

JAVA FOR BEGINNERS



CURATED BY @SAMEER RAZA

Contents

Introduction.....	5
About JAVA	5
OOP – Object Oriented Programming	5
Part 1 - Getting Started.....	6
The Java Development Kit – JDK	6
My first Java program.....	6
Using an IDE	7
Variables and Data Types	8
Variables.....	8
☛ Test your skills – Example3.....	8
Mathematical Operators	9
Logical Operators	9
Character Escape Codes	11
Test your skills – Example7.....	12
Data Types.....	13
Introducing Control Statements.....	16
Blocks of Code	18
Test your skills – Example14	18
The Math Class	19
Scope and Lifetime of Variables	20
Type Casting and Conversions.....	21
Console Input	24
Using the Keyboard Class	24
Using the Scanner Class	33
Using Swing Components	34
Part 2 - Advanced Java Programming	35
Control Statements - The if Statement.....	35
Guessing Game (Guess.java)	36
Nested if	37
Guessing Game v.3	37
if-else-if Ladder	38
Ternary (?) Operator.....	39
switch Statement (case of)	41
Nested switch	45
Mini-Project – Java Help System (Help.java)	45
Complete Listing	46

The for Loop.....	48
Multiple Loop Control Variable	50
Terminating a loop via user intervention	50
Interesting For Loop Variations	51
Infinite Loops.....	52
No 'Body' Loops.....	52
Declaring variables inside the loop	52
Enhanced For loop	53
The While Loop	54
The do-while Loop.....	55
Mini-Project 2– Java Help System (Help2.java)	58
Complete listing	59
Using Break to Terminate a Loop	62
Terminating a loop with break and use labels to carry on execution	63
Use of Continue (complement of Break)	66
Continue + Label.....	67
Mini-Project 3– Java Help System (Help3.java)	68
Complete Listing	68
Nested Loops	71
Class Fundamentals	72
Definition	72
The Vehicle Class	72
Using the <i>Vehicle</i> class	73
Creating more than one instance	73
Creating Objects	74
Reference Variables and Assignment	74
Methods	75
Returning from a Method	76
Returning a Value	77
Methods which accept Parameters:	79
Project: Creating a Help class from the Help3.java	83
Method <code>helpon()</code>	83
Method <code>showmenu()</code>	84
Method <code>isvalid()</code>	85
Class Help	85
Main Program:	87
Constructors	88

Constructor having parameters	89
Overloading Methods and Constructors	90
Method Overloading	90
Automatic Type Conversion for Parameters of overloaded Methods	92
Overloading Constructors	94
Access Specifiers: public and private	96
Arrays and Strings	101
Arrays	101
One-dimensional Arrays	101
Sorting an Array – The Bubble Sort	103
Two-Dimensional Arrays:	104
Different syntax used to declare arrays:	105
Array References:	106
The Length Variable:	107
Using Arrays to create a Queue data structure **	110
The Enhanced 'for' Loop:	113
Strings	114
Using String Methods	115
String Arrays	117
Vector and ArrayList	122
Employee.java	125
ComparableDemo.java	126
File Operations in Java	134
Template to read data from disk	138
Template to write (save) data to disk	142
Introduction to GUI using AWT/Swing	143
Using Swing to create a small Window	143
Inserting Text inside Window	144
Creating a simple application implementing JButton, JTextfield and JLabel	145

Introduction

About JAVA

“Java refers to a number of computer software products and specifications from Sun Microsystems (the Java™ technology) that together provide a system for developing and deploying cross-platform applications. Java is used in a wide variety of computing platforms spanning from embedded devices and mobile phones on the low end to enterprise servers and super computers on the high end. Java is fairly ubiquitous in mobile phones, Web servers and enterprise applications, and somewhat less common in desktop applications, though users may have come across Java applets when browsing the Web.

Writing in the Java programming language is the primary way to produce code that will be deployed as Java bytecode, though there are compilers available for other languages such as JavaScript, Python and Ruby, and a native Java scripting language called Groovy. Java syntax borrows heavily from C and C++ but it eliminates certain low-level constructs such as pointers and has a very simple memory model where every object is allocated on the heap and all variables of object types are references. Memory management is handled through integrated automatic garbage collection performed by the Java Virtual Machine (JVM).”¹

OOP – Object Oriented Programming

OOP is a particular style of programming which involves a particular way of designing solutions to particular problems. Most modern programming languages, including Java, support this paradigm. When speaking about OOP one has to mention:

- Inheritance
- Modularity
- Polymorphism
- Encapsulation (binding code and its data)

However at this point it is too early to try to fully understand these concepts.

This guide is divided into two major sections, the first section is an introduction to the language and illustrates various examples of code while the second part goes into more detail.

¹ http://en.wikipedia.org/wiki/Java_%28Sun%29

Part 1 - Getting Started

The Java Development Kit - JDK

In order to get started in Java programming, one needs to get a recent copy of the Java JDK. This can be obtained for free by downloading it from the Sun Microsystems website, <http://java.sun.com/>

Once you download and install this JDK you are ready to get started. You need a text editor as well and Microsoft's Notepad (standard with all Windows versions) suits fine.

My first Java program

Open your text editor and type the following lines of code:

```
/*
My first program
Version 1
*/

public class Example1 {
    public static void main (String args []) {
        System.out.println ("My first Java program");
    }
}
```

This is known as a Block Comment.
These lines are useful to the programmer and are ignored by the Compiler

Save the file as Example1.java². The name of the program has to be similar to the filename. Programs are called classes. Please note that Java is **case-sensitive**. You cannot name a file "Example.java" and then in the program you write "public class example". It is good practice to insert comments at the start of a program to help you as a programmer understand quickly what the particular program is all about. This is done by typing "/*" at the start of the comment and "*/" when you finish. The predicted output of this program is:

```
My first Java program
```

In order to get the above output we have to first compile the program and then execute the compiled class. The applications required for this job are available as part of the JDK:

- javac.exe – compiles the program
- java.exe – the interpreter used to execute the compiled program

In order to compile and execute the program we need to switch to the command prompt. On windows systems this can be done by clicking Start>Run>cmd

² Ideally you should create a folder on the root disk (c:\) and save the file there

At this point one needs some basic DOS commands in order to get to the directory (folder), where the java class resides:

- `cd\` (change directory)
- `cd\[folder name]` to get to the required folder/directory

When you get to the required destination you need to type the following:

```
c:\[folder name]\javac Example1.java
```

The above command will compile the java file and prompt the user with any errors. If the compilation is successful a new file containing the bytecode is generated: **Example1.class**

To execute the program, we invoke the interpreter by typing:

```
c:\[folder name]\java Example1
```

The result will be displayed in the DOS window.

Using an IDE

Some of you might already be frustrated by this point. However there is still hope as one can forget about the command prompt and use an IDE (integrated development environment) to work with Java programming. There are a number of IDE's present, all of them are fine but perhaps some are easier to work with than others. It depends on the user's level of programming and tastes! The following is a list of some of the IDE's available:

- BlueJ – www.bluej.org (freeware)
- NetBeans – www.netbeans.org (freeware/open-source)
- JCreator – www.jcreator.com (freeware version available, pro version purchase required)
- Eclipse – www.eclipse.org (freeware/open-source)
- IntelliJ IDEA – www.jetbrains.com (trial/purchase required)
- JBuilder – www.borland.com (trial/purchase required)

Beginners might enjoy BlueJ and then move onto other IDE's like JCreator, NetBeans, etc. Again it's just a matter of the user's tastes and software development area.

Variables and Data Types

Variables

A variable is a place where the program stores data temporarily. As the name implies the value stored in such a location can be changed while a program is executing (compare with constant).

```
class Example2 {  
    public static void main(String args[]) {  
        int var1; // this declares a variable  
        int var2; // this declares another variable  
        var1 = 1024; // this assigns 1024 to var1  
        System.out.println("var1 contains " + var1);  
        var2 = var1 / 2;  
        System.out.print("var2 contains var1 / 2: ");  
        System.out.println(var2);  
    }  
}
```

Predicted Output:

```
var2 contains var1 / 2: 512
```

The above program uses two variables, var1 and var2. var1 is assigned a value directly while var2 is filled up with the result of dividing var1 by 2, i.e. $\text{var2} = \text{var1}/2$. The words *int* refer to a particular data type, i.e. integer (whole numbers).

Test your skills – Example3

As we saw above, we used the '/' to work out the quotient of var1 by 2. Given that '+' would perform addition, '-' subtraction and '*' multiplication, write out a program which performs all the named operations by using two integer values which are hard coded into the program.

Hints:

- You need only two variables of type integer
- Make one variable larger and divisible by the other
- You can perform the required calculations directly in the print statements, remember to enclose the operation within brackets, e.g. (var1-var2)

Mathematical Operators

As we saw in the preceding example there are particular symbols used to represent operators when performing calculations:

Operator	Description	Example – given a is 15 and b is 6
+	Addition	a + b, would return 21
-	Subtraction	a - b, would return 9
*	Multiplication	a * b, would return 90
/	Division	a / b, would return 2
%	Modulus	a % b, would return 3 (the remainder)

```
class Example4 {
    public static void main(String args[]) {
        int iresult, irem;
        double dresult, drem;
        iresult = 10 / 3;
        irem = 10 % 3;
        dresult = 10.0 / 3.0;
        drem = 10.0 % 3.0;
        System.out.println("Result and remainder of 10 / 3: " +
            iresult + " " + irem);
        System.out.println("Result and remainder of 10.0 / 3.0: "
            + dresult + " " + drem);
    }
}
```

Predicted Output:

```
Result and Remainder of 10/3: 3 1
Result and Remainder of 10.0/3.0: 3.3333333333333335 1
```

The difference in range is due to the data type since 'double' is a double precision 64-bit floating point value.

Logical Operators

These operators are used to evaluate an expression and depending on the operator used, a particular output is obtained. In this case the operands must be Boolean data types and the result is also Boolean. The following table shows the available logical operators:

Operator	Description
&	AND gate behaviour (0,0,0,1)
	OR gate behaviour (0,1,1,1)
^	XOR – exclusive OR (0,1,1,0)
&&	Short-circuit AND
	Short-circuit OR
!	Not

```

class Example5 {
    public static void main(String args[]) {
        int n, d;

        n = 10;

        d = 2;

        if(d != 0 && (n % d) == 0)
            System.out.println(d + " is a factor of " + n);

        d = 0; // now, set d to zero

        // Since d is zero, the second operand is not evaluated.

        if(d != 0 && (n % d) == 0)
            System.out.println(d + " is a factor of " + n);

        /* Now, try same thing without short-circuit operator.
        This will cause a divide-by-zero error.
        */

        if(d != 0 & (n % d) == 0)
            System.out.println(d + " is a factor of " + n);

        }
    }
}

```

Predicted Output:

*Note if you try to execute the above program you will get an error (division by zero). To be able to execute it, first comment the last two statements, compile and then execute.

2 is a factor of 10

Trying to understand the above program is a bit difficult, however the program highlights the main difference in operation between a normal AND (&) and the short-circuit version (&&). In a normal AND operation, both sides of the expression are evaluated, e.g.

if(d != 0 & (n % d) == 0) – this returns an error as first d is compared to 0 to check inequality and then the operation (n%d) is computed yielding an error! (divide by zero error)

The short circuit version is smarter since if the left hand side of the expression is false, this means that the output has to be false whatever there is on the right hand side of the expression, therefore:

if(d != 0 && (n % d) == 0) – this does not return an error as the (n%d) is not computed since d is equal to 0, and so the operation (d!=0) returns false, causing the output to be false. Same applies for the short circuit version of the OR.

Character Escape Codes

The following codes are used to represent codes or characters which cannot be directly accessible through a keyboard:

Code	Description
\n	New Line
\t	Tab
\b	Backspace
\r	Carriage Return
\\	Backslash
\'	Single Quotation Mark
\"	Double Quotation Mark
*	Octal - * represents a number or Hex digit
\x*	Hex
\u*	Unicode, e.g. \u2122 = ™ (trademark symbol)

```
class Example6 {
    public static void main(String args[]) {
        System.out.println("First line\nSecond line");
        System.out.println("A\tB\tC");
        System.out.println("D\tE\tF") ;
    }
}
```

Predicted Output:

```
First Line
Second Line
A      B      C
D      E      F
```

🌟 Test your skills – Example7

Make a program which creates a sort of truth table to show the behaviour of all the logical operators mentioned. Hints:

- You need two Boolean type variables which you will initially set both to false
- Use character escape codes to tabulate the results

The following program can be used as a guide:

```
class LogicTable {
    public static void main(String args[]) {
        boolean p, q;

        System.out.println("P\tQ\tPANDQ\tPORQ\tPXORQ\tNOTP");

        p = true; q = true;

        System.out.print(p + "\t" + q + "\t");
        System.out.print((p&q) + "\t" + (p|q) + "\t");
        System.out.println((p^q) + "\t" + (!p));

        p = true; q = false;

        System.out.print(p + "\t" + q + "\t");
        System.out.print((p&q) + "\t" + (p|q) + "\t");
        System.out.println((p^q) + "\t" + (!p));

        p = false; q = true;

        System.out.print(p + "\t" + q + "\t");
        System.out.print((p&q) + "\t" + (p|q) + "\t");
        System.out.println((p^q) + "\t" + (!p));

        p = false; q = false;

        System.out.print(p + "\t" + q + "\t");
        System.out.print((p&q) + "\t" + (p|q) + "\t");
        System.out.println((p^q) + "\t" + (!p));

    }
}
```

Predicted Output:

P	Q	PANDQ	PORQ	PXORQ	NOTP
true	true	true	true	false	fals
true	false	false	true	true	fals
false	true	false	true	true	true
false	false	false	false	false	true

Data Types

The following is a list of Java's primitive data types:

Data Type	Description
int	Integer – 32bit ranging from -2,147,483,648 to 2,147,483,648
byte	8-bit integer ranging from -128 to 127
short	16-bit integer ranging from -32,768 to 32,768
long	64-bit integer from -9,223,372,036,854,775,808 to -9,223,372,036,854,775,808
float	Single-precision floating point, 32-bit
double	Double-precision floating point, 64-bit
char	Character , 16-bit unsigned ranging from 0 to 65,536 (Unicode)
boolean	Can be true or false only

The 'String' type has not been left out by mistake. It is not a primitive data type, but strings (a sequence of characters) in Java are treated as Objects.

```
class Example8 {
    public static void main(String args[]) {
        int var; // this declares an int variable
        double x; // this declares a floating-point variable
        var = 10; // assign var the value 10
        x = 10.0; // assign x the value 10.0
        System.out.println("Original value of var: " + var);
        System.out.println("Original value of x: " + x);
        System.out.println(); // print a blank line
    }
}
```

```
// now, divide both by 4
var = var / 4;
x = x / 4;

System.out.println("var after division: " + var);

System.out.println("x after division: " + x);

}

}
```

Predicted output:

Original value of var: 10

Original value of x: 10.0

var after division: 2

x after division: 2.5

One here has to note the difference in precision of the different data types. The following example uses the character data type. Characters in Java are encoded using Unicode giving a 16-bit range, or a total of 65,537 different codes.

```
class Example9 {
    public static void main(String args[]) {
        char ch;

        ch = 'X';

        System.out.println("ch contains " + ch);

        ch++; // increment ch

        System.out.println("ch is now " + ch);

        ch = 90; // give ch the value Z

        System.out.println("ch is now " + ch);

    }
}
```

Predicted Output:

```
ch is now X
ch is now Y
ch is now Z
```

The character 'X' is encoded as the number 88, hence when we increment 'ch', we get character number 89, or 'Y'.

The Boolean data type can be either TRUE or FALSE. It can be useful when controlling flow of a program by assigning the Boolean data type to variables which function as flags. Thus program flow would depend on the condition of these variables at the particular instance. Remember that the output of a condition is always Boolean.

```
class Example10 {
    public static void main(String args[]) {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");
        b = false;
        if(b) System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

Predicted output:

```
b is false
b is true
This is executed
10 > 9 is true
```

Introducing Control Statements

These statements will be dealt with in more detail further on in this booklet. For now we will learn about the **if** and the **for loop**.

```
class Example11 {
    public static void main(String args[]) {
        int a,b,c;
        a = 2;
        b = 3;
        c = a - b;
        if (c >= 0) System.out.println("c is a positive number");
        if (c < 0) System.out.println("c is a negative number");
        System.out.println();
        c = b - a;
        if (c >= 0) System.out.println("c is a positive number");
        if (c < 0) System.out.println("c is a negative number");
    }
}
```

Predicted output:

```
c is a negative number
c is a positive number
```

The 'if' statement evaluates a condition and if the result is true, then the following statement/s are executed, else they are just skipped (refer to program output). The line `System.out.println()` simply inserts a blank line. Conditions use the following comparison operators:

Operator	Description
<	Smaller than
>	Greater than
<=	Smaller or equal to, (a<=3) : if a is 2 or 3, then result of comparison is TRUE
>=	Greater or equal to, (a>=3) : if a is 3 or 4, then result of comparison is TRUE
==	Equal to
!=	Not equal

The for loop is an example of an iterative code, i.e. this statement will cause the program to repeat a particular set of code for a particular number of times. In the following example we will be using a counter which starts at 0 and ends when it is smaller than 5, i.e. 4. Therefore the code following the for loop will iterate for 5 times.

