

Java with BlueJ and Pi

Ron McFadyen

May 11, 2015

Copyright

©2015 by Ron McFadyen

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/

To Callum

Contents

1	Introduction	1
1.1	Java, the beginning	1
1.2	Running Java Programs	2
1.3	A First Program	3
1.3.1	Exercises	3
1.4	BlueJ	4
1.5	Raspberry Pi	5
2	Basics	7
2.1	Variables	8
2.1.1	Exercises	9
2.2	<code>char</code>	9
2.3	<code>boolean</code>	9
2.4	<code>byte</code> , <code>short</code> , <code>int</code> , <code>long</code>	10
2.5	<code>float</code> , <code>double</code>	10
2.6	Calculations in Java	11
2.6.1	Expressions	12
2.6.2	Exercises	13
2.6.3	Exercises	17
2.6.4	Mixed Mode Expressions	17
2.6.5	Unary Operators	18
2.6.6	Exercises	19
2.7	The <code>String</code> Class	19
2.7.1	Catenation operator <code>+</code>	23
2.7.2	Exercises	23
2.8	Output	24
2.8.1	<code>System.out</code>	24
2.8.2	<code>JOptionPane</code>	25
2.9	Input	27
2.9.1	The <code>Scanner</code> Class	27
2.9.2	The <code>JOptionPane</code> Class	28

3	Control Structures	31
3.1	Compound statements	31
3.2	The while Statement	33
3.2.1	Exercises	36
3.2.2	Nesting statements	38
3.2.3	Autoincrement	39
3.2.4	Exercises	39
3.3	The if Statement	40
3.3.1	Exercises	41
3.3.2	Nesting statements	42
3.3.3	Exercises	45
3.4	The for Statement	46
3.5	The do while Statement	46
3.6	The switch Statement	46
3.7	Logical Expressions	47
3.7.1	Examples	48

Chapter 1

Introduction

This book is about programming in Java. We begin with short descriptions of Java, BlueJ, and the Raspberry Pi. 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 on one of the smallest and inexpensive computers available today, the Raspberry Pi. You will find exercises at the end of each section. Answers are available on the website for this text to all exercises with the exception of extension exercises.

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 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:

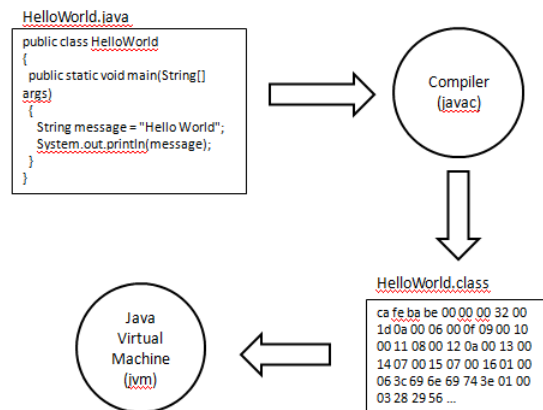
- 2007 - appointed an Officer of the Order of Canada.[1]
- 2013 - named an Association of Computing Machinery Fellow for "Java, NeWS, Emacs, NetBeans, and other contributions to programming languages, tools, and environments".[2]
- 2015 - awarded the IEEE John von Neumann Medal for "the Java programming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

1.2 Running Java Programs

When someone develops a Java program they must first enter the Java code in a text file; however, these files must have names that end with ".java". These files are known as source code files. In order to execute a Java program it must first of all be translated into Java bytecode. Source code files are human-readable but bytecode is just 0's and 1's. A program that does this 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 A First Program

The traditional first program to be seen in many texts is the HelloWorld program. When it is executed, this program does one simple thing: it displays the message "Hello World" and then ends.

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. A class like this has a method, and the third line gives the name *main* to this method. 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 a later chapter.
4. The lines that comprise the main method begin with the { in the fourth line and end with the } in the seventh line.
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 a memory location 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.

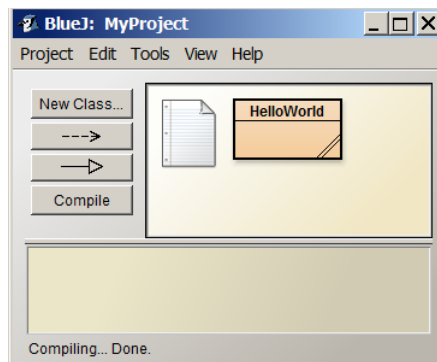
1.3.1 Exercises

1. Run the Hello World program.
2. Modify the Hello World program so it displays a different message.

1.4 BlueJ

BlueJ is an integrated development environment that provides the programmer with a framework that includes an editor, 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>. There is no point repeating a description and installation instructions that can be found at the BlueJ site. Below is a picture showing HelloWorld in a BlueJ project. Note the button available to compile the source code.



1.5 Raspberry Pi

The Raspberry Pi is a small computer developed by the Raspberry Pi foundation in order to promote the teaching of computer science. One remarkable point about the Raspberry Pi is its price: US\$35 (at the time of writing). The current model comes with 1 GB of memory, 4 USB ports, an ethernet port, a camera interface, HDMI and composite outputs, a micro SD slot, and a set of general purpose input/output pins. The computer weighs 45 grams and it all fits inside a case with dimensions $9.5 \times 6.2 \times 2.7\text{cm}$.

At the Raspberry Pi and BlueJ sites you will see that the Pi runs BlueJ (and all the programs included in this text have been run on the Pi). Of course one does need some additional components to make it useful: keyboard, mouse, monitor, internet. Below is a picture of one installed inside a clear case with a wireless keyboard/mouse adapter, wifi adapter, and HDMI cable attached.



Chapter 2

Basics

The focus of this chapter is literals, variables, primitive data types, and related expressions. It is common for programs to include constants; in Java these are referred to as literals. Examples include: 123, 123.45, 'a', "Gosling", true. You will see literals used in many examples to follow.

