**This study material is intended for reference only. Text books must be followed thoroughly**.

**JAVA BASIC OPERATORS**

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

* Arithmetic Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Assignment Operators
* Misc Operators

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition - Adds values on either side of the operator | A + B will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | A - B will give -10 |
| \* | Multiplication - Multiplies values on either side of the operator | A \* B will give 200 |
| / | Division - Divides left hand operand by right hand operand | B / A will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | B % A will give 0 |
| ++ | Increment - Increases the value of operand by 1 | B++ gives 21 |
| -- | Decrement - Decreases the value of operand by 1 | B-- gives 19 |

The Relational Operators:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20, then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 1111 |
| >>> | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. | A >>>2 will give 15 which is 0000 1111 |

The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

The Assignment Operators:

There are following assignment operators supported by Java language:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is same as C = C | 2 |

Misc Operators

There are few other operators supported by Java Language.

Conditional Operator ( ? : ):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

variable x = (expression) ? value if true : value if false

Following is the example:

public class Test {

public static void main(String args[]){

int a , b;

a = 10;

b = (a == 1) ? 20: 30;

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

b = (a == 10) ? 20: 30;

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

}

}

This would produce the following result:

Value of b is : 30

Value of b is : 20

instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is wriiten as:

( Object reference variable ) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

public class Test {

public static void main(String args[]){

String name = "James";

// following will return true since name is type of String

boolean result = name instanceof String;

System.out.println( result );

}

}

This would produce the following result:

true

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

class Vehicle {}

public class Car extends Vehicle {

public static void main(String args[]){

Vehicle a = new Car();

boolean result = a instanceof Car;

System.out.println( result );

}

}

This would produce the following result:

true

**Precedence of Java Operators:**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 \* 2; here x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] . (dot operator) | Left to right |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %= >>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

**JAVA LOOP CONTROL**

There may be a situation when we need to execute a block of code several number of times, and is often referred to as a loop.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

* while Loop
* do...while Loop
* for Loop

## The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

## Syntax:

The syntax of a while loop is:

while(Boolean\_expression)

{

//Statements

}

When executing, if the *boolean\_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

## Example:

public class Test {

public static void main(String args[]) {

int x = 10;

while( x < 20 ) {

System.out.print("value of x : " + x );

x++;

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

}

}

}

This would produce the following result:

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

## The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

## Syntax:

The syntax of a do...while loop is:

do

{

//Statements

}while(Boolean\_expression);

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

## Example:

public class Test {

public static void main(String args[]){

int x = 10;

do{

System.out.print("value of x : " + x );

x++;

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

}while( x < 20 );

}

}

This would produce the following result:

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

## The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

## Syntax:

The syntax of a for loop is:

for(initialization; Boolean\_expression; update)

{

//Statements

}

Here is the flow of control in a for loop:

* The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
* Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
* After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
* The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

## Example:

public class Test {

public static void main(String args[]) {

for(int x = 10; x < 20; x = x+1) {

System.out.print("value of x : " + x );

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

}

}

}

This would produce the following result:

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

## Enhanced for loop in Java:

As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.

## Syntax:

The syntax of enhanced for loop is:

for(declaration : expression)

{

//Statements

}

* **Declaration:** The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
* **Expression:** This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

## Example:

public class Test {

public static void main(String args[]){

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ){

System.out.print( x );

System.out.print(",");

}

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

String [] names ={"James", "Larry", "Tom", "Lacy"};

for( String name : names ) {

System.out.print( name );

System.out.print(",");

}

}

}

This would produce the following result:

10,20,30,40,50,

James,Larry,Tom,Lacy,

## The break Keyword:

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

## Syntax:

The syntax of a break is a single statement inside any loop:

break;

## Example:

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ) {

if( x == 30 ) {

break;

}

System.out.print( x );

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

}

}

}

This would produce the following result:

10

20

## The continue Keyword:

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

* In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
* In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

## Syntax:

The syntax of a continue is a single statement inside any loop:

continue;

## Example:

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ) {

if( x == 30 ) {

continue;

}

System.out.print( x );

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

}

}

}

This would produce the following result:

10

20

40

50

## JAVA DECISION MAKING

There are two types of decision making statements in Java. They are:

* if statements
* switch statements

## The if Statement:

An if statement consists of a Boolean expression followed by one or more statements.