```
class Example12 {
    public static void main(String args[]) {
        int count;
        for(count = 0; count < 5; count = count+1)
            System.out.println("This is count: " + count);
        System.out.println("Done!");
    }
}
```

Predicted Output:

```
This is count: 0
This is count: 1
This is count: 2
This is count: 3
This is count: 4
Done!
```

Instead of `count = count+1`, this increments the counter, we can use `count++`

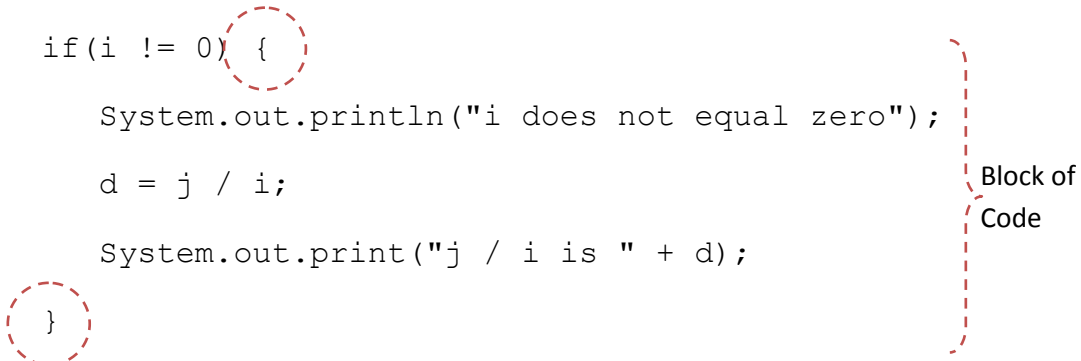
The following table shows all the available shortcut operators:

Operator	Description	Example	Description
++	Increment	a++	a = a + 1 (adds one from a)
--	Decrement	a--	a = a - 1 (subtract one from a)
+=	Add and assign	a+=2	a = a + 2
-=	Subtract and assign	a-=2	a = a - 2
=	Multiply and assign	a=3	a = a * 3
/=	Divide and assign	a/=4	a = a / 4
%=	Modulus and assign	a%=5	a = a mod 5

Blocks of Code

Whenever we write an IF statement or a loop, if there is more than one statement of code which has to be executed, this has to be enclosed in braces, i.e. '{ }'

```
class Example13 {
    public static void main(String args[]) {
        double i, j, d;
        i = 5;
        j = 10;
        if(i != 0) {
            System.out.println("i does not equal zero");
            d = j / i;
            System.out.print("j / i is " + d);
        }
        System.out.println();
    }
}
```



Predicted Output:

```
i does not equal to zero
j/i is 2
```

🌟* Test your skills – Example14

Write a program which can be used to display a conversion table, e.g. Euros to Malta Liri, or Metres to Kilometres.

Hints:

- One variable is required
- You need a loop

The Euro Converter has been provided for you for guidance. Note loop starts at 1 and finishes at 100 (<101). In this case since the conversion rate does not change we did not use a variable, but assigned it directly in the print statement.

```
class EuroConv {
```

```
public static void main (String args []){  
    double eu;  
    System.out.println("Euro conversion table:");  
    System.out.println();  
    for (eu=1;eu<101;eu++)  
        System.out.println(eu+" Euro is euivalent to Lm  
"+(eu*0.43));  
    }  
}
```

The Math Class

In order to perform certain mathematical operations like square root (sqrt), or power (pow); Java has a built in class containing a number of methods as well as static constants, e.g.

Pi = 3.141592653589793 and E = 2.718281828459045. All the methods involving angles use radians and return a double (excluding the Math.round()).

```
class Example15 {  
    public static void main(String args[]) {  
        double x, y, z;  
        x = 3;  
        y = 4;  
        z = Math.sqrt(x*x + y*y);  
        System.out.println("Hypotenuse is " +z);  
    }  
}
```

Predicted Output:

Hypotenuse is 5.0

Please note that whenever a method is called, a particular nomenclature is used where we first specify the class that the particular method belongs to, e.g. Math.round(); where Math is the class name and round is the method name. If a particular method accepts parameters, these are placed in brackets, e.g. Math.max(2.8, 12.9) – in this case it would return 12.9 as being the larger number. A

useful method is the `Math.random()` which would return a random number ranging between 0.0 and 1.0.

Scope and Lifetime of Variables

The following simple programs, illustrate how to avoid programming errors by taking care where to initialize variables depending on the scope.

```
class Example16 {  
    public static void main(String args[]) {  
        int x; // known to all code within main  
        x = 10;  
        if(x == 10) { // start new scope  
            int y = 20; // known only to this block  
            // x and y both known here.  
            System.out.println("x and y: " + x + " " + y);  
            x = y * 2;  
        }  
        // y = 100; // Error! y not known here  
        // x is still known here.  
        System.out.println("x is " + x);  
    }  
}
```

Predicted Output:

```
x and y: 10 20  
x is 40
```

If we had to remove the comment marks from the line, `y = 100;` we would get an error during compilation as `y` is not known since it only exists within the block of code following the 'if' statement.

The next program shows that `y` is initialized each time the code belonging to the looping sequence is executed; therefore `y` is reset to -1 each time and then set to 100. This operation is repeated for three (3) times.

```
class Example17 {  
    public static void main(String args[]) {  
        int x;  
        for(x = 0; x < 3; x++) {  
            int y = -1; // y is initialized each time block is  
entered  
            System.out.println("y is: " + y); // this always  
prints -1  
            y = 100;  
            System.out.println("y is now: " + y);  
        }  
    }  
}
```

Predicted Output:

```
y is: -1  
y is now: 100  
  
y is: -1  
y is now: 100  
  
y is: -1  
y is now: 100
```

Type Casting and Conversions

Casting is the term used when a value is converted from one data type to another, except for Boolean data types which cannot be converted to any other type. Usually conversion occurs to a data type which has a larger range or else there could be loss of precision.

```
class Example18 {    //long to double automatic conversion  
    public static void main(String args[]) {  
        long L;  
        double D;  
        L = 100123285L;  
        D = L;    // L = D is impossible
```

```

        System.out.println("L and D: " + L + " " + D);
    }
}

```

Predicted Output:

L and D: 100123285 1.00123285E8

The general formula used in casting is as follows: (target type) expression, where target type could be int, float, or short, e.g. (int) (x/y)

```

class Example19 { //CastDemo
    public static void main(String args[]) {
        double x, y;
        byte b;
        int i;
        char ch;
        x = 10.0;
        y = 3.0;
        i = (int) (x / y); // cast double to int
        System.out.println("Integer outcome of x / y: " + i);
        i = 100;
        b = (byte) i;
        System.out.println("Value of b: " + b);
        i = 257;
        b = (byte) i;
        System.out.println("Value of b: " + b);
        b = 88; // ASCII code for X
        ch = (char) b;
        System.out.println("ch: " + ch);
    }
}

```

Predicted Output:

Integer outcome of x / y: 3

Value of b: 100

Value of b: 1

ch: X

In the above program, x and y are doubles and so we have loss of precision when converting to integer. We have no loss when converting the integer 100 to byte, but when trying to convert 257 to byte we have loss of precision as 257 exceeds the size which can hold byte. Finally we have casting from byte to char.

```
class Example20 {  
    public static void main(String args[]) {  
        byte b;  
        int i;  
        b = 10;  
        i = b * b; // OK, no cast needed  
        b = 10;  
        b = (byte) (b * b); // cast needed!! as cannot assign int  
to byte  
        System.out.println("i and b: " + i + " " + b);  
    }  
}
```

Predicted Output:

i and b: 100 100

The above program illustrates the difference between automatic conversion and casting. When we are assigning a byte to integer, `i = b * b`, the conversion is automatic. When performing an arithmetic operation the byte type are promoted to integer automatically, but if we want the result as byte, we have to cast it back to byte. This explains why there is the statement: `b = (byte) (b * b)`. Casting has to be applied also if adding variables of type char, as result would else be integer.

Console Input

Most students at this point would be wondering how to enter data while a program is executing. This would definitely make programs more interesting as it adds an element of interactivity at run-time. This is not that straight forward in Java, since Java was not designed to handle console input. The following are the three most commonly used methods to cater for input:

Using the Keyboard Class

One can create a class, which would contain methods to cater for input of the various data types. Another option is to search the internet for the Keyboard Class. This class is easily found as it is used in beginners Java courses. This class is usually found in compiled version, i.e. keyboard.class. This file has to be put in the project folder or else placed directly in the Java JDK. The following is the source code for the Keyboard class just in case it is not available online!

```
import java.io.*;

import java.util.*;

public class Keyboard {

    /******* Error Handling Section

        private static boolean printErrors = true;

        private static int errorCount = 0;

        // Returns the current error count.

        public static int getErrorCount(){

            return errorCount;

        }

        // Resets the current error count to zero.

        public static void resetErrorCount (int count){

            errorCount = 0;

        }

        // Returns a boolean indicating whether input errors are

        // currently printed to standard output.

        public static boolean getPrintErrors(){

            return printErrors;

        }

    }
```



```
// Sets a boolean indicating whether input errors are to be
// printed to standard output.

public static void setPrintErrors (boolean flag){
    printErrors = flag;
}

// Increments the error count and prints the error message
// if appropriate.

private static void error (String str){
    errorCount++;
    if (printErrors)
        System.out.println (str);
}

//***** Tokenized Input Stream Section ****

private static String current_token = null;
private static StringTokenizer reader;
private static BufferedReader in = new BufferedReader
    (new InputStreamReader(System.in));

// Gets the next input token assuming it may be on
// subsequent input lines.

private static String getNextToken() {
    return getNextToken (true);
}

// Gets the next input token, which may already have been
// read.

private static String getNextToken (boolean skip) {
    String token;

    if (current_token == null)
        token = getNextInputToken (skip);
```

```
else {
    token = current_token;
    current_token = null;
}
return token;
}

// Gets the next token from the input, which may come from
// the current input line or a subsequent one. The
// parameter determines if subsequent lines are used.
private static String getNextInputToken (boolean skip) {
    final String delimiters = " \t\n\r\f";
    String token = null;
    try {
        if (reader == null)
            reader = new StringTokenizer
                (in.readLine(), delimiters, true);
        while (token == null ||
            ((delimiters.indexOf (token) >= 0) && skip)){
            while (!reader.hasMoreTokens())
                reader = new StringTokenizer
                    (in.readLine(), delimiters,true);
            token = reader.nextToken();
        }
    }
    catch (Exception exception) {
        token = null;
    }
}
```

```
        return token;
    }

    // Returns true if there are no more tokens to read on the
    // current input line.
    public static boolean endOfLine() {
        return !reader.hasMoreTokens();
    }

//***** Reading Section

    // Returns a string read from standard input.
    public static String readString() {
        String str;
        try {
            str = getNextToken(false);
            while (!endOfLine()) {
                str = str + getNextToken(false);
            }
        }
        catch (Exception exception){
            error ("Error reading String data, null value
returned.");
            str = null;
        }
        return str;
    }

    // Returns a space-delimited substring (a word) read from
    // standard input.
    public static String readWord() {
        String token;
```

```
try {
    token = getNextToken();
}

catch (Exception exception) {
    error ("Error reading String data, null value
returned.");
    token = null;
}

return token;
}

// Returns a boolean read from standard input.
public static boolean readBoolean() {
    String token = getNextToken();
    boolean bool;
    try {
        if (token.toLowerCase().equals("true"))
            bool = true;
        else if (token.toLowerCase().equals("false"))
            bool = false;
        else {
            error ("Error reading boolean data, false value
returned.");
            bool = false;
        }
    }

    catch (Exception exception) {
        error ("Error reading boolean data, false value
returned.");
    }
}
```

```
        bool = false;
    }
    return bool;
}

// Returns a character read from standard input.
public static char readChar() {
    String token = getNextToken(false);
    char value;
    try {
        if (token.length() > 1) {
            current_token = token.substring (1,
token.length());
        } else  current_token = null;
        value = token.charAt (0);
    }
    catch (Exception exception) {
        error ("Error reading char data, MIN_VALUE value
returned.");
        value = Character.MIN_VALUE;
    }
    return value;
}

// Returns an integer read from standard input.
public static int readInt() {
    String token = getNextToken();
    int value;
    try {
        value = Integer.parseInt (token);
    }
```

```
    }

    catch (Exception exception) {

        error ("Error reading int data, MIN_VALUE value
returned.");

        value = Integer.MIN_VALUE;

    }

    return value;

}

// Returns a long integer read from standard input.
public static long readLong(){

    String token = getNextToken();

    long value;

    try {

        value = Long.parseLong (token);

    }

    catch (Exception exception) {

        error ("Error reading long data, MIN_VALUE value
returned.");

        value = Long.MIN_VALUE;

    }

    return value;

}

// Returns a float read from standard input.
public static float readFloat() {

    String token = getNextToken();

    float value;

    try {

        value = (new Float(token)).floatValue();

    }
```

```

    }

    catch (Exception exception) {

        error ("Error reading float data, NaN value
returned.");

        value = Float.NaN;

    }

    return value;

}

// Returns a double read from standard input.
public static double readDouble() {

    String token = getNextToken();

    double value;

    try {

        value = (new Double(token)).doubleValue();

    }

    catch (Exception exception) {

        error ("Error reading double data, NaN value
returned.");

        value = Double.NaN;

    }

    return value;

}

}

```

The above class contains the following methods:

- `public static String readString ()`
 - Reads and returns a string, to the end of the line, from standard input.
- `public static String readWord ()`
 - Reads and returns one space-delimited word from standard input.
- `public static boolean readBoolean ()`

- Reads and returns a boolean value from standard input. Returns false if an exception occurs during the read.
- `public static char readChar ()`
 - Reads and returns a character from standard input. Returns `MIN_VALUE` if an exception occurs during the read.
- `public static int readInt ()`
 - Reads and returns an integer value from standard input. Returns `MIN_VALUE` if an exception occurs during the read.
- `public static long readLong ()`
 - Reads and returns a long integer value from standard input. Returns `MIN_VALUE` if an exception occurs during the read.
- `public static float readFloat ()`
 - Reads and returns a float value from standard input. Returns `NaN` if an exception occurs during the read.
- `public static double readDouble ()`
 - Reads and returns a double value from standard input. Returns `NaN` if an exception occurs during the read.
- `public static int getErrorCount()`
 - Returns the number of errors recorded since the `Keyboard` class was loaded or since the last error count reset.
- `public static void resetErrorCount (int count)`
 - Resets the current error count to zero.
- `public static boolean getPrintErrors ()`
 - Returns a boolean indicating whether input errors are currently printed to standard output.
- `public static void setPrintErrors (boolean flag)`
 - Sets the boolean indicating whether input errors are to be printed to standard input.

Let's try it out by writing a program which accepts three integers and working the average:

```
public class KeyboardInput {  
    public static void main (String args[]) {  
        System.out.println("Enter a number:");  
        int a = Keyboard.readInt ();  
        System.out.println("Enter a second number:");  
        int b = Keyboard.readInt ();  
        System.out.println("Enter a third number:");  
        int c = Keyboard.readInt ();  
    }  
}
```



```
        System.out.println("The average is " + (a+b+c)/3);
    }
}
```

After printing a statement, the program will wait for the user to enter a number and store it in the particular variable. It utilizes the **readInt()** method. Finally it will display the result of the average.

Using the Scanner Class

In Java 5 a particular class was added, the Scanner class. This class allows users to create an instance of this class and use its methods to perform input. Let us look at the following example which performs the same operation as the one above (works out the average of three numbers):

```
import java.util.Scanner;

public class ScannerInput {

    public static void main(String[] args) {

        //... Initialize Scanner to read from console.
        Scanner input = new Scanner(System.in);

        System.out.print("Enter first number : ");

        int a = input.nextInt();

        System.out.print("Enter second number: ");

        int b = input.nextInt();

        System.out.print("Enter last number  : ");

        int c = input.nextInt();

        System.out.println("Average is " + (a+b+c)/3);

    }

}
```

By examining the code we see that first we have to import the `java.util.Scanner` as part of the `java.util` package. Next we create an instance of `Scanner` and name it as we like, in this case we named it "input". We have to specify also the type of input expected (`System.in`). The rest is similar to the program which uses the `Keyboard` class, the only difference is the name of the method used, in this case it is called **nextInt()** rather than `readInt()`. This time the method is called as part of the instance created, i.e. **input.nextInt()**

Using Swing Components

This is probably the most exciting version, since the Swing package offers a graphical user interface (GUI) which allows the user to perform input into a program via the mouse, keyboard and other input devices.