Variables are indispensable to programming. A variable is simply a name given to a piece of computer memory - a piece of memory that holds a value that a program can use and change as it executes. The Java programming language requires us to declare the type of data to be associated with a variable. There are eight primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`. When you develop a program you choose a specific data type depending on the nature of the data your program processes:

- `byte`, `short`, `int`, and `long` are used for cases where the data is to be treated as whole numbers. For example, 33, 498, -100 are whole numbers (numbers without a fractional component). 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', '\$'. When specific values are used they are enclosed in single quotes as shown.
- `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`.

2.1 Variables

The concept of a variable is very important to programming. A variable is a named location in a computer's memory. A Java programmer will declare variables in declaration statements and then use those variable names later in the program to refer to the value currently associated with the variable. Consider the program below which does the following:

- it declares an `int` variable called *i*,
- sets the value of *i* to 1,
- displays the value of *i*,
- changes the value of *i* to be 25,
- and then displays its new value.

Listing 2.1: Variable Declarations

```
1  /**
2   * This Java class declares an int variable,
3   * assigns values, and displays the values
4   */
5  public class Variables
6  {
7      public static void main(String[] args){
8          // declare a variable i to hold an int value
9          int i;
10         // set i to 1 and display the value
11         i = 1;
12         System.out.println ("i is: "+i);
13         // change the value of i and display it again
14         i = 25;
15         System.out.println ("i is: "+i);
16     }
17 }
```

As you study Java you will find that certain names are **keywords** and as such they cannot be used for variable names - keywords are reserved for special purposes only. The word `int` in a Java program is reserved for situations where one declares a variable to be of type `int`. For instance, you cannot declare a variable to have the name *int*. In sample programs we have seen a few of these reserved words: `public`, `class`, `void`, `static`.

Naming Variables

A convention used by many Java programmers is to choose names that are short 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 (that is not a keyword) 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 two distinct variables.

2.1.1 Exercises

1. Choose one of the programs from above and modify it to use a keyword for a variable name. What is the response you get from the Java compiler.
2. Modify the `println` in the Hello World so that the variable `message` is misnamed as `Message` with a capital M. What is the response you get from the Java compiler.

2.2 char

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

Java organizes memory for char values so that each one is stored in 2 bytes of memory. 2 bytes of memory means that there can be as many as $65536(2^{16})$ individual characters.

2.3 boolean

The `boolean` type has two values: `true` and `false`. We will see that the boolean type is particularly important when we discuss logical expressions and control structures.

2.4 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, as shown in the following:

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

With respect to the last three rows in the above table:

- the multiplication operator in Java is the `*`.
- `/` is the division operator. If we divide one `int` by another the result is also an `int` - that is, there is no fractional component; any remainder is lost.
- `%` is the modulo operator which yields the remainder when the first operand is divided by the second operand.

Exercise: Write programs to determine what happens when a program:

- adds 1 to the largest value.
- subtracts 1 from the smallest value.
- divides a value by zero.

We will discuss other operators later in the text.

2.5 float, double

These are used to represent values that have decimal places. Just as we cannot write down the fraction $1/3$ completely (it is a repeating decimal 0.33333 etc.) there are fractions that cannot be represented in a computer. With limited space we are often storing just an approximation. One must be aware that round-off can occur and so this type should not be used in some cases: for example if your

program needs to represent monetary values. More about this much later on in the text. These 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:

data type	memory	minimum (approx)	maximum (approx)
float	4 bytes	$\pm 1.4 \times 10^{-45}$	$\pm 3.4 \times 10^{38}$
double	8 bytes	$\pm 4.9 \times 10^{-324}$	$\pm 1.79769 \times 10^{308}$

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

operator	example of use	example's result
+	7.1 + 1.1	8.2
-	12.1 - 5.0	7.1
*	2.2 * 2.2	4.84
/	10/4	2.5

When programmers use float or double they must be aware that some numbers are only approximated in memory. Consider the program below that calculates the difference between two values:

Listing 2.2: Approximations.java

```

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

Try running the above program and verify the output is as indicated.

If a data type for a value with decimal places is not given, the default is double; therefore, the calculations in the above example were carried out using double arithmetic.

2.6 Calculations in Java

Our previous examples of calculations were simple: two operands and one operator. If you require more complicated calculations then you must code the expression carefully. The notation for expressions used in Java is called infix: you type the first operand, then the operator, and then the second operand, as in `9.0 * 5.0`. Operations can be combined. For instance, suppose the variable `c` is of data type `double` and represents a value in degrees Celsius, then the formula for converting Celsius temperatures to Fahrenheit is coded as:

```
( (9.0 / 5.0) * c) - 32.0
```

There are several points to be made regarding the above expression:

- a value expressed with a decimal point is by default a `double` value;
- parentheses are used to create sub-expressions - sub-expressions are evaluated first before anything else;
- the order of operations can be critical: in this case the formula for converting Celsius to Fahrenheit requires the division to be performed before the multiplication and the subtraction must be done last;
- all operations in this example involve doubles.

The Java language evaluates an expression according to priorities (which can be overridden using sub-expressions). The highest priority operators are performed first and then the next highest, and so on.

2.6.1 Expressions

An expression is Java code which, when evaluated, yields a value. We will firstly consider expressions that involve literals, variables and operators.

The Assignment Operator

In terms of Java syntax, we previously referred to the following as an *assignment* statement.

```
i = 1;
```

This is actually a Java statement with the expression

```
i = 1
```

followed by a semi-colon. In the above, the equals (=) sign is an operator, and the expression

```
i = 1
```

is to be read "the value 1 is assigned to the variable *i*" - the value 1 is stored in the memory location belonging to *i*. The assignment operator has two operands. The operand to the left of the equals sign must be a variable; the operand to the right must be an expression. When the assignment is executed, the value of the expression (on the right of the assignment operator) will be stored in the memory location reserved for the variable.

2.6.2 Exercises

1. Modify the following program so the two print statements produce exactly the same results. For this to work, Java statements that interchange the values of x and y are required.

