**Java Access modifiers**

**Private access modifiers**

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

public class Logger {

private String format;

public String getFormat() {

returnthis.format;

}

public void setFormat(String format) {

this.format = format;

}

}

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

**Public Access Modifier - public:**

Can be accessed from any class and from any where

If you need to access public class members to be accssed you need to import the puplic class to access the public members in that class

For example :

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

publicstaticvoid main(String[] arguments){

// ...

}

**Protected access modifiers :**

A protected access modifier cannot be applied to a class and interface, methods and fields can be declared protected.

But methods and fields in interface cannot be declared protected

Protected key word gives a chance to its sub classes to access helper methods and variables and prevents a non-related class accessing the members

classAudioPlayer{

protectedbooleanopenSpeaker(Speakersp){

// implementation details

}

}

classStreamingAudioPlayer{

booleanopenSpeaker(Speakersp){

// implementation details

}

}

Here, if we define openSpeaker() method as private, then it would not be accessible from any other class other than AudioPlayer. If we define it as public, then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, thats why we used protected modifier.

## Access Control and Inheritance:

* Methods declared public in a superclass also must be public in all subclasses.
* Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
* Methods declared without access control (no modifier was used) can be declared more private in subclasses.
* Methods declared private are not inherited at all, so there is no rule for them.

**Java Non access modifiers:**

## The static Modifier:

No matter how many you create of static methods/classes/variables but the JVM will keep only one copy of the static instance.IF at all any modification are done this would be reflected in original copy.

## Static Variables:

## The static key word is used to create variables that will exist independently of any instances created for the class.

## Only one copy of the static variable exists regardless of the number of instances of the class

## Static variables are also known as class variables. Local variables cannot be declared static.

## Static Methods:

## Static methods do not uses instance variables of any object of the class they are defined

## Static methods take all the data from the parameter and compute something from those parameters ,with no reference to variables.

## Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

## Example:

public class InstanceCounter {

private static intnumInstances = 0;

protected static intgetCount() {

returnnumInstances;

}

private static void addInstance() {

numInstances++;

}

InstanceCounter() {

InstanceCounter.addInstance();

}

public static void main(String[] arguments) {

System.out.println("Starting with " +

InstanceCounter.getCount() + " instances");

for (int i = 0; i < 500; ++i){

newInstanceCounter();

}

System.out.println("Created " +

InstanceCounter.getCount() + " instances");

}

}

## The final Modifier:

## final Variables: A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different objective. the value or the refrence cannot be changed at any time in the life cycle.

## Final Methods:

Final methods cannot be overridden by subclasses.

Final keyword itself will restrict its subclasses to modify the method

Main funda of making a method final is to disallow others to modify the content of the method

Example:

public class Test{

public final void changeName(){

// body of method

}

}

## final Classes:

the main funda of making a class final is to make it uavilable for its inherit any of the method or variable.

Example :

public final class Test {

// body of class

}

## The abstract Modifier:

* Abstract class can never be instantiated.If a class is declared as abstract then the only way to use the class is extended the class.
* A class cannot be both abstract and final. (since a final class cannot be extended)
* If a class contains abstract methods then the class should be declared abstract
* An abstract class may contain both abstract methods as well normal methods.

Example:

abstract class Caravan{

private double price;

private String model;

private String year;

public abstract void goFast();

//an abstract method

public abstract void changeColor();

}

## abstract Methods:

* An abstract method is a method declared with out any implementation.
* The methods body(implementation) is provided by the subclass
* Abstract methods can never be final or strict
* Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.
* If a class contains one or more abstract methods then the class must be declared abstract.
* An abstract class does not need to contain abstract methods.
* The abstract method ends with a semicolon. Example: public abstract sample();

Example:

public abstract class SuperClass{

abstract void m(); //abstract method

}

classSubClass extends SuperClass{

// implements the abstract method

void m(){

.........

}

}

## synchronized Modifier:

* The synchronized key word used to indicate that a method can be accessed by only one thread at a time
* The synchronized modifier can be applied with any of the four access level modifiers.

Example :

public synchronized void showDetails(){

.......