## Syntax:

The syntax of an if statement is:

if(Boolean\_expression)

{

//Statements will execute if the Boolean expression is true

}

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.

## Example:

public class Test {

public static void main(String args[]){

int x = 10;

if( x < 20 ){

System.out.print("This is if statement");

}

}

}

This would produce the following result:

This is if statement

## The if...else Statement:

An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

## Syntax:

The syntax of an if...else is:

if(Boolean\_expression){

//Executes when the Boolean expression is true

}else{

//Executes when the Boolean expression is false

}

## Example:

public class Test {

public static void main(String args[]){

int x = 30;

if( x < 20 ){

System.out.print("This is if statement");

}else{

System.out.print("This is else statement");

}

}

}

This would produce the following result:

This is else statement

## The if...else if...else Statement:

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

* An if can have zero or one else's and it must come after any else if's.
* An if can have zero to many else if's and they must come before the else.
* Once an else if succeeds, none of the remaining else if's or else's will be tested.

## Syntax:

The syntax of an if...else is:

if(Boolean\_expression 1){

//Executes when the Boolean expression 1 is true

}else if(Boolean\_expression 2){

//Executes when the Boolean expression 2 is true

}else if(Boolean\_expression 3){

//Executes when the Boolean expression 3 is true

}else {

//Executes when the none of the above condition is true.

}

## Example:

public class Test {

public static void main(String args[]){

int x = 30;

if( x == 10 ){

System.out.print("Value of X is 10");

}else if( x == 20 ){

System.out.print("Value of X is 20");

}else if( x == 30 ){

System.out.print("Value of X is 30");

}else{

System.out.print("This is else statement");

}

}

}

This would produce the following result:

Value of X is 30

## Nested if...else Statement:

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

## Syntax:

The syntax for a nested if...else is as follows:

if(Boolean\_expression 1){

//Executes when the Boolean expression 1 is true

if(Boolean\_expression 2){

//Executes when the Boolean expression 2 is true

}

}

You can nest *else if...else* in the similar way as we have nested *if* statement.

## Example:

public class Test {

public static void main(String args[]){

int x = 30;

int y = 10;

if( x == 30 ){

if( y == 10 ){

System.out.print("X = 30 and Y = 10");

}

}

}

}

This would produce the following result:

X = 30 and Y = 10

## The switch Statement:

A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

## Syntax:

The syntax of enhanced for loop is:

switch(expression){

case value :

//Statements

break; //optional

case value :

//Statements

break; //optional

//You can have any number of case statements.

default : //Optional

//Statements

}

The following rules apply to a switch statement:

* The variable used in a switch statement can only be a byte, short, int, or char.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
* When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
* When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a break. If no break appears, the flow of control will *fall through*to subsequent cases until a break is reached.
* A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

## Example:

public class Test {

public static void main(String args[]){

//char grade = args[0].charAt(0);

char grade = 'C';

switch(grade)

{

case 'A' :

System.out.println("Excellent!");

break;

case 'B' :

case 'C' :

System.out.println("Well done");

break;

case 'D' :

System.out.println("You passed");

case 'F' :

System.out.println("Better try again");

break;

default :

System.out.println("Invalid grade");

}

System.out.println("Your grade is " + grade);

}

}

Compile and run above program using various command line arguments. This would produce the following result:

$ java Test

Well done

Your grade is a C

$

**JAVA METHOD**

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, overload methods using the same names, and apply method abstraction in the program design.