```
import javax.swing.*; // * means 'all'

public class SwingInput {

    public static void main(String[] args) {

        String temp; // Temporary storage for input.

        temp = JOptionPane.showInputDialog(null, "First
number");

        int a = Integer.parseInt(temp); // String to int

        temp = JOptionPane.showInputDialog(null, "Second
number");

        int b = Integer.parseInt(temp);

        temp = JOptionPane.showInputDialog(null, "Third
number");

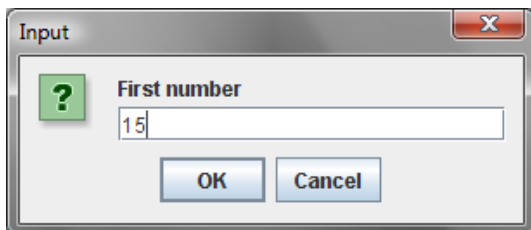
        int c = Integer.parseInt(temp);

        JOptionPane.showMessageDialog(null, "Average is " +
(a+b+c)/3);

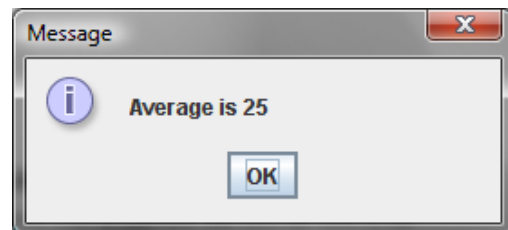
    }

}
```

One has to note that the input is stored as a string, **temp**, and then parsed to integer using the method **parseInt()**. This time the method accepts a parameter, **temp**, and returns an integer. When the above program is executed, a dialog box will appear on screen with a field to accept input from user via keyboard (**JOptionPane.showInputDialog**). This is repeated three times and finally the result is again displayed in a dialog box (**JOptionPane.showMessageDialog**).



JOptionPane.showInputDialog



JOptionPane.showMessageDialog

Part 2 - Advanced Java Programming

Control Statements - The if Statement

```
if(condition) statement;  
else statement;
```

Note:

- **else** clause is optional
- targets of both the **if** and **else** can be blocks of statements.

The general form of the **if**, using blocks of statements, is:

```
if(condition)  
    {  
        statement sequence  
    }  
else  
    {  
        statement sequence  
    }
```

If the conditional expression is true, the target of the **if** will be executed; otherwise, if it exists, the target of the **else** will be executed. At no time will both of them be executed. The conditional expression controlling the **if** must produce a **boolean** result.

Guessing Game (Guess.java)

The program asks the player for a letter between A and Z. If the player presses the correct letter on the keyboard, the program responds by printing the message ****Right****.

```
// Guess the letter game.

class Guess {

    public static void main(String args[])
        throws java.io.IOException {

        char ch, answer = 'K';

        System.out.println("I'm thinking of a letter between A
and Z.");

        System.out.print("Can you guess it: ");

        ch = (char) System.in.read(); // read a char from the
        keyboard

        if(ch == answer) System.out.println("** Right **");

    }

}
```

Extending the above program to use the else statement:

```
// Guess the letter game, 2nd version.

class Guess2 {

    public static void main(String args[])
        throws java.io.IOException {

        char ch, answer = 'K';

        System.out.println("I'm thinking of a letter between A
and Z.");

        System.out.print("Can you guess it: ");

        ch = (char) System.in.read(); // get a char

        if(ch == answer) System.out.println("** Right **");
        else System.out.println("...Sorry, you're wrong.");

    }

}
```

Nested if

The main thing to remember about nested **ifs** in Java is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and not already associated with an **else**. Here is an example:

```
if(i == 10) {
    if(j < 20) a = b;
    if(k > 100) c = d;
    else a = c; // this else refers to if(k > 100)
}
else a = d; // this else refers to if(i == 10)
```

Guessing Game v.3

```
// Guess the letter game, 3rd version.
class Guess3 {
    public static void main(String args[])
        throws java.io.IOException {
        char ch, answer = 'K';

        System.out.println("I'm thinking of a letter between A
and Z.");

        System.out.print("Can you guess it: ");
        ch = (char) System.in.read(); // get a char
        if(ch == answer) System.out.println("*** Right ***");
        else {
            System.out.print("...Sorry, you're ");
            // a nested if
            if(ch < answer) System.out.println("too low");
            else System.out.println("too high");
        }
    }
}
```

A sample run is shown here:

I'm thinking of a letter between A and Z.

Can you guess it: Z

...Sorry, you're too high

if-else-if Ladder

```
if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
...
else
    statement;
```

The conditional expressions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is bypassed. If none of the conditions is true, the final **else** statement will be executed. The final **else** often acts as a default condition; that is, if all other conditional tests fail, the last **else** statement is performed. If there is no final **else** and all other conditions are false, no action will take place.

```
// Demonstrate an if-else-if ladder.

class Ladder {

public static void main(String args[]) {

    int x;

    for(x=0; x<6; x++) {

        if(x==1)

            System.out.println("x is one");

        else if(x==2)

            System.out.println("x is two");
```

```
else if(x==3)

    System.out.println("x is three");

else if(x==4)

    System.out.println("x is four");

else

    System.out.println("x is not between 1 and 4");

    }

    }

}
```

The program produces the following output:

```
x is not between 1 and 4

x is one

x is two

x is three

x is four

x is not between 1 and 4
```

Ternary (?) Operator

Declared as follows:

```
Exp1 ? Exp2 : Exp3;
```

Exp1 would be a **boolean** expression, and Exp2 and Exp3 are expressions of any type other than void. The type of Exp2 and Exp3 must be the same, though. Notice the use and placement of the colon. Consider this example, which assigns absval the absolute value of val:

```
absval = val < 0 ? -val : val; // get absolute value of val
```

Here, absval will be assigned the value of val if val is zero or greater. If val is negative, then absval will be assigned the negative of that value (which yields a positive value).

The same code written using the **if-else** structure would look like this:

```
if(val < 0) absval = -val;
else absval = val;
```

e.g. 2 This program divides two numbers, but will not allow a division by zero.

```
// Prevent a division by zero using the ?.
class NoZeroDiv {
    public static void main(String args[]) {
        int result;
        for(int i = -5; i < 6; i++) {
            result = i != 0 ? 100 / i : 0;
            if(i != 0)
                System.out.println("100 / " + i + " is " + result);
        }
    }
}
```

The output from the program is shown here:

```
100 / -5 is -20
100 / -4 is -25
100 / -3 is -33
100 / -2 is -50
100 / -1 is -100
100 / 1 is 100
100 / 2 is 50
100 / 3 is 33
100 / 4 is 25
100 / 5 is 20
```

Please note:

```
result = i != 0 ? 100 / i : 0;
```

result is assigned the outcome of the division of 100 by **i**. However, this division takes place only if **i** is not zero. When **i** is zero, a placeholder value of zero is assigned to **result**. Here is the preceding program rewritten a bit more efficiently. It produces the same output as before.

```
// Prevent a division by zero using the ?.
class NoZeroDiv2 {
    public static void main(String args[]) {
        for(int i = -5; i < 6; i++)
            if(i != 0 ? true : false)
                System.out.println("100 / " + i +
                    " is " + 100 / i);
    }
}
```

Notice the **if** statement. If **i** is zero, then the outcome of the **if** is false, the division by zero is prevented, and no result is displayed. Otherwise the division takes place.

switch Statement (case of)

The **switch** provides for a multi-way branch. Thus, it enables a program to select among several alternatives. Although a series of nested **if** statements can perform multi-way tests, for many situations the **switch** is a more efficient approach.

```
switch(expression) {  
    case constant1:  
        statement sequence  
        break;  
    case constant2:  
        statement sequence  
        break;  
    case constant3:  
        statement sequence  
        break;  
    ...  
    default:  
        statement sequence  
}
```

- The **switch** expression can be of type **char**, **byte**, **short**, or **int**. (Floating-point expressions, for example, are not allowed.)
- Frequently, the expression controlling the **switch** is simply a variable.
- The **case** constants must be literals of a type compatible with the expression.
- No two **case** constants in the same **switch** can have identical values.
- The **default** statement sequence is executed if no **case** constant matches the expression. The **default** is optional; if it is not present, no action takes place if all matches fail. When a match is found, the statements associated with that **case** are executed until the **break** is encountered or, in the case of **default** or the last **case**, until the end of the **switch** is reached.

The following program demonstrates the **switch**.

```
// Demonstrate the switch.

class SwitchDemo {

    public static void main(String args[]) {

        int i;

        for(i=0; i<10; i++)

            switch(i) {

                case 0:

                    System.out.println("i is zero");

                    break;

                case 1:

                    System.out.println("i is one");

                    break;

                case 2:

                    System.out.println("i is two");

                    break;

                case 3:

                    System.out.println("i is three");

                    break;

                case 4:

                    System.out.println("i is four");

                    break;

                default:

                    System.out.println("i is five or more");

            }

    }

}
```

The output produced by this program is shown here:

```
i is zero
i is one
i is two
i is three
i is four
i is five or more
i is five or more
i is five or more
i is five or more
i is five or more
```

The **break** statement is optional, although most applications of the **switch** will use it. When encountered within the statement sequence of a **case**, the **break** statement causes program flow to exit from the entire **switch** statement and resume at the next statement outside the **switch**. However, if a **break** statement does not end the statement sequence associated with a **case**, then all the statements *at and following* the matching **case** will be executed until a **break** (or the end of the **switch**) is encountered. For example,

```
// Demonstrate the switch without break statements.

class NoBreak {

    public static void main(String args[]) {

        int i;

        for(i=0; i<=5; i++) {

            switch(i) {

                case 0:

                    System.out.println("i is less than one");

                case 1:

                    System.out.println("i is less than two");

                case 2:

                    System.out.println("i is less than three");

                case 3:

                    System.out.println("i is less than four");

                case 4:

                    System.out.println("i is less than five");

            }

        }

    }

}
```

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

Output:

```
i is less than one  
i is less than two  
i is less than three  
i is less than four  
i is less than five  
i is less than two  
i is less than three  
i is less than four  
i is less than five  
i is less than three  
i is less than four  
i is less than five  
i is less than four  
i is less than five  
i is less than four  
i is less than five
```

Execution will continue into the next **case** if no **break** statement is present.

You can have empty **cases**, as shown in this example:

```
switch(i) {  
    case 1:  
    case 2:  
    case 3: System.out.println("i is 1, 2 or 3");  
    break;  
    case 4: System.out.println("i is 4");  
    break;  
}
```

Nested switch

```
switch(ch1) {
    case 'A': System.out.println("This A is part of outer
switch.");
        switch(ch2) {
            case 'A':
                System.out.println("This A is part of inner
switch");
                break;
            case 'B': // ...
        } // end of inner switch
    break;
    case 'B': // ...
```

🔧* Mini-Project – Java Help System (Help.java)

Your program should display the following options on screen:

Help on:

1. if
2. switch

Choose one:

To accomplish this, you will use the statement

```
System.out.println("Help on:");
System.out.println(" 1. if");
System.out.println(" 2. switch");
System.out.print("Choose one:
```

Next, the program obtains the user's selection

```
choice = (char) System.in.read();
```

Once the selection has been obtained, the display the syntax for the selected statement.

```
switch(choice) {  
    case '1':  
        System.out.println("The if:\n");  
        System.out.println("if(condition)\n");  
        System.out.println("else statement;");  
        break;  
    case '2':  
        System.out.println("The switch:\n");  
        System.out.println("switch(\n");  
        System.out.println(" case\n");  
        System.out.println(" statement\n");  
        System.out.println(" break;");  
        System.out.println(" // ...");  
        System.out.println("}");  
        break;  
    default:  
        System.out.print("Selection not found.");  
}
```

***default** clause catches invalid choices. For example, if the user enters 3, no **case** constants will match, causing the **default** sequence to execute.*

Complete Listing

```
/*  
Project 3-1  
A simple help system.  
*/  
  
class Help {  
    public static void main(String args[])  
        throws java.io.IOException {
```

```
char choice;

System.out.println("Help on:");

System.out.println(" 1. if");

System.out.println(" 2. switch");

System.out.print("Choose one: ");

choice = (char) System.in.read();

System.out.println("\n");

switch(choice) {

case '1':

    System.out.println("The if:\n");

    System.out.println("if(condition) statement;");

    System.out.println("else statement;");

    break;

case '2':

    System.out.println("The switch:\n");

    System.out.println("switch(expression) {");

    System.out.println(" case constant:");

    System.out.println(" statement sequence");

    System.out.println(" break;");

    System.out.println(" // ...");

    System.out.println("}");

    break;

default:

    System.out.print("Selection not found.");

}

}
```

Sample run:

Help on:

1. if

2. switch

Choose one: 1

The if:

```
if(condition) statement;
```

```
else statement;
```

The for Loop

Loops are structures used to make the program repeat one or many instructions for 'n' times as specified in the declaration of the loop.

The for Loop can be used for just one statement:

```
for(initialization; condition; iteration) statement;
```

or to repeat a block of code:

```
for(initialization; condition; iteration)
{
    statement sequence
}
```

- *Initialization* = assignment statement that sets the initial value of the *loop control variable*, (counter)
- *Condition* = Boolean expression that determines whether or not the loop will repeat
- *Iteration* = amount by which the loop control variable will change each time the loop is repeated.

► The **for** loop executes only/till the condition is true.

Example: using a 'for' loop to print the square roots of the numbers between 1 and 99. (It also displays the rounding error present for each square root).

```
// Show square roots of 1 to 99 and the rounding error.

class SqrRoot {

    public static void main(String args[]) {

        double num, sroot, rerr;

        for(num = 1.0; num < 100.0; num++) {

            sroot = Math.sqrt(num);

            System.out.println("Square root of " + num +

                " is " + sroot);

            // compute rounding error

            rerr = num - (sroot * sroot);

            System.out.println("Rounding error is " + rerr);

            System.out.println();

        }

    }

}
```

'For' loop counters (loop control variables) can either increment or decrement,

// A negatively running for loop.

```
class DecrFor {

    public static void main(String args[]) {

        int x;

        for(x = 100; x > -100; x -= 5)

            System.out.println(x);

    }

}
```

Counter
decrements by 5
($x = x - 5$)

Multiple Loop Control Variable

Using more than one variable in the same loop is possible:

```
// Use commas in a for statement.
```

```
class Comma {  
    public static void main(String args[]) {  
        int i, j;  
        for(i=0, j=10; i < j; i++, j--)  
            System.out.println("i and j: " + i + " " + j);  
    }  
}
```

*'i' and 'j' are the
two variables used
in the same loop*

Expected output:

```
i and j: 0 10  
i and j: 1 9  
i and j: 2 8  
i and j: 3 7  
i and j: 4 6
```

Terminating a loop via user intervention

Let us write a program which involves a loop and this is stopped when the user types 's' on the keyboard:

```
// Loop until an S is typed.
```

```
class ForTest {  
    public static void main(String args[])  
        throws java.io.IOException {  
        int i;  
        System.out.println("Press S to stop.");  
        for(i = 0; (char) System.in.read() != 'S'; i++)  
            System.out.println("Pass #" + i);  
    }  
}
```

Interesting For Loop Variations

It is possible to leave out parts of the loop declaration:

// Example 1 - Parts of the for can be empty.

```
class Empty {  
    public static void main(String args[]) {  
        int i;  
        for(i = 0; i < 10; ) {  
            System.out.println("Pass #" + i);  
            i++; // increment loop control var  
        }  
    }  
}
```

// Example 2 - Parts of the for can be empty.

```
class Empty2 {  
    public static void main(String args[]) {  
        int i;  
        i = 0; // move initialization out of loop  
        for(; i < 10; ) {  
            System.out.println("Pass #" + i);  
            i++; // increment loop control var  
        }  
    }  
}
```

Initialising the loop out of the 'for' statement is only required when the value needs to be a result of another complex process which cannot be written inside the declaration.

Infinite Loops

Sometimes one needs to create an infinite loop, i.e. a loop which never ends! (However it can be stopped using the break statement). An example of an infinite loop declaration is as follows:

```
for(;;)

{

    // ... statements

}
```

*N.B. Using **break** to terminate an infinite loop will be discussed later on in the course.*

No 'Body' Loops

Loops can be declared without a body. This can be useful in particular situations, consider the following example:

```
// Loop without body.

class Empty3 {

    public static void main(String args[]) {

        int i;

        int sum = 0;

        // sum the numbers through 5

        for(i = 1; i <= 5; sum += i++) ;

        System.out.println("Sum is " + sum);

    }

}
```

Two operations are carried on, $sum = sum + i$ and $i = i + 1$

Predicted Output:

Sum is 15

Declaring variables inside the loop

Variables can be declared inside the loop itself but one must remember that in such case the variable exists only inside the loop!