```

1  public class InterchangeVariableValues
2  {
3      public static void main(String[] args){
4          {
5              int x = 33, y = 222;
6              System.out.println(x+" "+y);
7              int temp;
8              temp = x;
9
10
11             // This next print statement is to produce exactly the
12             // same output (character-for-character) as the
13             // previous print statement
14             System.out.println(y+" "+x);
15         }
16     }
17 }

```

2. Modify question 1 so that the two values for x and y are obtained as input from the user.

Other Operators

Other operators we consider in this section are +, -, *, /, and %. Note that any example that uses % will be using `int` operands.

As with the equals operator these are used in an infix fashion: one operand to the left of the operator and one operand to the right of the operator. Some examples:

Operator Examples
netPay + overtimePay
grossPay - netPay
hours * 20.50
13 / 5
13 % 5

Complex expressions

Expressions can be more complicated; we can write more complex expressions involving these operators so long as each variable, literal, or sub-expression is separated from the next variable or literal by an operator. At least one space

must separate an operand and an operator. Consider these examples of Java expressions:

Java Expressions
9.0 / 5.0 * c - 32.0
f = 9.0 / 5.0 * c - 32.0
number / 10 * 10
grossPay - deductions * taxRate

Operator Priorities

When you examine the above expressions, in what order do you think the operations are performed? Are calculations performed in a left-to-right manner, or, are some operations performed before others?

Consider the last example

```
grossPay - deductions * taxRate
```

Suppose the intention of the programmer is to implement the business rule: taxes are calculated as net pay (gross pay minus deductions) multiplied by the tax rate.

This would not be the result as Java gives `*` a higher priority than `-`. By this we mean that, in Java, the multiplication would be computed first giving us the result of `deductions * taxRate`, the result of which would then be subtracted from `grossPay`. In situations like this a programmer will create a sub-expression enclosed in a pair of parentheses. To get the desired result above the programmer would code:

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

forcing the subtraction to be performed first and for that result to be multiplied by the tax rate. Sub-expressions are evaluated first, before the expression of which it a part.

The order in which expressions are evaluated is determined as indicated in the following table:

Order of Evaluation
Highest to Lowest
sub-expressions
* / %
+ -
=

Next we discuss a few examples to illustrate the evaluation of Java expressions.

Example 1

Consider the expression

```
13 % 5
```

We will only consider in this text the use of the modulo operator, %, where both operands are integers. When the operands are integers, the modulo operator returns the remainder when the first operand is divided by the second operand. So this example evaluates to 3, the remainder when 13 is divided by 5.

Exercise: What expression would yield the rightmost digit of an integer stored in the variable *number*?

Example 2

Now consider the expression

```
netPay = grossPay - deductions * taxRate
```

Of the operators in the expression, the multiplication operator has the highest priority and so it is performed first, the minus operation is performed next, and since assignment has the lowest priority it is performed last. We can show this order of evaluation using parenthesis where sub-expressions match what we have just stated:

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

Suppose the intention of the programmer was to implement a business rule where the taxes to be paid were to be calculated as the product of the tax rate and the difference between gross pay and deductions. According to the business rule, the default evaluation according to operator priorities is not what the programmer requires; to get the expression evaluated as per the business rule, we need to have:

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

Example 3

For the operators +, -, /, *, and %, when there is more than one operator of the same priority then they are carried out from **left to right**. Consider the expression

```
9.0 / 5.0 * c - 32.0
```

where the programmer intends that a value in degrees Celsius is converted to Fahrenheit using the well-known formula:

$$F = \frac{9}{5}C - 32$$

This expression has two operators of the same priority (/ and *). Since multiplication and division have the same priority, 9.0 is divided by 5.0 and that result is multiplied by c - the expression is evaluated as required to convert Celsius to Fahrenheit.

Example 4

If an expression has more than one assignment operator then those are carried out from **right to left**. Consider the expression

```
taxesPaid = netPay = grossPay - deductions * taxRate
```

and consider the priorities of the operators. Let us suppose grossPay is 100.00, deductions is 10.00, and that taxRate is 0.10.

The values of netPay and taxesPaid will be the same, 99.00. The expression is evaluated according to the sequence:

- deductions is multiplied by taxRate yielding a value of 1.00 ;
- 1.00 is subtracted from grossPay yielding a value of 99.00;
- 99.00 is assigned to netPay;
- the value of netPay is assigned to taxesPaid and so the value of taxesPaid becomes 99.00

Example 5

Recall from Example 4, if an expression has more than one assignment operator then those are carried out from **right to left**. Now consider a similar expression but where we have a sub-expression:

```
taxesPaid = (netPay = grossPay - deductions) * taxRate
```

Consider the priorities of the operators and the use of the subexpression to override operator priorities. Again, let us suppose grossPay is 100.00, deductions is 10.00, and that taxRate is 0.10.

The values of netPay and taxesPaid will be different. The expression is evaluated according to the sequence:

- deductions is subtracted from grossPay yielding a value of 90.00;
- this result, 90.00, is assigned to the variable netPay;
- 90.00 (the value of the sub-expression) is multiplied by 0.10 yielding a value of 9.00;
- this result, 9.00, is assigned to taxesPaid;

If the intention of the programmer was to implement two business rules where (1) net pay is determined as gross pay minus deductions, and (2) the taxes to be paid were to be calculated as the product of the tax rate and net pay, then the above expression implements both rules.

2.6.3 Exercises

The last two exercises refer to programs available at the website for this text. These two programs could be modified to assist in answering questions 1, 2, and 3.

1. Use the expression in Example 1 in a program.
2. Use the expressions in Example 2 in a program.
3. Use the expression in Example 3 in a program. You will need to declare `c` to be a double and assign it a value.
4. Run the `TaxesPaid` program and verify the output from Example 4. Note in the program that the expression is followed by a semi-colon to be a proper Java statement.
5. Run the `TaxesPaidCalculatedCorrectly` program and verify the output is as stated in Example 5. Note in the program that the expression is followed by a semi-colon to be a proper Java statement.

2.6.4 Mixed Mode Expressions

Expressions could contain a mixture of types. Java permits conversions between integer and floating-point types, and also allows characters to be converted to integer and floating-point types. Every character corresponds to a number.

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). So, the following will involve an automatic conversion of a `short` to an `int`.

Listing 2.3: 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         s = 100;
11         int t;
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 char to int, 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, and 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 minus 32.0. For this to be performed the 1 is converted to 1.0 and the final result is -31.0. Note that this would be considered inaccurate (wrong) for the conversion of 1 degree Celsius to Fahrenheit. To get a more accurate result the expression should have involved 9.0/5.0.

Narrowing conversions are cases where there could be a loss of precision going from one type to another. For example converting an double to a char is not allowed unless the programmer directly indicates that casting from one type to another is to be performed. We will leave the `cast` language construct until a later chapter.

2.6.5 Unary Operators

At this point we introduce the unary minus. A unary minus, -, can be placed immediately in front of an expression to negate the value of the expression. For instance the value of