Creating Method:

Considering the following example to explain the syntax of a method:

public static int funcName(int a, int b) {

// body

}

Here,

* **public static** : modifier.
* **int**: return type
* **funcName**: function name
* **a, b**: formal parameters
* **int a, int b**: list of parameters

Methods are also known as Procedures or Functions:

* **Procedures:** They don't return any value.
* **Functions:** They return value.

Method definition consists of a method header and a method body. The same is shown below:

modifier returnType nameOfMethod (Parameter List) {

// method body

}

The syntax shown above includes:

* **modifier:** It defines the access type of the method and it is optional to use.
* **returnType:** Method may return a value.
* **nameOfMethod:** This is the method name. The method signature consists of the method name and the parameter list.
* **Parameter List:** The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
* **method body:** The method body defines what the method does with statements.

Example:

Here is the source code of the above defined method called max(). This method takes two parameters num1 and num2 and returns the maximum between the two:

/\*\* the snippet returns the minimum between two numbers \*/

public static int minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

Method Calling:

For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when:

* return statement is executed.
* reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Let’s consider an example:

System.out.println("This is tutorialspoint.com!");

The method returning value can be understood by the following example:

int result = sum(6, 9);

Example:

Following is the example to demonstrate how to define a method and how to call it:

public class ExampleMinNumber{

public static void main(String[] args) {

int a = 11;

int b = 6;

int c = minFunction(a, b);

System.out.println("Minimum Value = " + c);

}

/\*\* returns the minimum of two numbers \*/

public static int minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

}

This would produce the following result:

Minimum value = 6

The void Keyword:

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown below.

Example:

public class ExampleVoid {

public static void main(String[] args) {

methodRankPoints(255.7);

}

public static void methodRankPoints(double points) {

if (points >= 202.5) {

System.out.println("Rank:A1");

}

else if (points >= 122.4) {

System.out.println("Rank:A2");

}

else {

System.out.println("Rank:A3");

}

}

}

This would produce the following result:

Rank:A1

Passing Parameters by Value:

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this the argument value is passed to the parameter.

Example:

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

public class swappingExample {

public static void main(String[] args) {

int a = 30;

int b = 45;

System.out.println("Before swapping, a = " +

a + " and b = " + b);

// Invoke the swap method

swapFunction(a, b);

System.out.println("\n\*\*Now, Before and After swapping values will be same here\*\*:");

System.out.println("After swapping, a = " +

a + " and b is " + b);

}

public static void swapFunction(int a, int b) {

System.out.println("Before swapping(Inside), a = " + a

+ " b = " + b);

// Swap n1 with n2

int c = a;

a = b;

b = c;

System.out.println("After swapping(Inside), a = " + a

+ " b = " + b);

}

}

This would produce the following result:

Before swapping, a = 30 and b = 45

Before swapping(Inside), a = 30 b = 45

After swapping(Inside), a = 45 b = 30

\*\*Now, Before and After swapping values will be same here\*\*:

After swapping, a = 30 and b is 45

**Method Overloading:**

When a class has two or more methods by same name but different parameters, it is known as method overloading. It is different from overriding. In overriding a method has same method name, type, number of parameters etc.

Let’s consider the example shown before for finding minimum numbers of integer type. If, let’s say we want to find minimum number of double type. Then the concept of Overloading will be introduced to create two or more methods with the same name but different parameters.

The below example explains the same:

public class ExampleOverloading{

public static void main(String[] args) {

int a = 11;

int b = 6;

double c = 7.3;

double d = 9.4;

int result1 = minFunction(a, b);

// same function name with different parameters

double result2 = minFunction(c, d);

System.out.println("Minimum Value = " + result1);

System.out.println("Minimum Value = " + result2);

}

// for integer

public static int minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

// for double

public static double minFunction(double n1, double n2) {

double min;

if (n1 > n2)

min = n2;

else

min = n1;

return min;

}

}

This would produce the following result:

Minimum Value = 6

Minimum Value = 7.3

Overloading methods makes program readable. Here, two methods are given same name but with different parameters. The minimum number from integer and double types is the result.

Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy.they are stored as strings in the String array passed to main( ).

Example:

The following program displays all of the command-line arguments that it is called with:

public class CommandLine {

public static void main(String args[]){

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

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

args[i]);

}

}

}

Try executing this program as shown here:

java CommandLine this is a command line 200 -100

This would produce the following result:

args[0]: this

args[1]: is

args[2]: a

args[3]: command

args[4]: line

args[5]: 200

args[6]: -100

The Constructors:

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example:

Here is a simple example that uses a constructor:

// A simple constructor.

class MyClass {

int x;

// Following is the constructor

MyClass() {

x = 10;

}

}

You would call constructor to initialize objects as follows:

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass();

MyClass t2 = new MyClass();

System.out.println(t1.x + " " + t2.x);

}

}

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example:

Here is a simple example that uses a constructor:

// A simple constructor.

class MyClass {

int x;

// Following is the constructor

MyClass(int i ) {

x = i;

}

}

You would call constructor to initialize objects as follows:

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass( 10 );

MyClass t2 = new MyClass( 20 );

System.out.println(t1.x + " " + t2.x);

}

}

This would produce the following result:

10 20

Variable Arguments(var-args):

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

typeName... parameterName

In the method declaration, you specify the type followed by an ellipsis (...) Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example:

public class VarargsDemo {

public static void main(String args[]) {

// Call method with variable args

printMax(34, 3, 3, 2, 56.5);

printMax(new double[]{1, 2, 3});

}

public static void printMax( double... numbers) {

if (numbers.length == 0) {

System.out.println("No argument passed");

return;

}

double result = numbers[0];

for (int i = 1; i < numbers.length; i++)

if (numbers[i] > result)

result = numbers[i];

System.out.println("The max value is " + result);

}

}

This would produce the following result:

The max value is 56.5

The max value is 3.0

The finalize( ) Method:

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize( ) to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize( ) method. The Java runtime calls that method whenever it is about to recycle an object of that class.

Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.

The finalize( ) method has this general form:

protected void finalize( )

{

// finalization code here

}

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.

This means that you cannot know when or even if finalize( ) will be executed. For example, if your program ends before garbage collection occurs, finalize( ) will not execute.

**MORE ABOUT VARIABLES**

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. The basic form of a variable declaration is shown here:

data type variable [ = value][, variable [= value] ...] ;

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java:

int a, b, c; // Declares three ints, a, b, and c.

int a = 10, b = 10; // Example of initialization

byte B = 22; // initializes a byte type variable B.

double pi = 3.14159; // declares and assigns a value of PI.

char a = 'a'; // the char variable a iis initialized with value 'a'

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java:

* Local variables
* Instance variables
* Class/static variables

## Local variables:

* Local variables are declared in methods, constructors, or blocks.
* Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
* Access modifiers cannot be used for local variables.
* Local variables are visible only within the declared method, constructor or block.
* Local variables are implemented at stack level internally.
* There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

## Example:

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

public class Test{

public void pupAge(){

int age = 0;

age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[]){

Test test = new Test();

test.pupAge();

}

}

This would produce the following result:

Puppy age is: 7

## Example:

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

public class Test{

public void pupAge(){

int age;

age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[]){

Test test = new Test();

test.pupAge();

}

}

This would produce the following error while compiling it:

Test.java:4:variable number might not have been initialized

age = age + 7;

^

1 error

## Instance variables:

* Instance variables are declared in a class, but outside a method, constructor or any block.
* When a space is allocated for an object in the heap, a slot for each instance variable value is created.
* Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
* Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
* Instance variables can be declared in class level before or after use.
* Access modifiers can be given for instance variables.
* The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
* Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
* Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class ( when instance variables are given accessibility) should be called using the fully qualified name . *ObjectReference.VariableName*.