```
// Declare loop variable inside the for.
class ForVar {
    public static void main(String args[]) {
        int sum = 0;
        int fact = 1;
        // compute the factorial of the numbers through 5
        for(int i = 1; i <= 5; i++) {
            sum += i; // i is known throughout the loop
            fact *= i;
        }
        // but, i is not known here.
        System.out.println("Sum is " + sum);
        System.out.println("Factorial is " + fact);
    }
}
```

Enhanced For loop

This type of loop will be discussed later on in the course as it involves arrays.

The While Loop

```
while (condition) statement; //or more than one statement
```

The condition could be any valid Boolean expression. The loop will function only if the condition is true. If false it will move on to the next line of code.

```
// Demonstrate the while loop.

class WhileDemo {

    public static void main(String args[]) {

        char ch;

        // print the alphabet using a while loop

        ch = 'a';

        while(ch <= 'z') {

            System.out.print(ch);

            ch++;

        }

    }

}
```

The above program will output the alphabet. As can be seen in the code the while loop will result false when the character is greater than 'z'. The condition is tested at the beginning of the program.

```
// Compute integer powers of 2.

class Power {

    public static void main(String args[]) {

        int e;

        int result;

        for(int i=0; i < 10; i++) {

            result = 1;

            e = i;

            while(e > 0) {

                result *= 2;

                e--;

            }

        }

    }

}
```

```
        System.out.println("2 to the power of " + i + " is "
+ result);
    }

}

}
```

Predicted Output:

```
2 to the power of 0 is 1
2 to the power of 1 is 2
2 to the power of 2 is 4
2 to the power of 3 is 8
2 to the power ... (up to 'of 9 is 512')
```

The do-while Loop

This conditional statement checks the Boolean expression after going at least one time through the loop. The do-while is declared as follows:

```
do {
    statements;
} while(condition);
```

Braces are used if there is more than one statements and to improve program readability.

```
// Demonstrate the do-while loop.
class DWDemo {
    public static void main(String args[])
        throws java.io.IOException {
        char ch;
        do {
            System.out.print("Press a key followed by ENTER: ");
            ch = (char) System.in.read(); // get a char
        } while(ch != 'q');
    }
}
```

```
// Guess the letter game, 4th version.
class Guess4 {
    public static void main(String args[])
        throws java.io.IOException {
        char ch, answer = 'K';
        do {
            System.out.println("I'm thinking of a letter between
            A and Z.");
            System.out.print("Can you guess it: ");
            // read a letter, but skip cr/lf
            do {
                ch = (char) System.in.read(); // get a char
            } while(ch == '\n' | ch == '\r');
            if(ch == answer) System.out.println("** Right **");
            else {
                System.out.print("...Sorry, you're ");
                if(ch < answer) System.out.println("too low");
                else System.out.println("too high");
                System.out.println("Try again!\n");
            }
        } while(answer != ch);
    }
}
```

The function of this statement is to skip carriage return and line feed characters

Predicted Output:

I'm thinking of a letter between A and Z.

Can you guess it: A

...Sorry, you're too low

Try again!

I'm thinking of a letter between A and Z.

Can you guess it: Z

...Sorry, you're too high

Try again!

I'm thinking of a letter between A and Z.

Can you guess it: K

** Right **

☛ Mini-Project 2– Java Help System (Help2.java)

We are going to work on our previous project. Copy all the code and add the following code:

```
do {
    System.out.println("Help on:");
    System.out.println(" 1. if");
    System.out.println(" 2. switch");
    System.out.println(" 3. for");
    System.out.println(" 4. while");
    System.out.println(" 5. do-while\n");
    System.out.print("Choose one: ");
    do {
        choice = (char) System.in.read();
    } while(choice == '\n' | choice == '\r');
} while( choice < '1' | choice > '5');
```

Now extend the switch as follows:

```
switch(choice) {
    case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
    case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) {");
        System.out.println(" case constant:");
        System.out.println(" statement sequence");
        System.out.println(" break;");
        System.out.println(" // ...");
        System.out.println("}");
```

```
        break;
    case '3':
        System.out.println("The for:\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break;
    case '4':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
    case '5':
        System.out.println("The do-while:\n");
        System.out.println("do {");
        System.out.println(" statement;");
        System.out.println("} while (condition);");
        break;
    }
```

The default statement has been removed as the loop ensures that a proper response is entered or else the program will continue to execute.

Complete listing

```
/*
    Project 3-2
    An improved Help system that uses a
    do-while to process a menu selection.
*/
class Help2 {
    public static void main(String args[])
        throws java.io.IOException {
        char choice;
```

```
do {  
    System.out.println("Help on:");  
    System.out.println(" 1. if");  
    System.out.println(" 2. switch");  
    System.out.println(" 3. for");  
    System.out.println(" 4. while");  
    System.out.println(" 5. do-while\n");  
    System.out.print("Choose one: ");  
    do {  
        choice = (char) System.in.read();  
    } while(choice == '\n' | choice == '\r');  
} while( choice < '1' | choice > '5');  
System.out.println("\n");  
switch(choice) {  
    case '1':  
        System.out.println("The if:\n");  
        System.out.println("if(condition)  
statement;");  
        System.out.println("else statement;");  
        break;  
    case '2':  
        System.out.println("The switch:\n");  
        System.out.println("switch(expression) {");  
        System.out.println(" case constant:");  
        System.out.println(" statement sequence");  
        System.out.println(" break;");  
        System.out.println(" // ...");  
        System.out.println("}");  
        break;  
    case '3':  
        System.out.println("The for:\n");
```

```
        System.out.print("for(init; condition;
        iteration)");

        System.out.println(" statement;");

        break;

    case '4':

        System.out.println("The while:\n");

        System.out.println("while(condition)
        statement;");

        break;

    case '5':

        System.out.println("The do-while:\n");

        System.out.println("do {");

        System.out.println(" statement;");

        System.out.println("} while (condition);");

        break;

    }

}

}
```

Using Break to Terminate a Loop

One can use the 'break' command to terminate voluntarily a loop. Execution will continue from the next line following the loop statements,

e.g. 1 Automatic termination (hard-coded)

```
class BreakDemo {
    public static void main(String args[]) {
        int num;
        num = 100;
        for(int i=0; i < num; i++) {
            if(i*i >= num) break; // terminate loop if i*i >= 100
            System.out.print(i + " ");
        }
        System.out.println("Loop complete.");
    }
}
```

*When $i = 10$, $i*i = 100$. Therefore the 'if' condition is satisfied and the loop terminates before $i = 100$*

Expected Output:

0 1 2 3 4 5 6 7 8 9 Loop complete.

e.g. 2 Termination via user intervention

```
class Break2 {
    public static void main(String args[])
        throws java.io.IOException {
        char ch;
        for( ; ; ) {
            ch = (char) System.in.read(); // get a char
            if(ch == 'q') break;
        }
        System.out.println("You pressed q!");
    }
}
```

In the above program there is an infinite loop, `for(; ;)`. This means that the program will never terminate unless the user presses a particular letter on the keyboard, in this case 'q'.

If we have nested loops, i.e. a loop within a loop, the 'break' will terminate the inner loop. It is not recommended to use many 'break' statement in nested loops as it could lead to bad programs. However there could be more than one 'break' statement in a loop. If there is a switch statement in a loop, the 'break' statement will affect the switch statement **only**. The following code shows an example of nested loops using the 'break' to terminate the inner loop;

```
// Using break with nested loops

class Break3 {
    public static void main(String args[]) {
        for(int i=0; i<3; i++) {
            System.out.println("Outer loop count: " + i);
            System.out.print(" Inner loop count: ");
            int t = 0;
            while(t < 100) {
                if(t == 10) break; // terminate loop if t is 10
                System.out.print(t + " ");
                t++;
            }
            System.out.println();
        }
        System.out.println("Loops complete.");
    }
}
```

Predicted Output:

```
Outer loop count: 0
Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 1
Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 2
Inner loop count: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

Terminating a loop with break and use labels to carry on execution

Programmers refer to this technique as the GOTO function where one instructs the program to jump to another location in the code and continue execution. However Java does not offer this feature but it can be implemented by using break and labels. Labels can be any valid Java identifier and a colon should follow. Labels have to be declared before a block of code, e.g.

```
// Using break with labels.
```

```
class Break4 {
    public static void main(String args[]) {
        int i;
        for(i=1; i<4; i++) {
            one: {
                two: {
                    three: {
                        System.out.println("\ni is " + i);
                        if(i==1) break one;
                        if(i==2) break two;
                        if(i==3) break three;
                        // the following statement is never executed
                        System.out.println("won't print");
                    }
                }
            }
        }
    }
}
```

One, two and three are labels

```

        System.out.println("After block three."); ← three
    }
    System.out.println("After block two."); ← two
}
System.out.println("After block one."); ← one
}

//the following statement executes on termination of the
for loop
System.out.println("After for.");
}
}

```

Predicted Output:

```

i is 1
After block one.
i is 2
After block two.
After block one.
i is 3
After block three.
After block two.
After block one.
After for.

```

Interpreting the above code can prove to be quite tricky. When 'i' is 1, execution will break after the first 'if' statement and resume where there is the label 'one'. This will execute the statement labelled 'one' above. When 'i' is 2, this time execution will resume at the point labelled 'two' and hence will also execute the following statements including the one labelled 'one'.

The following code is another example using labels but this time the label marks a point outside the loop:

```

class Break5 {
    public static void main(String args[]) {
        done:
        for(int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                for(int k=0; k<10; k++) {
                    System.out.println(k);
                    if(k == 5) break done; // jump to done
                }
                System.out.println("After k loop"); // skipped
            }
            System.out.println("After j loop"); // skipped
        }
        System.out.println("After i loop");
    }
}

```


Predicted Output:

```
0
1
2
3
4
5
After i loop
```

The 'k' loop is terminated when 'k' = 5. However the 'j' loop is skipped since the break operation cause execution to resume at a point outside the loops. Hence only the 'i' loop terminates and thus the relevant statement is printed.

The next example shows the difference in execution taking care to where one puts the label:

```
class Break6 {
    public static void main(String args[]) {
        int x=0, y=0;
        stop1: for(x=0; x < 5; x++) {
            for(y = 0; y < 5; y++) {
                if(y == 2) break stop1;
                System.out.println("x and y: " + x + " " + y);
            }
        }
        System.out.println();
        for(x=0; x < 5; x++)
        stop2: {
            for(y = 0; y < 5; y++) {
                if(y == 2) break stop2;
                System.out.println("x and y: " + x + " " + y);
            }
        }
    }
}
```

Predicted Output:

```
x and y: 0 0
x and y: 0 1

x and y: 0 0
x and y: 0 1
x and y: 1 0
x and y: 1 1
x and y: 2 0
x and y: 2 1
x and y: 3 0
x and y: 3 1
x and y: 4 0
x and y: 4 1
```

In the first part the inner loop stops when 'y' = 2. The break operation forces the program to skip the outer 'for' loop, print a blank line and start the next set of loops. This time the label is placed after the 'for' loop declaration. Hence the break operation is only operating on the inner loop this time. In fact 'x' goes all the way from 0 to 4, with 'y' always stopping when it reaches a value of 2.

The break – label feature can only be used within the same block of code. The following code is an example of misuse of the break – label operation:

```
class BreakErr {
    public static void main(String args[]) {
        one: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
        }
        for(int j=0; j<100; j++) {
            if(j == 10) break one;
            System.out.print(j + " ");
        }
    }
}
```

This break cannot continue from the assigned label since it is not part of the same block

Use of Continue (complement of Break)

The 'continue' feature can force a loop to iterate out of its normal control behaviour. It is regarded as the complement of the break operation. Let us consider the following example,

```
class ContDemo {
    public static void main(String args[]) {
        int i;
        for(i = 0; i<=100; i++) {
            if((i%2) != 0) continue;
            System.out.println(i);
        }
    }
}
```

Predicted Output:

```
0
2
4
6
...
100
```

The program prints on screen the even numbers. This happens since whenever the modulus results of 'i' by 2 are not equal to '0', the 'continue' statement forces loop to iterate bypassing the following statement (modulus refers to the remainder of dividing 'i' by 2).

In 'while' and 'do-while' loops, a 'continue' statement will cause control to go directly to the conditional expression and then continue the looping process. In the case of the 'for' loop, the iteration expression of the loop is evaluated, then the conditional expression is executed, and then the loop continues.

Continue + Label

It is possible to use labels with the continue feature. It works the same as when we used it before in the break operation.

```
class ContToLabel {
    public static void main(String args[]) {
        outerloop:
            for(int i=1; i < 10; i++) {
                System.out.print("\nOuter loop pass " + i +
                    ", Inner loop: ");
                for(int j = 1; j < 10; j++) {
                    if(j == 5) continue outerloop;
                    System.out.print(j);
                }
            }
    }
}
```

Predicted Output:

```
Outer loop pass 1, Inner loop: 1234
Outer loop pass 2, Inner loop: 1234
Outer loop pass 3, Inner loop: 1234
Outer loop pass 4, Inner loop: 1234
Outer loop pass 5, Inner loop: 1234
Outer loop pass 6, Inner loop: 1234
Outer loop pass 7, Inner loop: 1234
Outer loop pass 8, Inner loop: 1234
Outer loop pass 9, Inner loop: 1234
```

Note that the inner loop is allowed to execute until 'j' is equal to 5. Then loop is forced to outer loop.

☛ Mini-Project 3– Java Help System (Help3.java)

This project puts the finishing touches on the Java help system that was created in the previous projects. This version adds the syntax for ‘break’ and ‘continue’. It also allows the user to request the syntax for more than one statement. It does this by adding an outer loop that runs until the user enters ‘q’ as a menu selection.

1. Copy all the code from Help2.java into a new file, Help3.java
2. Create an outer loop which covers the whole code. This loop should be declared as infinite but terminate when the user presses ‘q’ (use the break)
3. Your menu should look like this:

```
do {
    System.out.println("Help on:");
    System.out.println(" 1. if");
    System.out.println(" 2. switch");
    System.out.println(" 3. for");
    System.out.println(" 4. while");
    System.out.println(" 5. do-while");
    System.out.println(" 6. break");
    System.out.println(" 7. continue\n");
    System.out.print("Choose one (q to quit): ");
    do {
        choice = (char) System.in.read();
    } while(choice == '\n' | choice == '\r');
} while( choice < '1' | choice > '7' & choice != 'q');
```

4. Adjust the switch statement to include the ‘break’ and ‘continue’ features.

Complete Listing

```
class Help3 {

    public static void main(String args[])

        throws java.io.IOException {

        char choice;

        for(;;) {

            do {

                System.out.println("Help on:");

                System.out.println(" 1. if");

                System.out.println(" 2. switch");

                System.out.println(" 3. for");

                System.out.println(" 4. while");
```

```
System.out.println(" 5. do-while");

System.out.println(" 6. break");

System.out.println(" 7. continue\n");

System.out.print("Choose one (q to quit): ");

do {

    choice = (char) System.in.read();

} while(choice == '\n' | choice == '\r');

while( choice < '1' | choice > '7' & choice != 'q');

if(choice == 'q') break;

System.out.println("\n");

switch(choice) {

    case '1':

        System.out.println("The if:\n");

        System.out.println("if(condition) statement;");

        System.out.println("else statement;");

        break;

    case '2':

        System.out.println("The switch:\n");

        System.out.println("switch(expression) {");

        System.out.println("  case constant:");

        System.out.println("    statement sequence");

        System.out.println("    break;");

        System.out.println("  // ...");

        System.out.println("}");

        break;

    case '3':
```

```
System.out.println("The for:\n");

System.out.print("for(init; condition; iteration)");

System.out.println(" statement;");

break;

case '4':

    System.out.println("The while:\n");

    System.out.println("while(condition) statement;");

    break;

case '5':

    System.out.println("The do-while:\n");

    System.out.println("do {");

    System.out.println("    statement;");

    System.out.println("} while (condition);");

    break;

case '6':

    System.out.println("The break:\n");

    System.out.println("break; or break label;");

    break;

case '7':

    System.out.println("The continue:\n");

    System.out.println("continue; or continue label;");

    break;

}

System.out.println();

}

}
```

Nested Loops

A nested loop is a loop within a loop. The previous examples already included such loops. Another example to consider is the following:

```
class FindFac {  
    public static void main(String args[]) {  
        for(int i=2; i <= 100; i++) {  
            System.out.print("Factors of " + i + ": ");  
            for(int j = 2; j < i; j++)  
                if((i%j) == 0) System.out.print(j + " ");  
            System.out.println();  
        }  
    }  
}
```

The above code prints the factors of each number starting from 2 up to 100. Part of the output is as follows:

```
Factors of 2:  
Factors of 3:  
Factors of 4: 2  
Factors of 5:  
Factors of 6: 2 3  
Factors of 7:  
Factors of 8: 2 4  
Factors of 9: 3  
Factors of 10: 2 5  
...
```

Can you think of a way to make the above code more efficient? (Reduce the number of iterations in the inner loop).