```
-(1000 / 10)
```

is -100.

2.6.6 Exercises

1. The expressions

```
(9/5) * 30 - 32.0
```

and

```
(9.0/5.0) * 30 - 32.0
```

produce different results. Write a program to show the different results.

2. Write a program that will print the integer values for the characters 'a', 'b', 'c', 'A', 'B', 'C', '1', '2', '3'. Note that you can use a statement such as

```
int value = 'a';
```

3. Change the program ShortToInt so that it assigns the value of an `int` to a `short`. What happens when you try to compile the program?

4. What values are produced by the following expressions:

```
99 / 10
```

```
99 / 10 * 10
```

```
99 % 10
```

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

2.7 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:

```
"this is a line of text"
```

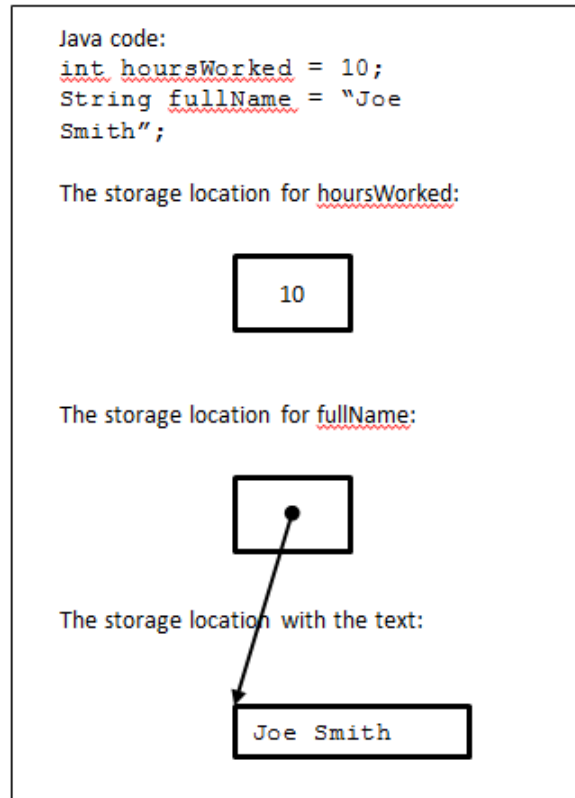
```
"my first name is Joe"
```

```
"515 Portage Avenue"
```

Because text strings are used so often Java provides a "short-cut" for assigning values to String variables without the need for using the new keyword:

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

Recall that String was not mentioned in the section on primitive data types. The assignment statement above causes an object to be created and a reference to that object is stored in the variable `fullName`. There is a subtle difference that is hard to appreciate at this time: the variable holds a reference to the value instead of holding the actual value. The diagram below attempts to show the difference.



Because strings are objects defined as `String` another way to declare `fullName` and assign it a value is:

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

The `String` class provides many methods for working with text strings such as:

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

The following program illustrates various methods.

Listing 2.4: Using String methods for input

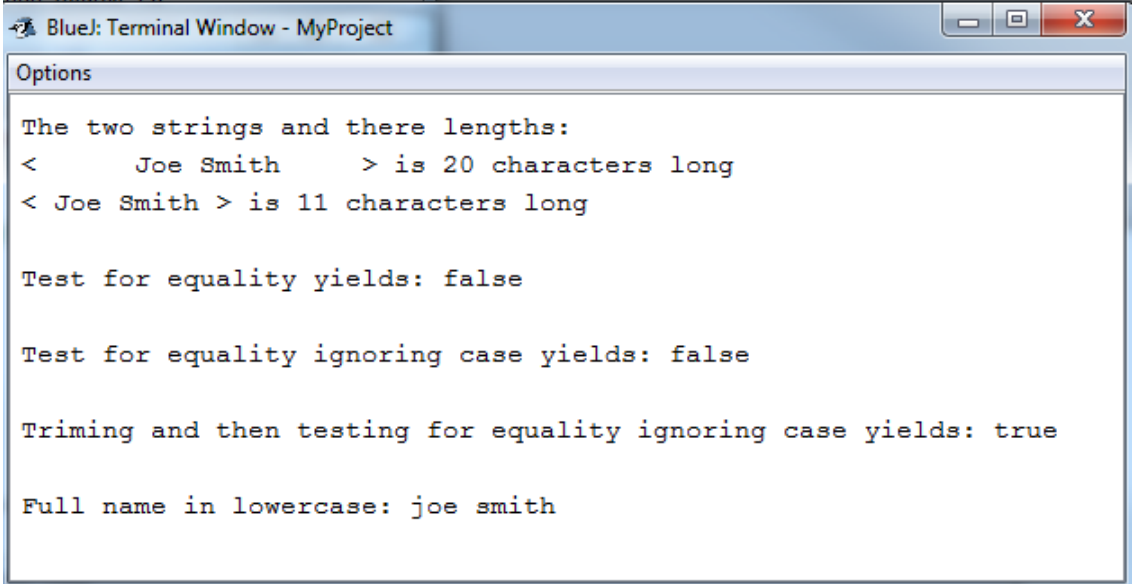
```