## Example:

import java.io.\*;

public class Employee{

// this instance variable is visible for any child class.

public String name;

// salary variable is visible in Employee class only.

private double salary;

// The name variable is assigned in the constructor.

public Employee (String empName){

name = empName;

}

// The salary variable is assigned a value.

public void setSalary(double empSal){

salary = empSal;

}

// This method prints the employee details.

public void printEmp(){

System.out.println("name : " + name );

System.out.println("salary :" + salary);

}

public static void main(String args[]){

Employee empOne = new Employee("Ransika");

empOne.setSalary(1000);

empOne.printEmp();

}

}

This would produce the following result:

name : Ransika

salary :1000.0

## Class/static variables:

* Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
* There would only be one copy of each class variable per class, regardless of how many objects are created from it.
* Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
* Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
* Static variables are created when the program starts and destroyed when the program stops.
* Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
* Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
* Static variables can be accessed by calling with the class name . *ClassName.VariableName*.
* When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

## Example:

import java.io.\*;

public class Employee{

// salary variable is a private static variable

private static double salary;

// DEPARTMENT is a constant

public static final String DEPARTMENT = "Development ";

public static void main(String args[]){

salary = 1000;

System.out.println(DEPARTMENT+"average salary:"+salary);

}

}

This would produce the following result:

Development average salary:1000

**Note:** If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

**MORE ABOUT DATATYPES**

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

* Primitive Data Types
* Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

byte:

* Byte data type is an 8-bit signed two's complement integer.
* Minimum value is -128 (-2^7)
* Maximum value is 127 (inclusive)(2^7 -1)
* Default value is 0
* Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
* Example: byte a = 100 , byte b = -50

short:

* Short data type is a 16-bit signed two's complement integer.
* Minimum value is -32,768 (-2^15)
* Maximum value is 32,767 (inclusive) (2^15 -1)
* Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
* Default value is 0.
* Example: short s = 10000, short r = -20000

int:

* Int data type is a 32-bit signed two's complement integer.
* Minimum value is - 2,147,483,648.(-2^31)
* Maximum value is 2,147,483,647(inclusive).(2^31 -1)
* Int is generally used as the default data type for integral values unless there is a concern about memory.
* The default value is 0.
* Example: int a = 100000, int b = -200000

long:

* Long data type is a 64-bit signed two's complement integer.
* Minimum value is -9,223,372,036,854,775,808.(-2^63)
* Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
* This type is used when a wider range than int is needed.
* Default value is 0L.
* Example: long a = 100000L, int b = -200000L

float:

* Float data type is a single-precision 32-bit IEEE 754 floating point.
* Float is mainly used to save memory in large arrays of floating point numbers.
* Default value is 0.0f.
* Float data type is never used for precise values such as currency.
* Example: float f1 = 234.5f

double:

* double data type is a double-precision 64-bit IEEE 754 floating point.
* This data type is generally used as the default data type for decimal values, generally the default choice.
* Double data type should never be used for precise values such as currency.
* Default value is 0.0d.
* Example: double d1 = 123.4

boolean:

* boolean data type represents one bit of information.
* There are only two possible values: true and false.
* This data type is used for simple flags that track true/false conditions.
* Default value is false.
* Example: boolean one = true

char:

* char data type is a single 16-bit Unicode character.
* Minimum value is '\u0000' (or 0).
* Maximum value is '\uffff' (or 65,535 inclusive).
* Char data type is used to store any character.
* Example: char letterA ='A'

Reference Data Types:

* Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
* Class objects, and various type of array variables come under reference data type.
* Default value of any reference variable is null.
* A reference variable can be used to refer to any object of the declared type or any compatible type.
* Example: Animal animal = new Animal("giraffe");

Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

byte a = 68;

char a = 'A'

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

int decimal = 100;

int octal = 0144;

int hexa = 0x64;

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

"Hello World"

"two\nlines"

"\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For example:

char a = '\u0001';

String a = "\u0001";

Java language supports few special escape sequences for String and char literals as well. They are:

|  |  |
| --- | --- |
| **Notation** | **Character represented** |
| \n | Newline (0x0a) |
| \r | Carriage return (0x0d) |
| \f | Formfeed (0x0c) |
| \b | Backspace (0x08) |
| \s | Space (0x20) |
| \t | tab |
| \" | Double quote |
| \' | Single quote |
| \\ | backslash |
| \ddd | Octal character (ddd) |
| \uxxxx | Hexadecimal UNICODE character (xxxx) |

#### 2) What to cover for the theory part?

Now in this portion you get questions from all parts of your syllabus. So, you need to go through your text books or class or tuition notes for brief theory of every chapters.