Class Fundamentals

Definition

A class is a sort of template which has attributes and methods. An object is an instance of a class, e.g. Riccardo is an object of type Person. A class is defined as follows:

```
class classname {
    // declare instance variables
    type var1;
    type var2;
    // ...
    type varN;
    // declare methods
    type method1(parameters) {
        // body of method
    }
    type method2(parameters) {
        // body of method
    }
    // ...
    type methodN(parameters) {
        // body of method
    }
}
```

The classes we have used so far had only one method, **main()**, however not all classes specify a main method. The main method is found in the main class of a program (starting point of program).

The Vehicle Class

The following is a class named 'Vehicle' having three attributes, 'passengers' – the number of passengers the vehicle can carry, 'fuelcap' – the fuel capacity of the vehicle and 'mpg' – the fuel consumption of the vehicle (miles per gallon).

```
class Vehicle {
    int passengers; //number of passengers
    int fuelcap;    //fuel capacity in gallons
    int mpg;        //fuel consumption
}
```

Please note that up to this point there is no OBJECT. By typing the above code a new data type is created which takes three parameters. To create an instance of the Vehicle class we use the following statement:

```
Vehicle minivan = new Vehicle ();
```

To set the values of the parameters we use the following syntax:

```
minivan.fuelcap = 16; //sets value of fuel capacity to 16
```


Note the general form of the previous statement: *object.member*

Using the *Vehicle* class

Having created the *Vehicle* class, let us create an instance of that class:

```
class VehicleDemo {  
    public static void main(String args[]) {  
        Vehicle minivan = new Vehicle();  
        int range;  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelcap = 16;  
        minivan.mpg = 21;
```

Till now we have created an instance of *Vehicle* called 'minivan' and assigned values to passengers, fuel capacity and fuel consumption. Let us add some statements to work out the distance that this vehicle can travel with a tank full of fuel:

```
        // compute the range assuming a full tank of gas  
        range = minivan.fuelcap * minivan.mpg;  
        System.out.println("Minivan can carry " +  
            minivan.passengers + " with a range of " + range);  
    }  
}
```

Creating more than one instance

It is possible to create more than one instance in the same program, and each instance would have its own parameters. The following program creates another instance, *sportscar*, which has different instance variables and finally display the range each vehicle can travel having a full tank.

```
class TwoVehicles {  
    public static void main(String args[]) {  
        Vehicle minivan = new Vehicle();  
        Vehicle sportscar = new Vehicle();  
  
        int range1, range2;  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelcap = 16;  
        minivan.mpg = 21;
```

```
// assign values to fields in sportscar
sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;

// compute the ranges assuming a full tank of gas
range1 = minivan.fuelcap * minivan.mpg;
range2 = sportscar.fuelcap * sportscar.mpg;
System.out.println("Minivan can carry " +
minivan.passengers +
" with a range of " + range1);
System.out.println("Sportscar can carry " +
sportscar.passengers +
" with a range of " + range2);
}
}
```

Creating Objects

In the previous code, an object was created from a class. Hence 'minivan' was an object which was created at run time from the 'Vehicle' class – vehicle minivan = **new** Vehicle(); This statement allocates a space in memory for the object and it also creates a reference. We can create a reference first and then create an object:

```
Vehicle minivan; // reference to object only
minivan = new Vehicle ( ); // an object is created
```

Reference Variables and Assignment

Consider the following statements:

```
Vehicle car1 = new Vehicle ( );
Vehicle car2 = car 1;
```

We have created a new instance of type Vehicle named car1. However note that car2 is **NOT** another instance of type Vehicle. car2 is the same object as car1 and has been assigned the same properties,

```
car1.mpg = 26; // sets value of mpg to 26
```

If we had to enter the following statements:

```
System.out.println(car1.mpg);
System.out.println(car2.mpg);
```

The expected output would be 26 twice, each on a separate line.

car1 and car2 are not linked. car2 can be re-assigned to another data type:

```
Vehicle car1 = new Vehicle();  
Vehicle car2 = car1;  
Vehicle car3 = new Vehicle();  
car2 = car3; // now car2 and car3 refer to the same object.
```

Methods

Methods are the functions which a particular class possesses. These functions usually use the data defined by the class itself.

```
// adding a range() method  
class Vehicle {  
    int passengers; // number of passengers  
    int fuelcap; // fuel capacity in gallons  
    int mpg; // fuel consumption in miles per gallon  
    // Display the range.  
    void range() {  
        System.out.println("Range is " + fuelcap * mpg);  
    }  
}
```

Note that 'fuelcap' and 'mpg' are called directly without the dot (.) operator. Methods take the following general form:

```
ret-type name(parameter-list ){  
    // body of method  
}
```

'ret-type' specifies the type of data returned by the method. If it does not return any value we write **void**. 'name' is the method name while the 'parameter-list' would be the values assigned to the variables of a particular method (empty if no arguments are passed).

```
class AddMeth {  
    public static void main(String args[]) {
```

```
Vehicle minivan = new Vehicle();
Vehicle sportscar = new Vehicle();
int range1, range2;
// assign values to fields in minivan
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;
// assign values to fields in sportscar
sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;
System.out.print("Minivan can carry " +
minivan.passengers + ". ");
minivan.range(); // display range of minivan
System.out.print("Sportscar can carry " +
sportscar.passengers + ". ");
sportscar.range(); // display range of sportscar.
    }
}
```

Returning from a Method

When a method is called, it will execute the statements which it encloses in its curly brackets, this is referred to what the method *returns*. However a method can be stopped from executing completely by using the **return** statement.

```
void myMeth() {
    int i;
    for(i=0; i<10; i++) {
        if(i == 5) return; // loop will stop when i = 5
        System.out.println();
    }
}
```

Hence the method will exit when it encounters the **return** statement or the **closing curly bracket '}'**

There can be more than one exit point in a method depending on particular conditions, but one must pay attention as too many exit points could render the code unstructured and the program will not function as desired (plan well your work).

```
void myMeth() {  
    // ...  
    if(done) return;  
    // ...  
    if(error) return;  
}
```

Returning a Value

Most methods return a value. You should be familiar with the **sqrt()** method which returns the square root of a number. Let us modify our range method to make it return a value:

```
// Use a return value.  
class Vehicle {  
    int passengers; // number of passengers  
    int fuelcap; // fuel capacity in gallons  
    int mpg; // fuel consumption in miles per gallon  
    // Return the range.  
    int range() {  
        return mpg * fuelcap; //returns range for a  
        particular vehicle  
    }  
}
```

Please note that now our method is no longer **void** but has **int** since it returns a value of type integer. It is important to know what type of variable a method is expected to return in order to set the parameter type correctly.

Main program:

```
class RetMeth {  
    public static void main(String args[]) {  
        Vehicle minivan = new Vehicle();  
        Vehicle sportscar = new Vehicle();  
        int range1, range2;  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelcap = 16;  
        minivan.mpg = 21;  
        // assign values to fields in sportscar  
        sportscar.passengers = 2;  
        sportscar.fuelcap = 14;  
        sportscar.mpg = 12;  
        // get the ranges  
        range1 = minivan.range();  
        range2 = sportscar.range();  
        System.out.println("Minivan can carry " +  
            minivan.passengers + " with range of " + range1 + "  
            Miles");  
        System.out.println("Sportscar can carry " +  
            sportscar.passengers + " with range of " + range2 +  
            " miles");  
    }  
}
```

Study the last two statements, can you think of a way to make them more efficient, eliminating the use of the two statements located just above them?

Methods which accept Parameters:

We can design methods which when called can accept a value/s. When a value is passed to a method it is called an **Argument**, while the variable that receives the argument is the **Parameter**.

```
// Using Parameters.
```

```
class ChkNum {
    // return true if x is even
    boolean isEven(int x) {
        if((x%2) == 0) return true;
        else return false;
    }
}

class ParmDemo {
    public static void main(String args[]) {
        ChkNum e = new ChkNum();
        if(e.isEven(10)) System.out.println("10 is even.");
        if(e.isEven(9)) System.out.println("9 is even.");
        if(e.isEven(8)) System.out.println("8 is even.");
    }
}
```

The method accepts a value of type integer, the parameter is x, while the argument would be any value passed to it

Predicted Output:

10 is even.

8 is even.

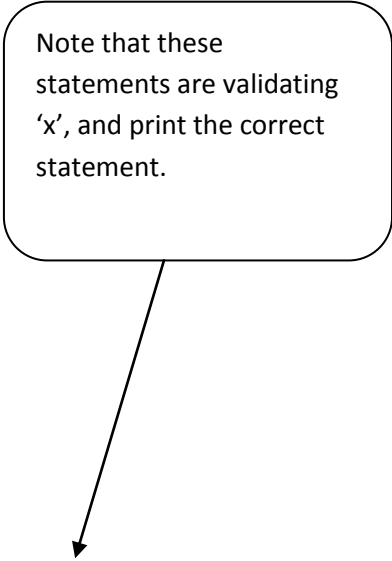
A method can accept more than one parameter. The method would be declared as follows:

```
int myMeth(int a, double b, float c) {
    // ...
}
```

The following examples illustrates this:

```
class Factor {  
    boolean isFactor(int a, int b) {  
        if( (b % a) == 0) return true;  
        else return false;  
    }  
}  
  
class IsFact {  
    public static void main(String args[]) {  
        Factor x = new Factor();  
  
        if(x.isFactor(2, 20)) System.out.println("2 is a  
        factor.");  
  
        if(x.isFactor(3, 20)) System.out.println("this won't be  
        displayed");  
    }  
}
```

Note that these statements are validating 'x', and print the correct statement.



Predicted Output:

```
2 is a factor.
```

If we refer back to our 'vehicle' example, we can now add a method which works out the fuel needed by a particular vehicle to cover a particular distance. One has to note here that the result of this method, even if it takes integers as parameters, might not be a whole number.

Therefore one has to specify that the value that the method returns, in this case we can use 'double':

```
double fuelneeded(int miles) {  
    return (double) miles / mpg;  
}
```


Updated vehicle class:

```
class Vehicle {  
    int passengers; // number of passengers  
    int fuelcap; // fuel capacity in gallons  
    int mpg; // fuel consumption in miles per gallon  
  
    // Return the range.  
    int range() {  
        return mpg * fuelcap;  
    }  
  
    // Compute fuel needed for a given distance.  
    double fuelneeded(int miles) {  
        return (double) miles / mpg;  
    }  
}
```

Main Program:

```
class CompFuel {  
    public static void main(String args[]) {  
        Vehicle minivan = new Vehicle();  
        Vehicle sportscar = new Vehicle();  
        double gallons;  
        int dist = 252;  
  
        // assign values to fields in minivan  
        minivan.passengers = 7;
```

```
minivan.fuelcap = 16;
minivan.mpg = 21;

// assign values to fields in sportscar
sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;

gallons = minivan.fuelneeded(dist);

System.out.println("To go " + dist + " miles   minivan
needs " +
gallons + " gallons of fuel.");

gallons = sportscar.fuelneeded(dist); //overwriting same
variable

System.out.println("To go " + dist + " miles sportscar
needs " +
gallons + " gallons of fuel.");
}
}
```

Predicted Output:

To go 252 miles minivan needs 12.0 gallons of fuel.

To go 252 miles sportscar needs 21.0 gallons of fuel.

🌟 Project: Creating a Help class from the Help3.java

In order to carry out this task we must examine the Help3.java and identifies ways how we can break down the code into classes and methods. The code can be broken down as follows:

1. A method which displays the 'help' text – `helpon()`
2. A method which displays the menu – `showmenu()`
3. A method which checks (validates) the entry by the user – `isvalid()`

Method `helpon()`

```
void helpon(int what) {

    switch(what) {

        case '1':

            System.out.println("The if:\n");

            System.out.println("if(condition) statement;");

            System.out.println("else statement;");

            break;

        case '2':

            System.out.println("The switch:\n");

            System.out.println("switch(expression) {");

            System.out.println(" case constant:");

            System.out.println(" statement sequence");

            System.out.println(" break;");

            System.out.println(" // ...");

            System.out.println("}");

            break;

        case '3':

            System.out.println("The for:\n");

            System.out.print("for(init; condition; iteration)");

            System.out.println(" statement;");

            break;
```

```
case '4':
    System.out.println("The while:\n");
    System.out.println("while(condition) statement;");
    break;
case '5':
    System.out.println("The do-while:\n");
    System.out.println("do {");
    System.out.println(" statement;");
    System.out.println("} while (condition);");
    break;
case '6':
    System.out.println("The break:\n");
    System.out.println("break; or break label;");
    break;
case '7':
    System.out.println("The continue:\n");
    System.out.println("continue; or continue label;");
    break;
}
System.out.println();
}
```

Method showmenu()

```
void showmenu() {
    System.out.println("Help on:");
    System.out.println(" 1. if");
    System.out.println(" 2. switch");
    System.out.println(" 3. for");
}
```

```
System.out.println(" 4. while");  
System.out.println(" 5. do-while");  
System.out.println(" 6. break");  
System.out.println(" 7. continue\n");  
System.out.print("Choose one (q to quit): ");  
  
}
```

Method isValid()

```
boolean isValid(int ch) {  
    if(ch < '1' | ch > '7' & ch != 'q') return false;  
    else return true;  
}
```

Class Help

```
class Help {  
    void helpon(int what) {  
        switch(what) {  
            case '1':  
                System.out.println("The if:\n");  
                System.out.println("if(condition) statement;");  
                System.out.println("else statement;");  
                break;  
            case '2':  
                System.out.println("The switch:\n");  
                System.out.println("switch(expression) {");  
                System.out.println(" case constant;");  
                System.out.println(" statement sequence");  
            }  
        }  
    }  
}
```

```
        System.out.println(" break;");

        System.out.println(" // ...");

        System.out.println("}");

        break;

    case '3':

        System.out.println("The for:\n");

        System.out.print("for(init; condition;
iteration)");

        System.out.println(" statement;");

        break;

    case '4':

        System.out.println("The while:\n");

        System.out.println("while(condition)
statement;");

        break;

    case '5':

        System.out.println("The do-while:\n");

        System.out.println("do {");

        System.out.println(" statement;");

        System.out.println("} while (condition);");

        break;

    case '6':

        System.out.println("The break:\n");

        System.out.println("break; or break label;");

        break;

    case '7':

        System.out.println("The continue:\n");
```

```
        System.out.println("continue; or continue  
        label;");  
  
        break;  
  
    }  
  
    System.out.println();  
  
}
```

```
void showmenu() {
```

```
    System.out.println("Help on:");  
    System.out.println(" 1. if");  
    System.out.println(" 2. switch");  
    System.out.println(" 3. for");  
    System.out.println(" 4. while");  
    System.out.println(" 5. do-while");  
    System.out.println(" 6. break");  
    System.out.println(" 7. continue\n");  
    System.out.print("Choose one (q to quit): ");  
  
}
```

```
boolean isValid(int ch) {
```

```
    if(ch < '1' | ch > '7' & ch != 'q') return false;  
    else return true;  
  
}
```

```
}
```

Main Program:

```
class HelpClassDemo {  
    public static void main(String args[])  
        throws java.io.IOException {  
        char choice;
```

```

Help hlpobj = new Help();
for(;;) {
    do {
        hlpobj.showmenu();
        do {
            choice = (char) System.in.read();
            } while(choice == '\n' | choice == '\r');
        } while( !hlpobj.isvalid(choice) );
        if(choice == 'q') break;
        System.out.println("\n");
        hlpobj.helpon(choice);
    }
}
}

```

Constructors

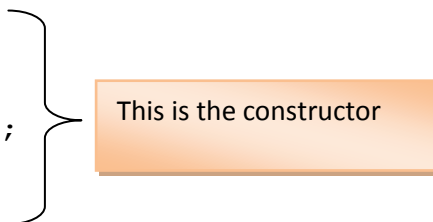
In previous examples when working with the vehicle class we did assign values to the class variables by using statements like: `minivan.passengers = 7;`

To accomplish this task Java programmers use constructors. A constructor is created by default and initializes all member variables to zero. However we can create our constructors and set the values the way we want, e.g.

```

class MyClass {
    int x;
    MyClass() {
        x = 10;
    }
}

```



This is the constructor


```
class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass();  
        System.out.println(t1.x + " " + t2.x);  
    }  
}
```

Predicted Output:

10 10

Constructor having parameters

We can edit our previous constructor to create a parameter:

```
MyClass(int i) {  
    x = i;  
}
```

If we edit the main program, by changing the statements which initiate the two objects:

```
MyClass t1 = new MyClass(10);  
MyClass t2 = new MyClass(88);
```

The output would now be:

10 88

The values 10 and 88 are first passed on to 'i' and then are assigned to 'x'.