1 public class UsingStringMethods
2 {
3     public static void main(String[] args)
4     {
5         // a string with leading and trailing spaces
6         String fullName = "   Joe Smith   ";
7         // same name fewer leading and trailing spaces
8         String fullNameLc = " Joe Smith ";
9
10        // Display the lengths of the two strings
11        System.out.println("The two strings and there lengths:"
12            + "\n<" + fullName + "> is " + fullName.length() + " characters long"
13            + "\n<" + fullNameLc + "> is " + fullNameLc.length() + " characters
14                long"
15            );
16
17        // compare the two strings for equality
18        boolean namesAreEqual = fullName.equals(fullNameLc);
19        System.out.println("\nTest for equality yields: " + namesAreEqual);
20
21        // compare the two strings for equality ignoring case
22        namesAreEqual = fullName.equalsIgnoreCase(fullNameLc);
23        System.out.println("\nTest for equality ignoring case yields:
24            " + namesAreEqual);
25
26        // trim both and then compare the two strings for equality
27        // ignoring case
28        fullName = fullName.trim();

```

```
26     fullNameLc = fullNameLc.trim();
27     namesAreEqual = fullName.equalsIgnoreCase(fullNameLc);
28     System.out.println("\nTrimming and then testing for equality
        ignoring case yields: "+namesAreEqual);
29
30     // display fullName in lowercase
31     fullName = fullName.toLowerCase();
32     System.out.println("\nFull name in lowercase: "+fullName);
33
34 }
35 }
```

The output is:

A screenshot of a Java IDE terminal window titled "BlueJ: Terminal Window - MyProject". The window has a standard Windows-style title bar with minimize, maximize, and close buttons. Below the title bar is a tab labeled "Options". The main area of the window displays the output of the Java program in a monospaced font. The output text is as follows:

```
The two strings and there lengths:
<      Joe Smith      > is 20 characters long
< Joe Smith > is 11 characters long

Test for equality yields: false

Test for equality ignoring case yields: false

Triming and then testing for equality ignoring case yields: true

Full name in lowercase: joe smith
```

2.7.1 Catenation operator +

The + operator is a binary operator that is used frequently in statements that generate output. If the two operands of + are strings then the two strings are concatenated together forming one larger string as a result. If one operand is not a string then the equivalent string representing its value is generated, and then the catenation of two strings is carried out forming a new string as a result.

2.7.2 Exercises

1. 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 == 100)
```

2. 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 in the format:

```
<last name> <a comma> <first name> <space> <initials>
```

for example: Smith, John A

2.8 Output

2.8.1 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 catenations. Notice the last `println()` introduces some special characters for new line and tabbing. The special characters are not displayed, they are used to control the appearance of the output.

Listing 2.5: Using `println()`

```

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

```

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
  
```

`println()` advances to a new line and then displays output. `print()` 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:

```

Options
100.0Gross Pay is 100.0Gross Pay = 100.0 Deductions = 100.0    Gross Pay = 100.0
    Deductions = 100.0
    Net Pay = 9.0
  
```

2.8.2 JOptionPane

In some situations a programmer may find `JOptionPane` message dialogs useful. The following program shows how to display some information. 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 find details of the `Scanner` class.

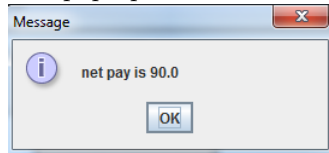
Listing 2.6: Using `println()`

```

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 }
```

The pop-up window the user showing the information and an OK button:



2.9 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.

2.9.1 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 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 <code>int</code> in the input stream
<code>nextDouble()</code> returns the next <code>double</code> in the input stream
<code>nextBoolean()</code> returns the next <code>boolean</code> 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.

Listing 2.7: Using `JOptionPane` for input

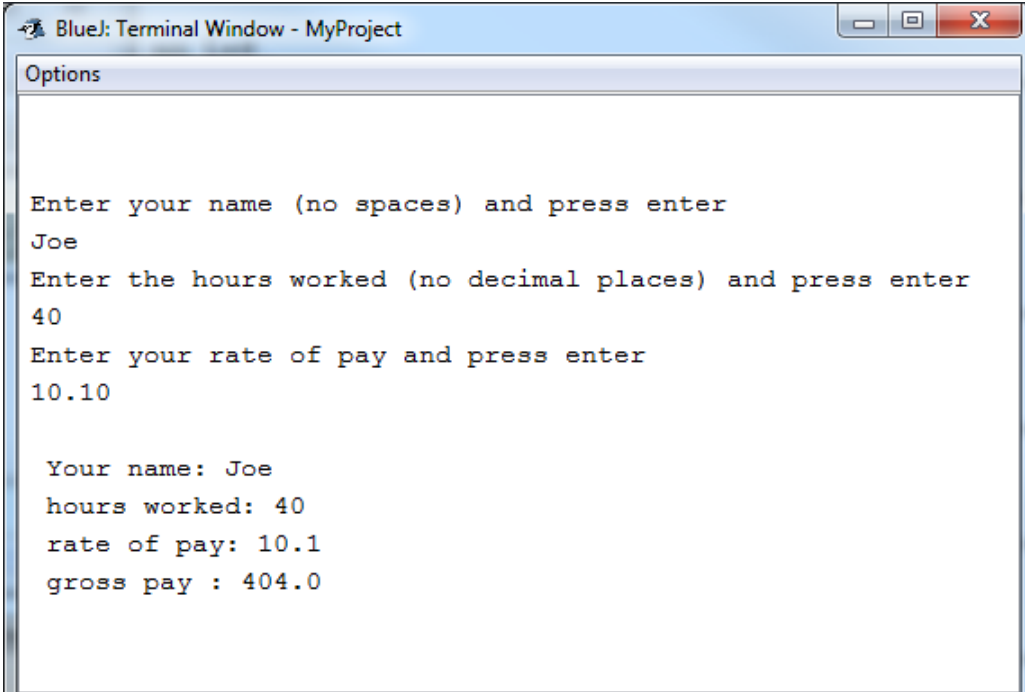
```

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 }
```

The above program was run, and the contents of the Terminal Window are shown below. This window shows the output/prompts from the program and the input provided by the user via the keyboard.