Remember, short output questions and fill in the blanks of small programs are also included in this theory portion. Give importance to the various Character, String and Mathematical functions.

I have tried to list down few of the important topics from the theory part which should be taken into consideration while going through the theory part:

* Elementary Concept of Objects and Classes.
* Objects encapsulate state and behavior – numerous examples; member variables; attributes or features.
* Classes as abstractions for sets of objects; class as an object factory; concept of type, primitive data types, composite data types. Variable declarations for both types; difference between the two types. Objects as instances of a class.
* Features of Java
* Features of OOP
* Compiler, JVM, Byte Code
* Tokens and types (Keyword, identifier, literal, operator, separator/punctuator).
* Operators – types and calculations using pre-fix or post-fix
* Data types and their size, range.
* Type Conversion – implicit, explicit
* Writing java expression for a given equation.
* Loops (while, do while and for), nested loops, break and continue.
* Simple output questions based on loops (includes nested loops)
* Converting from one form of loop to the other form.
* Functions and its types
* Need of functions. Types of functions (pure and impure). Function declaration and definition, ways of calling functions (call by value and call by reference)
* Returning information/messages from the functions and use of multiple functions and more than one function with the same name (function overloading).
* Use of static data member with static member function. Discuss invocation of functions on objects (through the reference).
* Constructor and its types – Default constructor, parameterized constructor, constructor with default parameter and constructor overloading.
* Writing prototypes of functions
* Access specifiers and scope and visibility
* Scope of variables, instance variables, argument variables, local variables.
* Class as a composite type, distinction between primitive type and composite or class types.
* Class may be considered as a new data type created by the user, that has its own functionality.
* Wrapper class.
* Some wrapper class functions. The following methods are to be covered:
  + int parseInt(String s), int valueOf(String s),
  + long parseLong(String s), long valueOf(String s),
  + float parseFloat(String s), float valueOf(String s),
  + double parseDouble(String s),
  + double valueOf(String s), boolean isDigit(char ch),
  + boolean isLetter(char ch),
  + boolean isLetterOrDigit(char ch),
  + boolean isLowerCase(char ch),
  + boolean isUpperCase(char ch),
  + boolean isWhitespace(char ch),
  + char toLowerCase (char ch)
  + char toUpperCase(char ch)
* Library functions. The following functions have to be covered:

**String library functions:**

* Char charAt (int n)
* int compareTo(String1, String2)
* String concat(String str)
* boolean endsWith(String str)
* boolean equals(String str)
* boolean equalsIgnoreCase(String str)
* int indexOf(char ch)
* int lastIndexOf(char ch)
* int length( )
* String replace (char oldChar,char newChar)
* boolean startsWith(String str)
* String substring(int beginIndex, int endIndex)
* String toLowerCase( )
* String toUpperCase( )
* String trim( )
* String valueOf(all types)

**Mathematical Library Functions:**

* pow(x,y), log(x), sqrt(x), ceil(x), floor(x), rint(x),
* abs(a), max(a, b), min(a,b), random( ), sin(x), cos(x), tan(x).
* Arrays – storing, retrieving and arranging data
* Arrays and their uses, sorting algorithms – selection sort and bubble sort; Search algorithms – linear search and binary search Example of a composite type.
* Scanner classes functions.

End of study material 4

**This study material is intended for reference only. Text books must be followed thoroughly**.