Now we can modify our vehicle class and add a constructor:

```
// Constructor for Vehicle class  
Vehicle(int p, int f, int m) {  
    passengers = p;  
    fuelcap = f;  
    mpg = m;  
}
```

The main program would be as follows:

```
class VehConsDemo {  
    public static void main(String args[]) {  
        // construct complete vehicles  
        Vehicle minivan = new Vehicle(7, 16, 21);  
        Vehicle sportscar = new Vehicle(2, 14, 12);  
        double gallons;  
        int dist = 252;  
        gallons = minivan.fuelneeded(dist);  
        System.out.println("To go " + dist + " miles minivan  
needs " + gallons + " gallons of fuel.");  
        gallons = sportscar.fuelneeded(dist);  
        System.out.println("To go " + dist + " miles sportscar  
needs " + gallons + " gallons of fuel.");  
    }  
}
```

Overloading Methods and Constructors

The term overloading refers to the act of using the same method/constructor name in a class but different parameter declarations. Method overloading is an example of Polymorphism.

Method Overloading

```
// Demonstrate method overloading.  
  
class Overload {  
    void ovlDemo() {  
        System.out.println("No parameters");  
    }  
  
    // Overload ovlDemo for one integer parameter.  
    void ovlDemo(int a) {
```

```
        System.out.println("One parameter: " + a);
    }

    // Overload ovlDemo for two integer parameters.

    int ovlDemo(int a, int b) {

        System.out.println("Two parameters: " + a + " " +
b);

        return a + b;

    }

    // Overload ovlDemo for two double parameters.

    double ovlDemo(double a, double b) {

        System.out.println("Two double parameters: " +
a + " "+ b);

        return a + b;

    }

}
```

Main Program:

```
class OverloadDemo {

    public static void main(String args[]) {

        Overload ob = new Overload();

        int resI;

        double resD;

        // call all versions of ovlDemo()

        ob.ovlDemo();

        System.out.println();

        ob.ovlDemo(2);

        System.out.println();

        resI = ob.ovlDemo(4, 6);

        System.out.println("Result of ob.ovlDemo(4, 6): " +
```

```
        resI);  
  
        System.out.println();  
  
        resD = ob.ovlDemo(1.1, 2.32);  
  
        System.out.println("Result of ob.ovlDemo(1.1, 2.32):  
        " + resD);  
    }  
}
```

Predicted Output:

```
No parameters  
One parameter: 2  
Two parameters: 4 6  
Result of ob.ovlDemo(4, 6): 10  
Two double parameters: 1.1 2.32  
Result of ob.ovlDemo(1.1, 2.32): 3.42
```

Automatic Type Conversion for Parameters of overloaded Methods

```
class Overload2 {  
    void f(int x) {  
        System.out.println("Inside f(int): " + x);  
    }  
    void f(double x) {  
        System.out.println("Inside f(double): " + x);  
    }  
}
```

Main Program:

```
class TypeConv {
    public static void main(String args[]) {
        Overload2 ob = new Overload2();

        int i = 10;

        double d = 10.1;

        byte b = 99;

        short s = 10;

        float f = 11.5F;

        ob.f(i); // calls ob.f(int)
        ob.f(d); // calls ob.f(double)
        ob.f(b); // calls ob.f(int) - type conversion
        ob.f(s); // calls ob.f(int) - type conversion
        ob.f(f); // calls ob.f(double) - type conversion
    }
}
```

Predicted Output:

Inside f(int): 10

Inside f(double): 10.1

Inside f(int): 99

Inside f(int): 10

Inside f(double): 11.5

Even though “f” had been defined with two parameters, ‘int’ and ‘double’, it is possible to pass a different data type and automatic conversion occurs. ‘byte’ and ‘short’ are converted to ‘int’ while ‘float’ is converted to ‘double’ and the respective methods are called.

Overloading Constructors

```
// Overloading constructors.

class MyClass {
    int x;

    MyClass() {
        System.out.println("Inside MyClass().");
        x = 0;
    }

    MyClass(int i) {
        System.out.println("Inside MyClass(int).");
        x = i;
    }

    MyClass(double d) {
        System.out.println("Inside MyClass(double).");
        x = (int) d;
    }

    MyClass(int i, int j) {
        System.out.println("Inside MyClass(int, int).");
        x = i * j;
    }
}
```

Main Program:

```
class OverloadConsDemo {

    public static void main(String args[]) {

        MyClass t1 = new MyClass();

        MyClass t2 = new MyClass(88);

        MyClass t3 = new MyClass(17.23);
    }
}
```

```
MyClass t4 = new MyClass(2, 4);

System.out.println("t1.x: " + t1.x);

System.out.println("t2.x: " + t2.x);

System.out.println("t3.x: " + t3.x);

System.out.println("t4.x: " + t4.x);

    }

}
```

Predicted Output:

```
Inside MyClass().
Inside MyClass(int).
Inside MyClass(double).
Inside MyClass(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8
```

In Java programming, overloading constructors is a technique used to allow an object to initialize another. This makes coding more efficient.

```
class Summation {

    int sum;

    // Construct from an integer

    Summation(int num) {

        sum = 0;

        for(int i=1; i <= num; i++)

            sum += i;

    }

    // Construct from another object (ob)
```

```
Summation(Summation ob) {  
    sum = ob.sum;  
}  
}
```

Main Program:

```
class SumDemo {  
    public static void main(String args[]) {  
        Summation s1 = new Summation(5);  
        Summation s2 = new Summation(s1);  
        System.out.println("s1.sum: " + s1.sum);  
        System.out.println("s2.sum: " + s2.sum);  
    }  
}
```

Predicted Output:

s1.sum: 15

s2.sum: 15

In the above example, when s2 is constructed, it takes the value of the summation of s1. Hence there is no need to recompute the value.

Access Specifiers: public and private

Whenever we started a class, we always wrote 'public'. If one writes 'class' only, by default it is taken to be public.

```
// Public and private access.  
class MyClass {  
    private int alpha; // private access  
    public int beta; // public access  
    int gamma; // default access (essentially public)
```


/* Methods to access alpha. Members of a class can access a private member of the same class.

*/

```
void setAlpha(int a) {
```

```
    alpha = a;
```

```
}
```

```
int getAlpha() {
```

```
    return alpha;
```

```
}
```

```
}
```

Main Program:

```
class AccessDemo {
```

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

```
        MyClass ob = new MyClass();
```

```
        ob.setAlpha(-99);
```

```
        System.out.println("ob.alpha is " + ob.getAlpha());
```

```
// You cannot access alpha like this:
```

```
// ob.alpha = 10; // Wrong! alpha is private!
```

```
// These are OK because beta and gamma are public.
```

```
        ob.beta = 88;
```

```
        ob.gamma = 99;
```

```
    }
```

```
}
```

Another example using arrays:

```
class FailSoftArray {  
    private int a[]; // reference to array  
    private int errval; // value to return if get() fails  
    public int length; // length is public  
    /* Construct array given its size and the value to  
    return if get() fails. */  
    public FailSoftArray(int size, int errv) {  
        a = new int[size];  
        errval = errv;  
        length = size;  
    }  
    // Return value at given index.  
    public int get(int index) {  
        if(ok(index)) return a[index];  
        return errval;  
    }  
    // Put a value at an index. Return false on failure.  
    public boolean put(int index, int val) {  
        if(ok(index)) {  
            a[index] = val;  
            return true;  
        }  
        return false;  
    }  
    // Return true if index is within bounds.  
    private boolean ok(int index) {
```

```
        if(index >= 0 & index < length) return true;

        return false;

    }

}
```

Main Program:

```
class FSDemo {

    public static void main(String args[]) {

        FailSoftArray fs = new FailSoftArray(5, -1);

        int x;

        // show quiet failures

        System.out.println("Fail quietly.");

        for(int i=0; i < (fs.length * 2); i++)

            fs.put(i, i*10);

        for(int i=0; i < (fs.length * 2); i++) {

            x = fs.get(i);

            if(x != -1) System.out.print(x + " ");

        }

        System.out.println("");

        // now, handle failures

        System.out.println("\nFail with error reports.");

        for(int i=0; i < (fs.length * 2); i++)

            if(!fs.put(i, i*10))

                System.out.println("Index " + i + " out-of-bounds");

        for(int i=0; i < (fs.length * 2); i++) {

            x = fs.get(i);

            if(x != -1) System.out.print(x + " ");

            else
```

```
        System.out.println("Index " + i + " out-of-  
        bounds");  
    }  
}  
}
```

Predicted Output:

```
Fail with error reports.  
Index 5 out-of-bounds  
Index 6 out-of-bounds  
Index 7 out-of-bounds  
Index 8 out-of-bounds  
Index 9 out-of-bounds  
0 10 20 30 40 Index 5 out-of-bounds  
Index 6 out-of-bounds  
Index 7 out-of-bounds  
Index 8 out-of-bounds  
Index 9 out-of-bounds
```

Arrays and Strings

Arrays

An array can be defined as a collection of variables of the **same** type defined by a common name, e.g. an array called *Names* which stores the names of your class mates:

Names [name1, name2, name3, ... nameX]

Arrays in Java are different from arrays in other programming languages because they are implemented as objects.

One-dimensional Arrays

Declaration: **type** array-name[] = new **type**[*size*];

e.g. `int sample[] = new int[10];`

The following code creates an array of ten integers, fills it up with numbers using a loop and then prints the content of each location (index) of the array:

```
class ArrayDemo {  
    public static void main(String args[]) {  
        int sample[] = new int[10];  
        int i;  
        for(i = 0; i < 10; i = i+1)  
            sample[i] = i;  
        for(i = 0; i < 10; i = i+1)  
            System.out.println("This is sample[" + i + "]: " +  
                               sample[i]);  
    }  
}
```

Predicted Output:

This is sample[0]: 0

This is sample[1]: 1

This is sample[2]: 2

This is sample[3]: 3

This is sample[4]: 4

This is sample[5]: 5

This is sample[6]: 6

This is sample[7]: 7

This is sample[8]: 8

This is sample[9]: 9

The following program contains two loops used to identify the smallest and the largest value stored in the array:

```
class MinMax {  
    public static void main(String args[]) {  
        int nums[] = new int[10];  
        int min, max;  
        nums[0] = 99;  
        nums[1] = -10;  
        nums[2] = 100123;  
        nums[3] = 18;  
        nums[4] = -978;  
        nums[5] = 5623;  
        nums[6] = 463;  
        nums[7] = -9;  
        nums[8] = 287;  
        nums[9] = 49;  
        min = max = nums[0];  
        for(int i=1; i < 10; i++) {  
            if(nums[i] < min) min = nums[i];
```

```

        if(nums[i] > max) max = nums[i];

    }

    System.out.println("min and max: " + min + " " +
max);

}

}

```

Predicted Output:

min and max: -978 100123

Sorting an Array – The Bubble Sort

The Bubble sort is one type, the simplest, of sorting algorithms. It can be used to sort data stored in arrays but it is not ideal when the size of the array to sort is large.

```

class Bubble {

    public static void main(String args[]) {

        int nums[] = { 99, -10, 100123, 18, -978,
5623, 463, -9, 287, 49 };

        int a, b, t;

        int size;

        size = 10; // number of elements to sort

        // display original array

        System.out.print("Original array is:");

        for(int i=0; i < size; i++)

            System.out.print(" " + nums[i]);

        System.out.println();

        // This is the Bubble sort.

        for(a=1; a < size; a++)

            for(b=size-1; b >= a; b--) {

```

```

        if(nums[b-1] > nums[b]) { // if out of order
            // exchange elements

            t = nums[b-1];
            nums[b-1] = nums[b];
            nums[b] = t;
        }
    }

    // display sorted array

    System.out.print("Sorted array is:");
    for(int i=0; i < size; i++)
        System.out.print(" " + nums[i]);
    System.out.println();
}
}

```

Predicted Output:

Original array is: 99 -10 100123 18 -978 5623 463 -9 287 49

Sorted array is: -978 -10 -9 18 49 99 287 463 5623 100123

Two-Dimensional Arrays:

A two dimensional array is like a list of one-dimensional arrays. Declaration is as follows:

```
int table[][] = new int[10][20];
```

This would create a table made up of 10 rows (index: 0 to 9) and 20 columns (index: 0 to 19). The following code creates a table, 3 rows by 4 columns, and fills it up with numbers from 1 to 12. Note that at index `[0][0] = 1`, `[0][1] = 2`, `[0][2] = 3`, `[0][3] = 4`, `[1][0] = 5`, etc.

```

class TwoD {

    public static void main(String args[]) {

```



```
int t, i;

int table[][] = new int[3][4];

for(t=0; t < 3; ++t) {

    for(i=0; i < 4; ++i) {

        table[t][i] = (t*4)+i+1;

        System.out.print(table[t][i] + " ");

    }

    System.out.println();

}

}
```

Different syntax used to declare arrays:

Consider the following: `type[] var-name;`

The square brackets follow the type specifier, not the name of the array variable. For example, the following two declarations are equivalent:

```
int counter[] = new int[3];

int[] counter = new int[3];
```

The following declarations are also equivalent:

```
char table[][] = new char[3][4];

char[][] table = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time.

```
int[] nums, nums2, nums3; // create three arrays
```

This creates three array variables of type **int**. It is the same as writing:

```
int nums[], nums2[], nums3[]; // also, create three arrays
```

The alternative declaration form is also useful when specifying an array as a return type for a method:

```
int[] someMeth( ) { ...
```

This declares that `someMeth ()` returns an array of type `int`.

Array References:

Consider a particular array, `nums1`, and another array, `nums2` and at one point in the code we assign one array reference to the other, i.e. `nums2 = nums1`. Then every action on `nums2` will be as if it were on `nums1` (`nums2` reference is lost).

```
class AssignARef {
    public static void main(String args[]) {
        int i;
        int nums1[] = new int[10];
        int nums2[] = new int[10];
        for(i=0; i < 10; i++)
            nums1[i] = i;
        for(i=0; i < 10; i++)
            nums2[i] = -i;
        System.out.print("Here is nums1: ");
        for(i=0; i < 10; i++)
            System.out.print(nums1[i] + " ");
        System.out.println();
        System.out.print("Here is nums2: ");
        for(i=0; i < 10; i++)
            System.out.print(nums2[i] + " ");
        System.out.println();
        nums2 = nums1; // nums2 is now nums1
        System.out.print("Here is nums2 after assignment:");
        System.out.println();
        for(i=0; i < 10; i++)
            System.out.print(nums2[i] + " ");
        System.out.println();
    }
}
```

```
// now operate on nums1 array through nums2

    nums2[3] = 99;

    System.out.print("Here is nums1 after change through
nums2: ");

    for(i=0; i < 10; i++)

        System.out.print(nums1[i] + " ");

        System.out.println();

    }

}
```

Predicted Output:

Here is nums1: 0 1 2 3 4 5 6 7 8 9

Here is nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9

Here is nums2 after assignment: 0 1 2 3 4 5 6 7 8 9

Here is nums1 after change through nums2: 0 1 2 99 4 5 6 7 8 9

The Length Variable:

This variable automatically keeps track of the number of items an array can hold and **NOT** the actual items. When an array is declared to have ten items, even if one does not put actual data, the length of the array is 10. Something to keep in mind is also when dealing with two dimensional arrays. As already explained in the theory lessons, two dimensional arrays are considered to be an array of arrays. Hence calling up the length of such array would return the number of sub-arrays. To access the length of the particular sub-array, one has to write the 'row' index, e.g. **arrayname [0].length**

```
// length of array demo

class LengthDemo {

    public static void main(String args[]) {

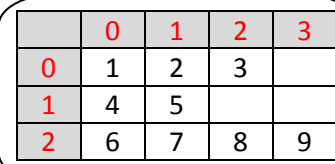
        int list[] = new int[10];

        int nums[] = { 1, 2, 3 };

    }

}
```

```
int table[][] = { // a variable-length table
    {1, 2, 3},
    {4, 5},
    {6, 7, 8, 9}
};
```



	0	1	2	3
0	1	2	3	
1	4	5		
2	6	7	8	9

```
System.out.println("length of list is " +
list.length);

System.out.println("length of nums is " +
nums.length);

System.out.println("length of table is " +
table.length); // returns number of rows

System.out.println("length of table[0] is " +
table[0].length);

System.out.println("length of table[1] is " +
table[1].length);

System.out.println("length of table[2] is " +
table[2].length);

System.out.println();

// using length to initialize list
for(int i=0; i < list.length; i++)
    list[i] = i * i;

System.out.print("Here is list: ");

// using length to display list
for(int i=0; i < list.length; i++)
    System.out.print(list[i] + " ");

System.out.println();

}

}
```

Predicted Output:

length of list is 10

length of nums is 3

length of table is 3

length of table[0] is 3

length of table[1] is 2

length of table[2] is 4

Here is list: 0 1 4 9 16 25 36 49 64 81

Using the Length to copy elements from one array to another while previously checking for size to prevent errors:

//copying an array

```
class ACopy {  
    public static void main(String args[]) {  
        int i;  
        int nums1[] = new int[10];  
        int nums2[] = new int[10];  
        for(i=0; i < nums1.length; i++)  
            nums1[i] = i;  
        if(nums2.length >= nums1.length)//size check  
            for(i = 0; i < nums2.length; i++)//limit set  
                nums2[i] = nums1[i];  
        for(i=0; i < nums2.length; i++)  
            System.out.print(nums2[i] + " ");  
    }  
}
```

Using Arrays to create a Queue data structure **

Data structures are used to organize data. Arrays can be used to create **Stacks** and **Queues**. Imagine a stack to be a pile of plates. Stacks implement a Last In First Out system (LIFO). Queues use a First In First Out order (FIFO). We can implement this as a class and also two methods to PUT and GET data to and from a queue. Therefore we put items at the end of the queue and get items from the front. Once a data item is retrieved, it cannot be retrieved again (consumptive). A queue can also be full or empty. The following code creates a noncircular queue (does not reuse locations once they are emptied).