```
BlueJ: Terminal Window - MyProject

Options

Enter your name (no spaces) 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
```

2.9.2 The `JOptionPane` Class

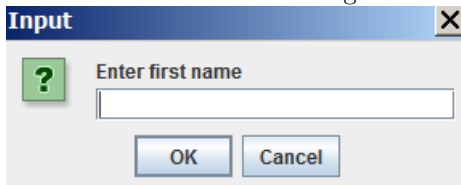
Another technique a programmer can use is in the form of dialog boxes. The programmer can use the Java pre-defined class `JOptionPane` to obtain String input from the user (the person running the program).

Consider the following program that obtains 2 values from the user.

Listing 2.8: 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 = JOptionPane.showInputDialog("Enter first
10             name");
11             String lastName = JOptionPane.showInputDialog("Enter first
12             name");
13             System.out.println ("Your name is: "+firstName+" "+lastName);
14     }
```

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 (see below). The user is able to enter a value in the box and press OK. Then control goes back to the program and the value entered is assigned to `firstName`. A similar dialog box is displayed when line 10 executes.



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. These three structures can be combined. Java has one sequence structure, two (some may say more) different decision structures, and several ways of coding loops. We will begin with the [compound](#), the [while](#) and the [if](#) statements.

3.1 [Compound](#) statements

Java statements delimited by curly braces { and } form a [compound](#) statement. { and } always appear as a pair. The { is first, followed by }. Anytime 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 pair can be entirely enclosed in some other pair.

In the example that follows you will see two compound statements: one forms the statement to be executed when x is greater than y, and the other forms the statement to be executed when x is not greater than y. Note the [while](#) and [if](#) statements are covered next.

We will see later that [compound](#) statements are a necessary component of many decision and loop structures.

Listing 3.1: Program with 2 compound statements

```
1 public class DisplayLargest
2 {
3     public static void main(String[] args)
4     {
5         int largest, smallest;
6         int x = 100;
7         int y = 500;
8         if (x > y) {
9             largest = x;
10            smallest = y;
11        }
12        else {
13            largest = y;
14            smallest = x;
15        }
16        System.out.println("x is "+x+" and y is "+y);
17        System.out.println("the largest is "+largest);
18        System.out.println("the smallest is "+smallest);
19    }
20 }
```

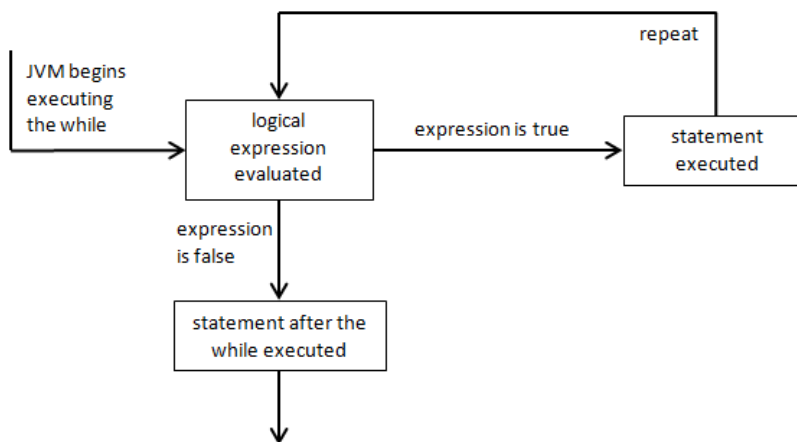
3.2 The `while` Statement

The JVM executes the statements in a program sequentially, one statement after another. However, the `while` statement can change this. Embedded in a `while` is another statement (often a compound statement) to be executed repeatedly as long as some logical expression is true. 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. The relational operators are `<`, `<=`, `>=`, and `>`:

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

The equality operators are `==` and `!=`. Later we will discuss boolean operators that are used to combine logical expressions.

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

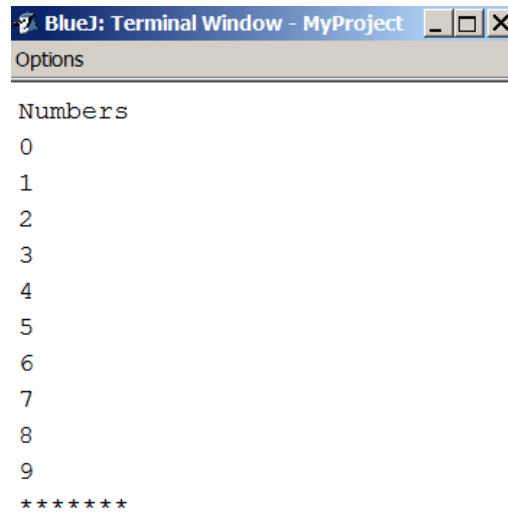
Example 1

Consider the following program that prints numbers from 0 to 9:

Listing 3.2: 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 form a compound statement (enclosed in curly braces). 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 value of 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:



```
BlueJ: Terminal Window - MyProject
Options
Numbers
0
1
2
3
4
5
6
7
8
9
*****
```

Figure 3.1: Output from Numbers0To9

Example 2

Consider another program which displays the digits of a positive number that the user provides as input. This program uses a scanner object in order to get input from the user via the keyboard.