}

## transient Modifier:

I really coudent understand this

* An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.
* This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

Example : public transient int limit = 55; // will not persist

publicint b; // will persist

## volatile Modifier:

* The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.
* Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory.
* Volatile can only be applied to instance variables

Example:

public class MyRunnable implements Runnable

{

private volatile boolean active;

public void run()

{

active = true;

while (active) // line 1

{

// some code here

}

}

public void stop()

{

active = false; // line 2

}

}

**Operators:**

## The Arithmetic Operators:

|  |  |  |
| --- | --- | --- |
| **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:

|  |  |  |
| --- | --- | --- |
| **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:

## Example:

a = 0011 1100

b = 0000 1101

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

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

|  |  |  |
| --- | --- | --- |
| **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:

|  |  |  |
| --- | --- | --- |
| **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:

|  |  |  |
| --- | --- | --- |
| **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 |
| **Operator** | **Description** | **Example** |

## Conditional 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

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 );

}

}

## Instance of Operator:

## This is used to check whether the given variable is an reference object or not . Just to confirm the object reference .the syntax is as follows

## Boolean variable = (instace variable)instanceof (class/interface)

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);

}

}

## Here the result will store the bolean returned from the method

## The while Loop:

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

Syntax

While(Boolean\_expression )

{

//staements

}

## 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

do

{

//Statements

}while(Boolean\_expression);

## 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.

Syntax:

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.

Enhanced for loop in Java:

This is mainly used for Arrays.

Syntax

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.

## The break Keyword:

## The break keyword is used to stop a loop from execution. Break keyword must be used inside any loop and with switch statement. The break keyword will stop execution of the loop from the inner loop and start executing next line of the code. I.e. the break keyword will entirely terminate the process of execution of the loop and starts executing the next line i.e. next line after end of the loop

## Syntax:break;

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");}

}

}

## 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.

Syntax: continue ;

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");

}

}

}

## Decision making :

## The If statement

## The if statement consists of one of more expressions with one or more statement.

if(Boolean expression)

{

//Statements will execute if the Boolean expression is true

}

## If the Boolean expression in the “IF “block is true the expression executes the block of code with in the “if” statement.

## 

## The if...else Statement:

## In this statement .if the Boolean expression with in the if block is true then the block under if ll be executed and if the expression is false then it wilececute the else block.

## 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: 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.

}

## 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.

## Example :

if(Boolean\_expression 1){

//Executes when the Boolean expression 1 is true

if(Boolean\_expression 2){

//Executes when the Boolean expression 2 is true

}

## }

Working 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");

}

}

}

}

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.

switch(expression){

case value :

//Statements

break; //optional

case value :

//Statements

break; //optional

//You can have any number of case statements.

default : //Optional

//Statements

}

* 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.