```
class Queue {
    char q[]; // array of type char

    int putloc, getloc; // the put and get indices

    Queue(int size) {
        q = new char[size+1]; // allocate memory for queue
        putloc = getloc = 0;
    }

    // method put - places a character into the queue
    void put(char ch) {
        if(putloc==q.length-1) {
            System.out.println(" - Queue is full.");
            return;
        }
        putloc++;
        q[putloc] = ch;
    }

    // method get - get a character from the queue
    char get() {
        if(getloc == putloc) {
            System.out.println(" - Queue is empty.");
            return (char) 0;
        }
    }
}
```

```

    }

    getloc++;

    return q[getloc];

}

}

// MAIN PROGRAM

class QDemo {

    public static void main(String args[]) {

        Queue bigQ = new Queue(100);

        Queue smallQ = new Queue(4);

        char ch;

        int i;

        System.out.println("Using bigQ to store the
        alphabet.");

        // put some numbers into bigQ

        for(i=0; i < 26; i++)

            bigQ.put((char) ('A' + i));

        // retrieve and display elements from bigQ

        System.out.print("Contents of bigQ: ");

        for(i=0; i < 26; i++) {

            ch = bigQ.get();

            if(ch != (char) 0) System.out.print(ch);

        }

        System.out.println("\n");

        System.out.println("Using smallQ to generate errors.");

        // Now, use smallQ to generate some errors

        for(i=0; i < 5; i++) {

```

```
        System.out.print("Attempting to store " +
            (char) ('Z' - i));
        smallQ.put((char) ('Z' - i));
        System.out.println();
    }

    System.out.println();

    // more errors on smallQ

    System.out.print("Contents of smallQ: ");
    for(i=0; i < 5; i++) {
        ch = smallQ.get();
        if(ch != (char) 0) System.out.print(ch);
    }
}
```

Predicted Output:

```
Using bigQ to store the alphabet.
Contents of bigQ: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Using smallQ to generate errors.
Attempting to store Z
Attempting to store Y
Attempting to store X
Attempting to store W
Attempting to store V - Queue is full.
Contents of smallQ: ZYXW - Queue is empty.
```


The Enhanced 'for' Loop:

While studying loops we had mentioned the 'enhanced for loop'. This special loop was designed to cater for situations where one would need to manipulate each data item of an array. Declaration of this for loop includes an *iteration variable (itr-var)*, which is a variable that will be used to store temporarily each item from a particular collection (e.g. array, vectors, lists, sets, maps):

```
for(type itr-var : collection) statement-block
```

The type of itr-var should be similar or compatible to the type of the data items held in the array.

Traditional vs. enhanced version:

Traditional:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

Enhanced:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

for(int x: nums) sum += x;
```

x will automatically
increment

Sample program:

```
class ForEach {

    public static void main(String args[]) {

        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        int sum = 0;

        // Use for-each style for to display and sum the values.

        for(int x : nums) {

            System.out.println("Value is: " + x);

            sum += x;

        }

        System.out.println("Summation: " + sum);

    }

}
```

Predicted Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Strings

The string data type defines is used in Java to handle character strings. It is important to note that in Java, strings are Objects. Every time we write an output statement, the characters we enclose within the quotes are automatically turned into a String object,

```
System.out.println("Hello World");
```

Strings are constructed just like other objects, e.g.

```
String str = new String ("Hello");
String str2 = new String (str); //str2 = "Hello"
String str = "I love Java.";
```

Basic String example:

```
// Introduce String.
class StringDemo {
    public static void main(String args[]) {
        // declare strings in various ways
```

```
String str1 = new String("Java strings are objects.");
String str2 = "They are constructed various ways.";
String str3 = new String(str2);
System.out.println(str1);
System.out.println(str2);
System.out.println(str3);
}
}
```

Predicted Output:

```
Java strings are objects.
They are constructed various ways.
They are constructed various ways.
```

Using String Methods

```
// Some String operations.
class StrOps {
    public static void main(String args[]) {
        String str1 =
            "When it comes to Web programming, Java is #1.";
        String str2 = new String(str1);
        String str3 = "Java strings are powerful.";
        int result, idx;
        char ch;

        System.out.println("Length of str1: " +
            str1.length());

        // display str1, one char at a time.
        for(int i=0; i < str1.length(); i++)
```

```
        System.out.print(str1.charAt(i));

        System.out.println();

        if(str1.equals(str2))

            System.out.println("str1 equals str2");

        else

            System.out.println("str1 does not equal str2");

        if(str1.equals(str3))

            System.out.println("str1 equals str3");

        else

            System.out.println("str1 does not equal str3");

            result = str1.compareTo(str3);

            if(result == 0)

                System.out.println("str1 and str3 are equal");

            else if(result < 0)

                System.out.println("str1 is less than str3");

            else

                System.out.println("str1 is greater than str3");

        // assign a new string to str2

        str2 = "One Two Three One";

        idx = str2.indexOf("One");

        System.out.println("Index of first occurrence of One: " + idx);

        idx = str2.lastIndexOf("One");

        System.out.println("Index of last occurrence of One: " + idx);

    }

}
```

Predicted Output:

Length of str1: 45

When it comes to Web programming, Java is #1.

str1 equals str2

str1 does not equal str3

str1 is greater than str3

Index of first occurrence of One: 0

Index of last occurrence of One: 14

String Arrays

```
// Demonstrate String arrays.

class StringArrays {

    public static void main(String args[]) {

        String strs[] = { "This", "is", "a", "test." };

        System.out.println("Original array: ");

        for(String s : strs)

            System.out.print(s + " ");

        System.out.println("\n");

        // change a string

        strs[1] = "was";

        strs[3] = "test, too!";

        System.out.println("Modified array: ");

        for(String s : strs)

            System.out.print(s + " ");

    }

}
```

Predicted Output:

Original array:

This is a test.

Modified array:

This was a test, too!

Strings are said to be 'immutable', i.e. they cannot be changed once they are created. However we can use the method `substring()` to capture part of a string. The method is declared as follows:

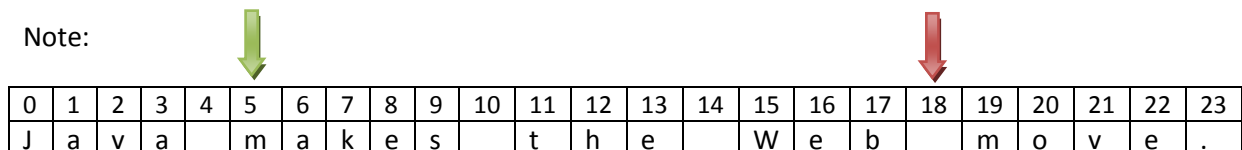
`String substring(int startIndex, int endIndex)`

Example:

`// Use substring().`

```
class SubStr {
    public static void main(String args[]) {
        String orig = "Java makes the Web move.";
        // construct a substring
        String subst = orig.substring(5, 18);
        System.out.println("Original String: " + orig);
        System.out.println("Sub String: " + subst);
    }
}
```

Note:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
J	a	v	a		m	a	k	e	s		t	h	e		W	e	b		m	o	v	e	.

Predicted Output:

Original String: Java makes the Web move.

Sub String: makes the Web



Understanding the “public static void main(String args[])” line of code – FYI only

This line really represents a method call, main, having ‘args’ (a string array) as parameter. This array stores *command-line arguments* (CLI arguments), which is any information which follows the program name. Consider the following examples:

Example 1:

```
// Display all command-line information.

class CLDemo {

    public static void main(String args[]) {

        System.out.println("There are " + args.length +

            " command-line arguments.");

        System.out.println("They are: ");

        for(int i=0; i<args.length; i++)

            System.out.println("arg[" + i + "]: " + args[i]);

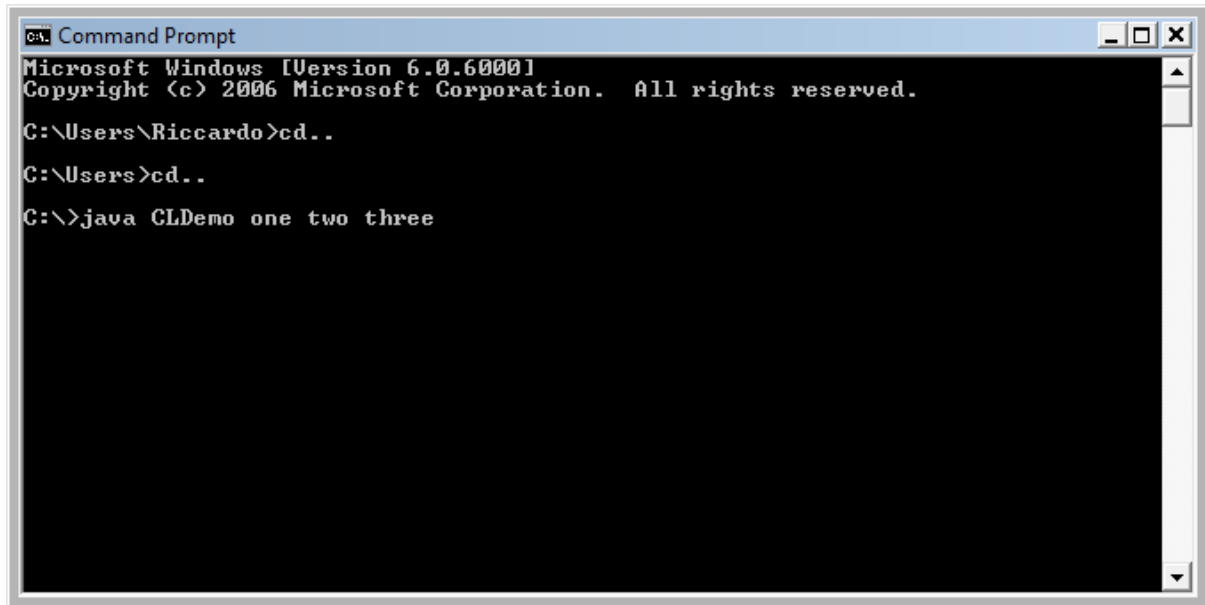
    }

}
```

If we execute this program in a DOS shell³ (not via IDE), we use the following syntax:

```
C:\> java CLDemo one two three (press enter)
```

³ Put your file (*.class, after compilation) in root drive. Then Start > Run > cmd



```

C:\> Command Prompt
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\Riccardo>cd..
C:\Users>cd..
C:\>java CLDemo one two three
  
```

The following output will be generated:

There are 3 command-line arguments.

They are:

arg[0]: one

arg[1]: two

arg[2]: three

Example 2:

// A simple automated telephone directory.

```

class Phone {

    public static void main(String args[]) {

        String numbers[][] = {

            { "Tom", "555-3322" },

            { "Mary", "555-8976" },

            { "Jon", "555-1037" },

            { "Rachel", "555-1400" }

        };

        int i;

        if(args.length != 1)
  
```



```
        System.out.println("Usage: java Phone <name>");
    else {
        for(i=0; i<numbers.length; i++) {
            if(numbers[i][0].equals(args[0])) {
                System.out.println(numbers[i][0] + ": " +
                    numbers[i][1]);
                break;
            }
        }
        if(i == numbers.length)
            System.out.println("Name not found.");
    }
}
```

Execute the code as follows (after compilation):

```
C>java Phone Mary
```

```
Mary: 555-8976
```

Vector and ArrayList

A vector can be defined simply as an Array which can 'grow'. Nowadays it has been replaced by ArrayList. The following code snippets are examples of implementing Vectors:

```
// beware Vector's List-compatible set method with the
parameters reversed:

vector.set( i, object );

// yet in the old form:

vector.setElementAt( object, i );

// get has two forms, the new List-compatible:

Object o = vector.get( i );

// or the old form:

Object o = vector.elementAt( i );

// there is no such thing as:

Object o = vector.getElementAt( i );

// similarly add has two forms: the new list-compatible:

vector.add ( o );

// or the old form:

vector.addElement( o );
```

[Further details can be obtained by referring to text book]

As already stated in the previous page ArrayList have replaced Vector in versions of Java following 1.1. ArrayList are much faster than Vector. You can add items to an ArrayList either at a particular index, 'i', or simply at the end of the list. The methods used are as follows:

```
// adding an element to the middle of a list

arrayList.add( i, object ); //i = index

// adding an element to the end of a list

arrayList.add( object );
```

If one tries to add an element to a list, and this operation results in an error one would get an *ArrayIndexOutOfBoundsException*. Possible sources of error include:

- Using a negative index.
- Indexing past the current end of the ArrayList with get or set.

- Doing a `lastIndexOf` starting out past the end.

Code snippets used to remove items from an `ArrayList`:

```
// Removing item at particular index
```

```
arrayList.remove( i, object );
```

To remove an item at an unknown index, one must first search for it (better if list is sorted). To remove elements from the entire list one must work backwards since the list shrinks as we go.

```
// removing empty Strings.
```

```
for ( int i=arrayList.size(); i>=0; i-- )    {  
    String element = (String)arrayList.get(i);  
    if ( element == null || element.length() == 0 )  
    {  
        arrayList.remove(i);  
    }  
else  
    {  
        break;  
    }  
}
```

To remove undesirable elements from the head end of a list one has to work forwards without incrementing, always working on element 0. The list shrinks as we go.

```
// removing empty Strings.
```

```
while ( arrayList.size() > 0 )  
{  
    String element = (String)arrayList.get(0);  
    if ( element == null || element.length() == 0 )  
    {  
        arrayList.remove(0);  
    }  
}
```

```

    }
else
    {
        break;
    }
}

```

To delete the last 'n' elements from the list one can use the following:

```
arrayList.setSize( arrayList.size() - n );
```

Or to delete a portion of it:

```
arrayList.subList( from, to ).clear();
```

Once all the unwanted items have been removed, unless the list will grow again, it is best to use the method `ArrayList.trimToSize()`. One has to note that some programmers prefer to convert ArrayLists into arrays to perform certain functions as it is much faster. Once done the arrays can be converted back to ArrayLists.

