Preca College

Java Programming Language Notes

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.

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**

  Below is a theoretical explanation of inheritance with real-life examples. For detailed explanation on this topic refer inheritance with examples.

  Inheritance is the mechanism by which an object acquires the some/all properties of another object. It supports the concept of hierarchical classification.

  For example: Car is a classification of Four Wheeler. Here Car acquires the properties of a four-wheeler. Other classifications could be a jeep, tempo, van etc. Four Wheeler defines a class of vehicles that have four wheels, and specific range of engine power, load carrying capacity etc. Car (termed as a sub-class) acquires these properties from Four Wheeler (termed as a super-class), and has some specific properties, which are different from other classifications of Four Wheeler, such as luxury, comfort, shape, size, usage etc.

  A car can have further classification such as an open car, small car, big car etc, which will acquire the properties from both Four Wheeler and Car, but will still have some specific properties. This way the level of hierarchy can be extended to any level.

- Modularity

  Modularity is the degree to which a system's components are made up of relatively independent components or parts which can be combined. Modularity refers to the concept of making multiple modules first and then linking and combining them to form a complete system. Modularity enables re-usability and minimizes duplication.

  In addition to re-usability, modularity also makes it easier to fix problems as bugs can be traced to specific system modules, thus limiting the scope of detailed error searching.

- Polymorphism

  Below is a real-life example of polymorphism. Polymorphism means to process objects differently based on their data type.

  In other words it means, one method with multiple implementation, for a certain class of action. And which implementation to be used is decided at runtime depending upon the situation (i.e., data type of the object)

  Lets us look at same example of a car. A car have a gear transmission system. It has four front gears and one backward gear. When the engine is accelerated then depending upon which gear is engaged different amount power and movement is delivered to the car.

  Polymorphism could be static and dynamic both. Overloading is static polymorphism while, overriding is dynamic polymorphism.

  Overloading in simple words means two methods having same method name but takes different input parameters. This called static because, which method to be invoked will be decided at the time of compilation

  Overriding means a derived class is implementing a method of its super class.

- Encapsulation (binding code and its data)

  Below is a real-life example of encapsulation.

  Encapsulation is:

  Binding the data with the code that manipulates it. It keeps the data and the code safe from external interference

Looking at the example of a power steering mechanism of a car. Power steering of a car is a complex system, which internally have lots of components tightly coupled together, they work synchronously to turn the car in the desired direction. It even controls the power delivered by the engine to the steering wheel. But to the external world there is only one interface is available and rest of the complexity is hidden. Moreover, the steering unit in itself is complete and independent. It does not affect the functioning of any other mechanism.

Similarly, same concept of encapsulation can be applied to code. Encapsulated code should have following characteristics:

- Everyone knows how to access it.
- Can be easily used regardless of implementation details.
- There shouldn't any side effects of the code, to the rest of the application.

The idea of encapsulation is to keep classes separated and prevent them from having tightly coupled with each other.

My first Java program

We are going to use the Netbeans IDE for our Java Applications. Open the NetBeans IDE.

Create a new Project named New Project. During this process choose the file option Java Application and namethe class Example1.

Then write the following code lines:

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

 public class Example1 {

public static void main (String args []) {

System.out.println ("My first Java program");

    }

 }
```

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

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). Create a second file named Example2 in the same Project and write the following command lines:

```
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. var2 = var1/2. The words int refer to a particular data type, i.e. integer (whole numbers).

Exercise 1:

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)

```
public 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 mean 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 represents codes or characters which cannot be directly accessible through a keyboard:

Code	Description
\n	New Line
\t	Tab
\\	Backslash
\’	Single Quotation Mark
\”	Double Quotation Mark

```
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

Exercise 2:

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

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 Example7 {

 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

var = var / 4;  // now, divide both by 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.

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();

        }
```

}

s

Predicted Output:

i does not equal to zero

j/i is 2

Exercise 14

Write a program which can be used to display a conversion table, Metres to Kilometres.

Hints:

- One variable is required
- You need a 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.");
```

```
        }
}
```

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");
         }
```

```
}
```

Predicted Output:

2 is a factor.

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.

As such, the java.lang.Math class provides us access to many of these functions. The table below lists some of the more common among these.

Additionally, the java.lang.Math class provides two very frequently used constants that you might recognize:

Constant	Type	Value
`Math.PI`	`double`	`3.14159265358979323846`
`Math.E`	`double`	`2.718281828459045235`

Method	Returns
`Math.sqrt(x)`	$\sqrt{x}$
`Math.pow(x,y)`	x^y
`Math.sin(x)`	$\sin x$
`Math.cos(x)`	$\cos x$
`Math.tan(x)`	$\tan x$
`Math.asin(x)`	$\sin^{-1} x$
`Math.acos(x)`	$\cos^{-1} x$
`Math.atan(x)`	$\tan^{-1} x$
`Math.exp(x)`	e^x
`Math.log(x)`	$\ln x$
`Math.log10(x)`	$\log_{10} x$
`Math.round(x)`	the closest integer to x, as a `long`
`Math.abs(x)`	$\lvert x \rvert$
`Math.max(x,y)`	The maximum of the two values
`Math.min(x,y)`	The minimum of the two values

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 assing 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 of you 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 runtime. 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 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).

Numeric and String Methods

Method	*Returns*
`int nextInt()`	Returns the next token as an `int`. If the next token is not an integer, `InputMismatchException` is thrown.
`long nextLong()`	Returns the next token as a `long`. If the next token is not an integer, `InputMismatchException` is thrown.
`float nextFloat()`	Returns the next token as a `float`. If the next token is not a float or is out of range, `InputMismatchException` is thrown.
`double nextDouble()`	Returns the next token as a `long`. If the next token is not a float or is out of range, `InputMismatchException` is thrown.
`String next()`	Finds and returns the next complete token from this scanner and returns it as a string; a token is usually ended by whitespace such as a blank or line break. If not token exists, `NoSuchElementException` is thrown.
`String nextLine()`	Returns the rest of the current line, excluding any line separator at the end.
`void close()`	Closes the scanner.

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).

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;
                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 is: " + min + "max is:  " + max);
        }
}
```

Predicted Output:

min is: -978 maxis: 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 woth 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();

		}

	}

}
```