Java - Numbers Class.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [xxxValue()](http://www.tutorialspoint.com/java/number_xxxvalue.htm) Converts the value of *this* Number object to the xxx data type and returned it. |
| 2 | [compareTo()](http://www.tutorialspoint.com/java/number_compareto.htm) Compares *this* Number object to the argument. |
| 3 | [equals()](http://www.tutorialspoint.com/java/number_equals.htm) Determines whether *this* number object is equal to the argument. |
| 4 | [valueOf()](http://www.tutorialspoint.com/java/number_valueof.htm) Returns an Integer object holding the value of the specified primitive. |
| 5 | [toString()](http://www.tutorialspoint.com/java/number_tostring.htm) Returns a String object representing the value of specified int or Integer. |
| 6 | [parseInt()](http://www.tutorialspoint.com/java/number_parseint.htm) This method is used to get the primitive data type of a certain String. |
| 7 | [abs()](http://www.tutorialspoint.com/java/number_abs.htm) Returns the absolute value of the argument. |
| 8 | [ceil()](http://www.tutorialspoint.com/java/number_ceil.htm) Returns the smallest integer that is greater than or equal to the argument. Returned as a double. |
| 9 | [floor()](http://www.tutorialspoint.com/java/number_floor.htm) Returns the largest integer that is less than or equal to the argument. Returned as a double. |
| 10 | [rint()](http://www.tutorialspoint.com/java/number_rint.htm) Returns the integer that is closest in value to the argument. Returned as a double. |
| 11 | [round()](http://www.tutorialspoint.com/java/number_round.htm) Returns the closest long or int, as indicated by the method's return type, to the argument. |
| 12 | [min()](http://www.tutorialspoint.com/java/number_min.htm) Returns the smaller of the two arguments. |
| 13 | [max()](http://www.tutorialspoint.com/java/number_max.htm) Returns the larger of the two arguments. |
| 14 | [exp()](http://www.tutorialspoint.com/java/number_exp.htm) Returns the base of the natural logarithms, e, to the power of the argument. |
| 15 | [log()](http://www.tutorialspoint.com/java/number_log.htm) Returns the natural logarithm of the argument. |
| 16 | [pow()](http://www.tutorialspoint.com/java/number_pow.htm) Returns the value of the first argument raised to the power of the second argument. |
| 17 | [sqrt()](http://www.tutorialspoint.com/java/number_sqrt.htm) Returns the square root of the argument. |
| 18 | [sin()](http://www.tutorialspoint.com/java/number_sin.htm) Returns the sine of the specified double value. |
| 19 | [cos()](http://www.tutorialspoint.com/java/number_cos.htm) Returns the cosine of the specified double value. |
| 20 | [tan()](http://www.tutorialspoint.com/java/number_tan.htm) Returns the tangent of the specified double value. |
| 21 | [asin()](http://www.tutorialspoint.com/java/number_asin.htm) Returns the arcsine of the specified double value. |
| 22 | [acos()](http://www.tutorialspoint.com/java/number_acos.htm) Returns the arccosine of the specified double value. |
| 23 | [atan()](http://www.tutorialspoint.com/java/number_atan.htm) Returns the arctangent of the specified double value. |
| 24 | [atan2()](http://www.tutorialspoint.com/java/number_atan2.htm) Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta. |
| 25 | [toDegrees()](http://www.tutorialspoint.com/java/number_todegrees.htm) Converts the argument to degrees |
| 26 | [toRadians()](http://www.tutorialspoint.com/java/number_toradians.htm) Converts the argument to radians. |
| 27 | [random()](http://www.tutorialspoint.com/java/number_random.htm) Returns a random number. |
| **SN** | **Methods with Description** |
| 1 | [xxxValue()](http://www.tutorialspoint.com/java/number_xxxvalue.htm) Converts the value of *this* Number object to the xxx data type and returned it. |
| 2 | [compareTo()](http://www.tutorialspoint.com/java/number_compareto.htm) Compares *this* Number object to the argument. |
| 3 | [equals()](http://www.tutorialspoint.com/java/number_equals.htm) Determines whether *this* number object is equal to the argument. |
| 4 | [valueOf()](http://www.tutorialspoint.com/java/number_valueof.htm) Returns an Integer object holding the value of the specified primitive. |
| 5 | [toString()](http://www.tutorialspoint.com/java/number_tostring.htm) Returns a String object representing the value of specified int or Integer. |
| 6 | [parseInt()](http://www.tutorialspoint.com/java/number_parseint.htm) This method is used to get the primitive data type of a certain String. |
| 7 | [abs()](http://www.tutorialspoint.com/java/number_abs.htm) Returns the absolute value of the argument. |
| 8 | [ceil()](http://www.tutorialspoint.com/java/number_ceil.htm) Returns the smallest integer that is greater than or equal to the argument. Returned as a double. |

**Java - Character Class**

Initialization of characters can be done in below fashion

Char hello = ‘h’;

Where this hello will hold the value ‘h’ within it. If at all you want to go with character array then

Char [] hello={‘h’,’e’,’l’,’l’,’o’,’.’}

Here we have initialized the character as an character array

So if the in stance is taken to system output buffer. it will hold the value of array instead of holding the name of it.

.

**Java - String Class**

**Creating String**

Here we can intiallize an string in the below fassion.

String greeting = "Hello world!";

Here the string greetings will hold the value “Hello world”

Here is the example :

public class StringDemo{

public static void main(String args[]){

char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};

String helloString = new String(helloArray);

System.out.println( helloString );

}

}

This is will give:

hello.

**String length :**

**Here we will use string classes to get the string length.**

**The string length is so called :**

**Lenget();**

**Here is the example :**

public class StringDemo {

public static void main(String args[]) {

String palindrome = "Dot saw I was Tod";

int len = palindrome.length();

System.out.println( "String Length is : " + len );

}

}

Here in this lines of code

palindrome. length (); this will return the actual length of the array of type integer

Concatenating Strings:

Here we can join two string into a single string.

Using .concat(); method in string class .here is the example.

string1.concat(string2);

here in this snippet the content of string1 will be concatenated with the content of string2.Here is an working programme example :

public class StringDemo2 {

public static void main(String args[]) {

String string1 = "saw I was ";

String string2= "hellow sir ";

System.out.println("Dot " + string1 + "Tod");

System.out.println(string1.concat(string2));

}

}

Here the output will be;

Dot saw I was Tod

saw I was hellow sir

Creating Format Strings:

System.out.printf("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

Here the string will be printed with all the strings characters float and integer values

**Java – Arrays:**

Declaring Array Variables:

This will allocate an sequence of memory in the stack to be used for date of same type.

Here is the declaration of arrays in two different manner

dataType[] arrayRefVar; // preferred way.

This is in a preferred manner.

dataType arrayRefVar[]; // works but not preferred way.

Creating Arrays:

arrayRefVar = new dataType[arraySize];

here an array is created with new datatype[arraysize] with the reference variable arrayRefVar.the refrence is for refering the array .

and the datatype is type casting of the input given.

You can also create the value as follows .

Datatype[] arrayRefVar= new datatype[arraysize];

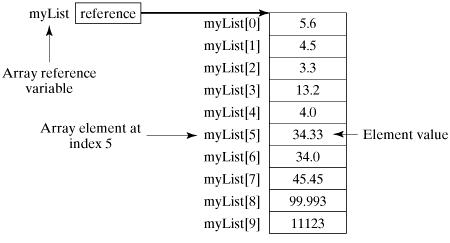
There is another way of initializing the array variable.

Datatype[] arrayRefVar={value0, value1, value2,…so on };

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

double[] mylist = new double[10];

this will create the variable in the below format .the above declaration will create an array with the name mylist.wich looks as shown below .



Processing Arrays:

Here is the code sample which will give a clear picture of what is an array.

Here is a complete example of showing how to create, initialize and process arrays:

public class TestArray {

public static void main(String[] args) {

double[] myList = {1.9, 2.9, 3.4, 3.5};

// Print all the array elements

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

System.out.println(myList[i] + " ");

}

// Summing all elements

double total = 0;

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

total += myList[i];

}

System.out.println("Total is " + total);

// Finding the largest element

double max = myList[0];

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

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

}

System.out.println("Max is " + max);

}

}

This is the output of the programme

1.9

2.9

3.4

3.5

Total is 11

Max is 3.5

**Java - Date & Time :**

Here we have special class called date which returns the accurate date and time with respect to the date and time of the system.

The Date class supports two constructors. The first constructor initializes the object with the current date and time.

import java.util.Date;

public class DateDemo {

public static void main(String args[]) {

// Instantiate a Date object

Date date = new Date();

// display time and date using toString()

System.out.println(date.toString());

}

}

The above snippet will print the date with day and date with time.there are other methouds which will return you date time and so

Date Formatting using SimpleDateFormat:

import java.util.\*;

import java.text.\*;

public class DateDemo {

public static void main(String args[]) {

Date dNow = new Date( );

SimpleDateFormat ft =

new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

System.out.println("Current Date: " + ft.format(dNow));

}

}

Here in this snippet we will give format in which we need to display.

The class DateFormat will covert date day and month for the display.