Listing 3.3: Display digits

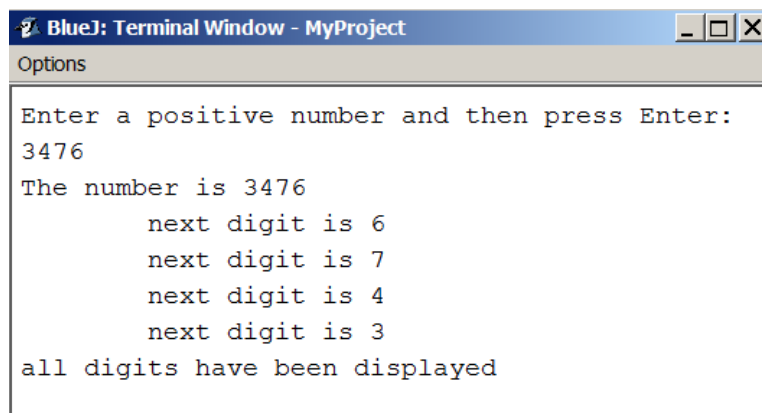
```
1 import java.util.Scanner;
2 public class DisplayDigits
3 {
4     public static void main(String[] args)
5     {
6         // Arrange to use a scanner object for keyboard input
7         Scanner keyboard = new Scanner(System.in);
8         // Prompt the user for a positive number
9         System.out.println("Enter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

This program has a while loop where the number obtained from the user is altered each time the loop executes. Line 16 contains the expression

```
number / 10
```

The value of this expression is assigned to variable *number*, and so eventually the value stored in *number* will be reduced to 0 and the loop terminates. Note in line 14 how the program obtains the rightmost digit via the modulo operator. A sample of output is:

Output from DisplayDigits



```
BlueJ: Terminal Window - MyProject
Options
Enter a positive number and then press Enter:
3476
The number is 3476
    next digit is 6
    next digit is 7
    next digit is 4
    next digit is 3
all digits have been displayed
```

3.2.1 Exercises

1. Write a program that will sum the digits from -100 to 100.
2. What happens when a user enters the value 0 when DisplayDigits is executed?
3. What happens when a user enters a negative value when DisplayDigits is executed?
4. What happens when a user enters something that is not an integer when DisplayDigits is executed?
5. Write a program that converts from Celsius to Fahrenheit for Celsius values starting at -40 and going up +40 in increments of 1.
6. Write a program that converts from Celsius to Celsius for Fahrenheit values starting at -40 and going up +40 in increments of 1.

7. 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.
8. 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 $7 = 1+2+4$
9. 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.
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 .

3.2.2 Nesting statements

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

Example 3

consider the program:

Listing 3.4: Nesting one while inside another while

```
1  /**
2   * Program with one while inside another while
3   * Outer while controls the value of variable i
4   * Inner while controls the value of variable j
5   * Inner while is executed once for each value of i
6   */
7  public class NestedWhiles
8  {
9      public static void main(String[] args)
10     {
11         int i, j;
12         System.out.println("i j");
13         // i takes on values 0, 1
14         i = 0;
15         while (i < 2){
16             j = 0;
17             // j takes on values 0, 1, 2
18             while (j < 3){
19                 System.out.println(i+" "+j);
20                 j = j + 1;
21             }
22             i = i + 1;
23         }
24         System.out.println("***");
25     }
26 }
27 }
```

The above program has two variables *i* and *j*. The outer while (lines 15-23) is executed twice, once with *i* equal to 0 and then with *i* equal to 1. The inner while (lines 18-21) is executed once for each value of *i*; for each value of *i*, the variable *j* takes on the values 0, 1, and 2. Study the program to verify the output shown next. How many times is the print statement (line 19) executed? Notice the indentation of lines 18-21; this is done simply to help a human read the code - one quickly sees that those lines are a control structure that is embedded inside another control structure.

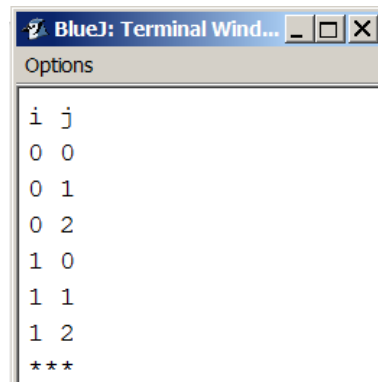


Figure 3.2: Output from NestedWhiles

3.2.3 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. See the section on Unary Operators for more information.

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--;
```

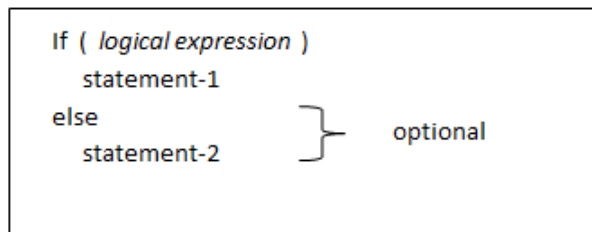
3.2.4 Exercises

1. Modify the programs in this section to use the `++` operator where it can be applied.
2. Use nested whiles to print a 4×4 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

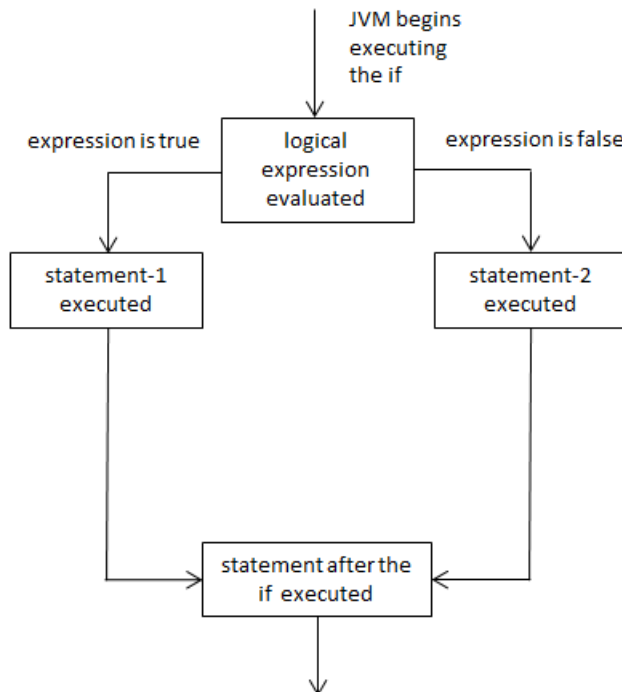
3.3 The **if** Statement

The structure of an **if** statement is as follows (note that the **else** and *statement-2* are optional. We say the if statement has an optional else 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

Consider the following program that displays a different message depending on the value of the expression *number > 0*. Note that compound statements are used even though it was not necessary - some programmers always code compound statements. Exactly one of the two compound statements will be executed.