Sample code:

```

import java.util.ArrayList;
public class AraryListDemo {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        System.out.print("Initial size of al : " + al.size());
        System.out.print("\n");

        //add.elements to the array list
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1,"A2");//inserts "A2" into array at index 1
    }
}

```

```
System.out.print("size of al after " + al.size());
System.out.print("\n");

//display the array list
System.out.print("contents of al: " + al );
System.out.print("\n");

//Remove elements from the array list
al.remove("F");
al.remove(2);
System.out.print("size after deletions : " + al.size());
System.out.print("\n");
System.out.print("contents of al:" + al);
}
}
```

Predicted Output:

```
Initial size of al:  0
size of al after 7
contents of al: [C, A2, A, E, B, D, F]
size after deletions : 5
contents of al:[C, A2, E, B, D]
```

Sorting Collections using Comparable Interface (collection of methods):

Employee.java

```
public class Employee implements Comparable {

    int EmpID;
    String Ename;
    double Sal;
    static int i;

    public Employee() {
        EmpID = i++;
        Ename = "dont know";
        Sal = 0.0;
    }

    public Employee(String ename, double sal) {
```

```
        EmpID = i++;
        Ename = ename;
        Sal = sal;
    }

    public String toString() {
        return "EmpID " + EmpID + "\n" + "Ename " + Ename + "\n" + "Sal" + Sal;
    }

    public int compareTo(Object o1) {
        if (this.Sal == ((Employee) o1).Sal)
            return 0;
        else if ((this.Sal) > ((Employee) o1).Sal)
            return 1;
        else
            return -1;
    }
}
```

ComparableDemo.java

```
import java.util.*;

public class ComparableDemo{

    public static void main(String[] args) {

List ts1 = new ArrayList();
        ts1.add(new Employee ("Tom",40000.00));
        ts1.add(new Employee ("Harry",20000.00));
        ts1.add(new Employee ("Maggie",50000.00));
        ts1.add(new Employee ("Chris",70000.00));
        Collections.sort(ts1);
        Iterator itr = ts1.iterator();

        while(itr.hasNext()){
            Object element = itr.next();
            System.out.println(element + "\n");
        }

    }
}
```

Predicted Output:

EmpID 1

Ename Harry

Sal20000.0

EmpID 0

Ename Tom

Sal40000.0

EmpID 2

Ename Maggie

Sal50000.0

EmpID 3

Ename Chris

Sal70000.0

The following is another example of sorting. Please note that this code is being presented for reference. The term package is used to group related pieces of a program together. All the related classes will be stored in a sort of folder:

```
package test;

import java.util.*;

public class Farmer implements Comparable {

    String name;

    int age;

    long income;

    public Farmer(String name, int age) {
```

```
        this.name = name;

        this.age = age;
    }

    public Farmer(String name, int age, long income)
    {
        this.name = name;
        this.age = age;
        this.income=income;
    }

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }

    public String toString()
    {
        return name + " : " + age;
    }

    // natural order for this class

    public int compareTo(Object o)
```



```
{
    return getName().compareTo(((Farmer)o).getName());
}

static class AgeComparator implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        Farmer p1 = (Farmer)o1;
        Farmer p2 = (Farmer)o2;
        if(p1.getIncome()==0 && p2.getIncome()==0 )
            return p1.getAge() - p2.getAge();
        else
            return (int) (p1.getIncome() -
p2.getIncome());
    }
}

public static void main(String[] args)
{
    List farmer = new ArrayList();
    farmer.add( new Farmer("Joe", 34) );
    farmer.add( new Farmer("Ali", 13) );
    farmer.add( new Farmer("Mark", 25) );
    farmer.add( new Farmer("Dana", 66) );

    Collections.sort(farmer);
}
```

```
System.out.println("Sort in Natural order");

System.out.println("t" + farmer);


Collections.sort(farmer,
Collections.reverseOrder());

System.out.println("Sort by reverse natural order");

System.out.println("t" + farmer);


List farmerIncome = new ArrayList();
farmerIncome.add( new Farmer("Joe", 34,33));
farmerIncome.add( new Farmer("Ali", 13,3));
farmerIncome.add( new Farmer("Mark", 25,666));
farmerIncome.add( new Farmer("Dana", 66,2));


Collections.sort(farmer, new AgeComparator());

System.out.println("Sort using Age Comparator");

System.out.println("t" + farmer);


Collections.sort(farmerIncome, new AgeComparator());

System.out.println("Sort using Age Comparator But
Income Wise");

System.out.println("t" + farmerIncome);

}

public long getIncome() {

    return income;

}
```

```

    public void setIncome(long income) {
        this.income = income;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

Predicted Output:

Sort in Natural order

```
[Ali : 13, Dana : 66, Joe : 34, Mark : 25]
```

Sort by reverse natural order

```
[Mark : 25, Joe : 34, Dana : 66, Ali : 13]
```

Sort using Age Comparator

```
[Ali : 13, Mark : 25, Joe : 34, Dana : 66]
```

Sort using Age Comparator But Income Wise

```
[Joe : 34, Ali : 13, Mark : 25, Dana : 66]
```

The following simple example utilizes the Scanner⁴ for input:

```

//import package containing scanner
import java.util.*;

```

⁴ Scanner is part of the java.util package and can be used for input (keyboard/file)

```
//read an integer and return it to user
public class Scan {
    public static void main (String args[]){
        //creating instance
        Scanner kb = new Scanner(System.in);
        System.out.println("Enter a number: ");
        //read integer
        int x = kb.nextInt();
        System.out.println("Number: " + x);
    }
}
```

Using the scanner to capture text (string variable) from keyboard:

```
import java.util.*;

public class Alphabetize {

    public static void main(String[] args) {
        //... Declare variables.
        Scanner in = new Scanner(System.in);
        ArrayList<String> words = new ArrayList<String>();

        //... Read input one word at a time.
        System.out.println("Enter words. End with EOF (CTRL-Z
then Enter)");

        //... Read input one word at a time, adding it to an
array list, hasNext to read more than one word
```

```
while (in.hasNext()) {  
    words.add(in.next());  
}  
  
//... Sort the words.  
Collections.sort(words);  
  
//... Print the sorted list.  
System.out.println("\n\nSorted words\n");  
for (String word : words) {  
    System.out.println(word);  
}  
}  
}
```

Using the Scanner is the most suggested method compared to the Keyboard Class or the System.in.read . Remember that the Keyboard class was created by you (or given) and is not a standard in Java.

File Operations in Java

Through file handling, we can read data from and write data to files besides doing all sorts of other operations. Java provides a number of methods for file handling through different classes which are a part of the “java.io” package. The question can arise in the mind of a new programmer as to why file-handling is required. The answer of this question would be in two parts, why do we need to read data from the files and why do we need to save it (write it) to a file.

To answer the first part, Let us suppose that we have a very large amount of data which needs to be input into a program, Something like a 1000 records, If we start inputting the data manually and while we are in half-way through the process, there is a power-failure, then once power is restored, the entire data has to be input again. This would mean a lot of extra work, an easier approach would be to write all the records in a file and save the file after writing 10 or so records, in this case even if there is a power-failure, only some records would be lost and once power is restored, there would be only a few records that would need to be input again. Once all the records are saved in that file, the file-name can be passed to the program, which can then read all the records from the file. For the second part, consider a system which needs to record the time and name of any error that occurs in the system, this can be achieved through saving the data into a file and the administrator can view the file any time he/she wishes to view it.

Note that if you use a “/” in a path definition in Java in Windows, the path would still resolve correctly and if you use the Windows conventional “\”, then you have to place two forward slashes “\\” as a single “\” would be taken as an escape-sequence.

```
import java.io.*;

public class streams
{
    public static void main(String []args)
    {
        File f1=new File("Folder/FILE");
        File f2=new File("Folder/FILE1");

        String s;

        if(f1.exists())
```

```
{
    if(f1.isFile())
    {
        System.out.println("File Name is
"+f1.getName());

        s=f1.getParent();

        File f3=new File(s);

        f1.renameTo(new File("Folder/abc"));

        f2.delete();

        if (f3.isDirectory())
        {
            System.out.println(f2.getPath());
        }
    }
    else
    {
        System.out.println("Not a File");
    }
}
}
```

The output of the program is:

File Name is FILE

Folder

If successfully run , the " FILE " file inside the folder " Folder " will be renamed to " abc " and the " FILE1 " file will be deleted.

Here is an example of a program that reads its own first six bytes, we have:

```
//0123
```

```
import java.io.*;
```

```
public class read
```

```
{
```

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

```
    {
```

```
        int s=6;
```

```
        int b[]=new int[6];
```

```
        char c[]=new char[6];
```

```
        try
```

```
        {
```

```
            FileInputStream f = new  
FileInputStream("read.java");
```

```
            for (int i=0; i<6; i++)
```

```
            {
```

```
                b[i] = f.read();
```

```
                c[i] = (char) b[i];
```

```
            }
```

```
            System.out.println("First 6 bytes of the file  
are :");
```

```
            for (int i=0;i<6;i++)
```



```

        System.out.print(b[i]+" ");

        System.out.println("\nFirst 6 Bytes as
characters  :");

        for (int i=0;i<6;i++)

            System.out.print(c[i]);

    }

    catch (Exception e)

    {

        System.out.println("Error");

    }

}

}

```

Predicted Ouptut:

First 6 bytes of the file are :

47 47 48 49 50 51

First 6 Bytes as characters are :

//0123

Notice that the `FileInputStream` object is created inside a try-catch block since if the specified-file does not exist, an exception is raised. In the same way to write data to a file byte-by-byte, we have:

```
import java.io.*;
```

```
public class writer
```

```
{
```

```
public static void main(String []args) throws IOException
{
    String s="Hello";

    byte b[]=s.getBytes();

    FileOutputStream f=new FileOutputStream("file.txt");

    int i=0;
    while(i < b.length)
    {
        f.write(b[i]);
        i++;
    }
}
```

If the file called file.txt does not exist, it is automatically created. If we place a true in the constructor for the FileOutputStream, then the file would be opened in append mode.

Template to read data from disk

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
class _____ {
    public static void main(String args[])
        throws FileNotFoundException {
        Scanner diskScanner = new Scanner(new
        File("_____"));
    }
}
```

```
        _____ = diskScanner.nextInt();  
        _____ = diskScanner.nextDouble();  
        _____ = diskScanner.nextLine();  
        _____ = diskScanner.findInLine(".").charAt(0);  
        // etc  
    }  
}
```

The following program reads item from a file on disk. You have to create the file using a text editor (MS Notepad), and save the file in the same location as your classes. You can use the following examples for the riddles file (save the file as riddles.txt):

What is black and white and red all over?

An embarrassed zebra

What is black and white and read all over?

A newspaper

What other word can be made with the letters of ALGORITHM?

LOGARITHM

Program Code:

```
public class Riddle{  
    private String question;  
    private String answer;  
    public Riddle(String q, String a) {  
        question = q;  
        answer = a;  
    }  
  
    public String getQuestion() {
```

```
        return question;
    }

    public String getAnswer(){
        return answer;
    }
}
```

Main Program:

```
import java.io.*;
import java.util.Scanner;

public class RiddleFileReader
{
    private Scanner fileScan; // For file input
    private Scanner kbScan;   // For keyboard input

    public RiddleFileReader(String fName)
    {
        kbScan = new Scanner(System.in);

        try
        {
            File theFile = new File(fName);
            fileScan = new Scanner(theFile);
            fileScan = fileScan.useDelimiter("\r\n");
        } catch (IOException e)
        {
            e.printStackTrace();
        } // catch()
    } // RiddleFileReader() constructor

    public Riddle readRiddle()
    {
        String ques = null;
        String ans = null;
        Riddle theRiddle = null;
    }
}
```

```

if (fileScan.hasNext())
    ques = fileScan.next();
if (fileScan.hasNext())
{
    ans = fileScan.next();
    theRiddle = new Riddle(ques, ans);
} // if
return theRiddle;
} // readRiddle()

public void displayRiddle(Riddle aRiddle)
{
    System.out.println(aRiddle.getQuestion());
    System.out.print("Input any letter to see answer:");
    String str = kbScan .next(); // Ignore KB input
    System.out.println(aRiddle.getAnswer());
    System.out.println();
} // displayRiddle()

public static void main(String[] args)
{
    RiddleFileReader rfr =
        new RiddleFileReader("riddles.txt");
    Riddle riddle = rfr.readRiddle();
    while (riddle != null)
    {
        rfr.displayRiddle(riddle);
        riddle = rfr.readRiddle();
    }
}
}

```

Template to write (save) data to disk

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
class _____ {
    public static void main(String args[])
        throws FileNotFoundException {
        PrintStream diskWriter =
            new PrintStream("_____");
        diskWriter.print(_____);
        diskWriter.println(_____);
        // Etc.
    }
}
```

Introduction to GUI using AWT/Swing

Java GUIs were built with components from the [Abstract Window Toolkit \(AWT\)](#) in package `java.awt`. When a Java application with an AWT GUI executes on different Java platforms, the application's GUI components display differently on each platform. Consider an application that displays an object of type `Button` (package `java.awt`). On a computer running the Microsoft Windows operating system, the `Button` will have the same appearance as the buttons in other Windows applications. Similarly, on a computer running the Apple Mac OS X operating system, the `Button` will have the same look and feel as the buttons in other Macintosh applications.

Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform's custom look-and-feel. An application can even change the look-and-feel during execution to enable users to choose their own preferred look-and-feel. Swing components are implemented in Java, so they are more portable and flexible than the original Java GUI components from package `java.awt`, which were based on the GUI components of the underlying platform. For this reason, Swing GUI components are generally preferred.

Swing advantages:

- Swing is faster.
- Swing is more complete.
- Swing is being actively improved.

AWT advantages:

- AWT is supported on older, as well as newer, browsers so Applets written in AWT will run on more browsers.
- The Java Micro-Edition, which is used for phones, TV set top boxes, PDAs, etc, uses AWT, not Swing.

Using Swing to create a small Window

[Refer to the code on the next page]

1. First we have to import all classes in the `javax.swing` package, although we use only the `JFrame` class in the following example. "Windows" are implemented by the `JFrame` class.
2. Make the application quit when the close box is clicked.
3. After the window has been constructed in memory, display it on the screen. The `setVisible` call also starts a separate thread to monitor user interaction with the interface.
4. When we are finished setting up and displaying the window, don't call `System.exit(0)`. We don't want to stop the program. Although `main` returns, execution continues because the call to `setVisible(true)` created another execution thread, A GUI program builds the user interface, then just "goes to sleep" until the user does something.

```
import javax.swing.*;

class FirstWindow {

    public static void main(String[] args) {

        JFrame window = new JFrame();

        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);

    }

}
```

The window created can be resized and dragged around. One can also minimize, maximize or close the window. Now that we created a window we can set the text which appears in the title bar:

```
import javax.swing.*;

class MyWindow2 extends JFrame {

    public static void main(String[] args) {

        MyWindow2 window = new MyWindow2();

        window.setVisible(true);

    }

    public MyWindow2() {    //constructor

        setTitle("My Window Title using JFrame Subclass");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }

}
```

Inserting Text inside Window

```
import java.awt.*;           // required for FlowLayout.
import javax.swing.*;

class MyWindow3 extends JFrame {

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



```
MyWindow3 window = new MyWindow3();

window.setVisible(true);

}

public MyWindow3 () {

    //Create content panel and set layout

    JPanel content = new JPanel();

    content.setLayout(new FlowLayout());

    //... Add one label to the content pane.

    JLabel greeting = new JLabel("Hello World.");

    content.add(greeting);

    //... Set window (JFrame) characteristics

    setContentPane(content);

    pack();

    setTitle("MyWindow using JFrame Subclass");

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    setLocationRelativeTo(null); // Centre window.

}

}
```

Creating a simple application implementing JButton, JTextfield and JLabel

```
import java.awt.*;

import javax.swing.*;

class DogYears extends JFrame {

    private JTextField _humanYearsTF = new JTextField(3);

    private JTextField _dogYearsTF    = new JTextField(3);
```

```

public DogYears() {

    JButton convertBtn = new JButton("Convert");

    JPanel content = new JPanel();
    content.setLayout(new FlowLayout());

    content.add(new JLabel("Dog Years"));

    content.add(_dogYearsTF);

    content.add(convertBtn);

    content.add(new JLabel("Human Years"));

    content.add(_humanYearsTF);

    setContentPane(content); //window attributes

    pack();

    setTitle("Dog Year Converter");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    setLocationRelativeTo(null);

}

public static void main(String[] args) { //main

    DogYears window = new DogYears();

    window.setVisible(true);

}
}

```

The final touch to our application is to set the action to perform while the user interacts with the application:

```

import java.awt.*;

import javax.swing.*;

import java.awt.event.*; // Needed for ActionListener

class DogYears2 extends JFrame {

    private static final int DOG_YEARS_PER_HUMAN_YEAR = 7;

    private JTextField _humanYearsTF = new JTextField(3);
    private JTextField _dogYearsTF = new JTextField(3);

```

```

public DogYears2() {
    JButton convertBtn = new JButton("Convert");
    convertBtn.addActionListener(new ConvertBtnListener());
    _dogYearsTF.addActionListener(new ConvertBtnListener());
    _humanYearsTF.setEditable(false);

    JPanel content = new JPanel();
    content.setLayout(new FlowLayout());

    content.add(new JLabel("Dog Years"));
    content.add(_dogYearsTF);
    content.add(convertBtn);
    content.add(new JLabel("Human Years"));
    content.add(_humanYearsTF);
    setContentPane(content);
    pack();
    setTitle("Dog Year Converter");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null); // Center window.
}

class ConvertBtnListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String dyStr = _dogYearsTF.getText();
        int dogYears = Integer.parseInt(dyStr);
        int humanYears = dogYears *
            DOG_YEARS_PER_HUMAN_YEAR;
    }
}

```

```
        _humanYearsTF.setText("" + humanYears);  
    }  
}  
  
public static void main(String[] args) {  
    DogYears2 window = new DogYears2();  
    window.setVisible(true);  
}  
}
```

Predicted Final Output:

