.add(open);
149         file.add(save);
150         file.addSeparator();
151         file.add(exit);
152         tools.add(preferences);
153         help.add(about);
154
155         menuBar.add(file);
156         menuBar.add(tools);
157         menuBar.add(help);
158     }
```

```
158
159
160
161     public static void main(String[] args){
162         System.setProperty("swing.defaultlaf",
163             "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel");
164         javax.swing.SwingUtilities.invokeLater(new
165             Runnable() {
166                 public void run() {
167                     new GUI_Project();
168                 }
169             });
170     }
```

### The Students Class

Listing 9.3: The Students class - a collection of students.

```
1 import java.util.ArrayList;
2 public class Students
3 {
4     ArrayList<Student> myList;
5     public Students(){
6         // create a list myList of three students
7         myList = new ArrayList();
8         myList.add(new
9             Student("Jill", "Lee", 'F', true));
10        myList.add(new
11            Student("Samantha", "Jones", 'F', true));
12        myList.add(new
13            Student("Robert", "Smith", 'M', true));
14    }
15    public Student findStudent(int id){
16        for (Student s: myList)
17            if (s.getId() == id) return s;
18        return null;
19    }
20 }
```

**The Student Class**

Listing 9.4: The Student class.

```

1  /**
2   * A student.
3   */
4  public class Student {
5      // class fields
6      private static int lastId;
7      // instance fields
8      private int id;
9      private String firstName;
10     private String lastName;
11     private char gender;
12     private boolean active;
13     private Subject major;
14     // first constructor, no arguments
15     public Student(){
16         id = nextId();
17         // default values for a student:
18         firstName = "unknown";
19         lastName = "unknown";
20         gender = '?';
21         active = false;
22     }
23     // second constructor, four arguments
24     public Student (String firstName, String
25         lastName, char gender, boolean active){
26         id = nextId();
27         //
28         // when parameters and fields have the same
29         // name they are distinguished this way:
30         // a field name alone refers to the
31         // parameter
32         // a field name prefixed with "this."
33         // refers to an object's fields.
34         this.firstName = firstName;
35         this.lastName = lastName;
36         this.gender = gender;

```

```
35         this.active = active;
36     }
37
38     private int nextId(){
39         // increment lastId and return the new value
40         // to be used for the new student.
41         return ++lastId;
42     }
43
44     public int getId(){
45         return id;
46     }
47
48     public static int getLastId(){
49         return lastId;
50     }
51
52     public String getFirstName(){
53         return firstName;
54     }
55
56     public String getLastName(){
57         return lastName;
58     }
59
60     public char getGender(){
61         return gender;
62     }
63
64     public boolean isActive(){
65         return active;
66     }
67
68     public Subject getMajor(){
69         return major;
70     }
71
72     public void setLastId(int newLastId){
73         lastId = newLastId;
74     }
```

```
75
76     // no setter for the student's id field
77     //     public void setId(int newId){
78     //         id = newId;
79     //     }
80
81     public void setFirstName(String newFirstName){
82         firstName = newFirstName;
83     }
84
85     public void setLastName(String newLastName){
86         lastName = newLastName;
87     }
88
89     public void setGender(char newGender){
90         gender = newGender;
91     }
92
93     public void setActive(boolean newActive){
94         active = newActive;
95     }
96
97     public void setMajor(Subject newMajor){
98         major = newMajor;
99     }
100
101     public String toString(){
102         return id+" "+firstName+" "+lastName;
103     }
104
105     public boolean equals(Student s){
106         return id == s.id;
107     }
108 }
```



# Bibliography

- [1] <http://publications.gc.ca/gazette/archives/p1/2007/2007-03-24/pdf/g1-14112.pdf>
- [2] <http://www.acm.org/press-room/news-releases/2013/fellows-2013>
- [3] <http://www.ieee.org/documents/vonneumannrl.pdf>
- [4] *Effective Java*; Joshua Bloch; Addison-Wesley; Second edition; 2008; ISBN-13: 978-0321356680
- [5] *Java in a Nutshell*; Benjamin J. Evans, David Flanagan; O'Reilly; Sixth edition; 2015; ISBN-13: 978-1-44937082-4
- [6] *The Art of Computer Programming, Volume 2*; Donald Knuth; Addison-Wesley Professional; 3 edition; 1997; ISBN-13: 978-0201896848
- [7] *Clean Code: A Handbook of Agile Software Craftsmanship*; Robert C. Martin; Prentice Hall; 2008; ISBN-13: 978-0132350884