Listing 3.5: Using an if statement

```
1 public class PositiveOrNot
2 {
3     public static void main(String[] args)
4     {
5         // Display a message if number is positive or not
6         int number = 11;
7         if (number > 0) {
8             System.out.println("the number "+number+" is positive");
9         }
10        else {
11            System.out.println("the number "+number+" is not positive");
12        }
13    }
14 }
```

3.3.1 Exercises

1. Modify PositiveOrNot so that it obtains a number from the user. Use a Scanner object so that you can get the number as an integer.
2. Write a program that obtains a number from the user and displays whether the number is an even number or an odd number. Consider using the % operator.
3. Write a program that obtains two numbers from the user and displays the larger of the two numbers.

3.3.2 Nesting statements

The syntax of the if statement provides for the conditional execution of any Java statement, including other if statements, whiles, etc. When the conditionally executed statement is an if statement then we are nesting an if inside another if.

Example 2

Consider where a customer is paying for some goods with either cash or a debit card. Suppose there are no pennies in circulation and if someone is paying with cash the cost is rounded to the nearest nickel. For instance if the cost is \$10.22 then the cost is rounded down to \$10.20, and if the cost is \$10.23 the cost is rounded up to \$10.25. Suppose also that a debit card payment has a surcharge of 25 cents. Consider the following program where the program prompts for the type and cost of a purchase, and handles a cash or debit purchase appropriately. To avoid rounding errors the program uses integers for the cost and so the cost is processed in cents (not as a double where a decimal point separates dollars and cents).

Listing 3.6: Using an if statement

```

1  import java.util.Scanner;
2  /**
3   * Determine value of payment to be received from customer
4   * based on whether or not it is cash payment.
5   * Cash payments are rounded off to the nearest nickel and
6   * debit card payments have a surcharge of 25 cents.
7   */
8  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 and "
15             +"value of purchase in pennies: ");
16         Scanner kb = new Scanner(System.in);
17         typePayment = kb.next();
18         originalCost = kb.nextInt();
19         if (typePayment.equals("cash")) {
20             if (originalCost % 5 < 3)
21                 actualCost = originalCost - originalCost%5;
22             else
23                 actualCost = originalCost + (5 - originalCost%5);
24         }
25         else
26             actualCost = originalCost + 25;
27         System.out.println(originalCost+" "+actualCost);

```


28 }
29 }

30 }

Observe the indentation in the listing above. When each logical expression is basically the same with one simple change (the value A, B, ...) a Java programmer will change the indentation to that shown below, and may then refer to 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 grade:");
14         Scanner kb = new Scanner(System.in);
15         letterGrade = kb.next();
16         if (letterGrade.equals("A"))
17             numericGrade = 4.0;
18         else if (letterGrade.equals("B"))
19             numericGrade = 3.0;
20         else if (letterGrade.equals("C"))
21             numericGrade = 2.0;
22         else if (letterGrade.equals("D"))
23             numericGrade = 1.0;
24         else
25             numericGrade = 0.0;
26         System.out.println(letterGrade+" is equivalent to
27                             "+numericGrade);
28     }
29 }
```

3.3.3 Exercises

1. 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 determine which range it falls into and determining 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 > ... )
```

2. 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 $[0, 100]$.
3. 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 The `for` Statement

3.5 The `do while` Statement

3.6 The `switch` Statement

3.7 Logical Expressions

A logical expression is an expression that evaluates to a boolean value, i.e. `true` or `false`. Java has several comparison operators that produce a value of `true` or `false`. The two types of operators we consider here are the relational and equality operators. The relational operators are `<`, `<=`, `>=`, and `>`:

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

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>

Logical expressions can be combined using boolean operators which are `&&`, `||`, and `!`:

Boolean operators		
operator	meaning	example
<code>&&</code>	AND	<code>count > 0 && count < 100</code>
<code> </code>	OR	<code>netPay > 100 grossPay > 150</code>
<code>!</code>	NOT	<code>! (number >= 0)</code>

Truth tables

The values produced by the boolean operators can be illustrated using truth tables. A truth table shows all possible combinations of operand(s) and the result of an operation for each combination.

`&&` is `true` only when both operands are `true`:

AND		
operand-1	operand-2	result
true	true	true
true	false	false
false	true	false
false	false	false

`||` is `true` when either one (or both) of the operands is `true`:

OR		
operand-1	operand-2	result
true	true	true
true	false	true
false	true	true
false	false	false

! simply negates its operand:

NOT	
operand	result
true	false
false	true

Operator priority

The table below shows the priorities from highest to lowest for the arithmetic, logical, and boolean operators.

Order of Evaluation Highest to Lowest	
sub-expressions	(...)
autoincrement/decrement	++, --
unary	-, !
arithmetic	*, /
arithmetic	+ - (including string catenation)
assignment	=
relational	< > <= >=
equality	== !=
boolean	&&
boolean	

3.7.1 Examples

In the following consider that x and y are [ints](#) and are both equal to 100 for each example.

Example 1

Consider

```
x+3 > 100
```

Since addition is performed before > this expression is equivalent to

```
103 > 100
```

which evaluates to [true](#).

Example 2

Consider

```
x++ > 100 && y++ <= 101
```

Since autoincrement is performed before relational operations this expression is equivalent to

```
101 > 100 && 101 <= 101
```

which evaluates to **true**.

Example 3

Consider

```
x + y > 150 && y = 100
```

The arithmetic operations are performed first followed by the relational operations and so the above is equivalent to

```
false && true
```

which evaluates to **false**. However, in a case like this where two expressions are ANDed, the JVM would evaluate the first operand of **&&** and since that is **false** the JVM would not evaluate a second operand since a **false** ANDed to anything yields false. Similarly the JVM does not evaluate the second operand for **||** when the first operand is **true**.

Example 4

Consider that

```
boolean found = true
```

and we have the expression

```
! found && -x < 100
```

The unary operation would be performed first and so the above is equivalent to

```
false && -x < 100
```

which evaluates to **false**. Again, the JVM would not evaluate the second operand of **&&** since the first operand is **false**.

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/von_neumann_rl.pdf.