Here E is for day

Yyyy is for year ,MM is for month ,a is for AM/PM zzz is for time zone

|  |  |  |
| --- | --- | --- |
| Character | Description | Example |
| G | Era designator | AD |
| y | Year in four digits | 2001 |
| M | Month in year | July or 07 |
| d | Day in month | 10 |
| h | Hour in A.M./P.M. (1~12) | 12 |
| H | Hour in day (0~23) | 22 |
| m | Minute in hour | 30 |
| s | Second in minute | 55 |
| S | Millisecond | 234 |
| E | Day in week | Tuesday |
| D | Day in year | 360 |
| F | Day of week in month | 2 (second Wed. in July) |
| w | Week in year | 40 |
| W | Week in month | 1 |
| a | A.M./P.M. marker | PM |
| k | Hour in day (1~24) | 24 |
| K | Hour in A.M./P.M. (0~11) | 10 |
| z | Time zone | Eastern Standard Time |
| ' | Escape for text | Delimiter |
| " | Single quote | ` |
|  |  |  |

Parsing Strings into Dates:

The SimpleDateFormat class has some additional methods, notably parse( ) , which tries to parse a string according to the format stored in the given SimpleDateFormat object. For example:

import java.util.\*;

import java.text.\*;

public class DateDemo {

public static void main(String args[]) {

SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");

String input = args.length == 0 ? "1818-11-11" : args[0];

System.out.print(input + " Parses as ");

Date t;

try {

t = ft.parse(input);

System.out.println(t);

} catch (ParseException e) {

System.out.println("Unparseable using " + ft);

}

}

}

Here in this snippet we will get the parsed date and un parsed date.

**Java – Methods:**

Creating a Method:

modifier returnValueType methodName(list of parameters) {

// Method body;

}

* **Modifiers:** The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.
* **Return Type:** A method may return a value. The returnValueType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the returnValueType is the keyword void.
* **Method Name:** This is the actual name of the method. The method name and the parameter list together constitute the method signature.
* **Parameters:** A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
* **Method Body:** The method body contains a collection of statements that define what the method does.

**Overloading Methods:**

The solution is to create another method with the same name but different parameters.

This implementation of two or more methods with the same name is referred as method overloading

The Constructors:

It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically this will initialize values to the instance variable of the class or to perform any startup procedure to create fully formed objects.

Example:

// A simple constructor.

class MyClass {

int x;

// Following is the constructor

MyClass() {

x = 10;

}

}

Now we can call this class as below this will print the value of X.

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass();

MyClass t2 = new MyClass();

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

}

}

The finalize ( ) Method:

We can define a method which can be called at the end of object destruction. This method will be called just before the object is going to be destroyed. That is we can use it with finalize key word.

protected void finalize( )

{

// finalization code here

}

**Java - Streams, Files and I/O:**

To obtain charter stream which is attached to the console we will wrap system. in into buffer object ,below is most commonly used character stream buffer

BufferedReader br = new BufferedReader( new InputStreamReader(System.in));

To obtain a character-based stream that is attached to the console, you wrap System.in in a BufferedReader object, to create a character stream.once the buffer reader is obtained then we can use read() method .

Reading Strings from Console:

To read a string from the keyboard we will use redline() method

String readLine()throws IOException //this is the syntax of reading the thing with a string.

Writing Console Output:

Console output is most easily accomplished with **print( )** and **println( )**. These methods are defined by the class **PrintStream** which is the type of the object referenced by **System.out**.

Even though System.out is a byte stream, using it for simple program output is still acceptable.

Reading and Writing Files:

A stream can be defined as sequence of data ,input stream is used to read data from source and output stream is used to write data to destination .

Below is the structure of I/O stream

****

FileInputStream: file input stream is used to read data from a file .This can be achieved as below

InputStream f =newFileInputStream("C:/java/hello");

Here “f” is the object for FileInputStream which will read from the file located in "C:/java/hello"

One we get the refrence in our hand we have helper methods to perform operation on it

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | public void close() throws IOException{}  This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | protected void finalize()throws IOException {}  This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | public int read(int r)throws IOException{}  This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file. |
| 4 | public int read(byte[] r) throws IOException{}  This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned. |
| 5 | public int available() throws IOException{}  Gives the number of bytes that can be read from this file input stream. Returns an int. |

ByteArrayInputStream :

ByteArrayInputStream class allows a buffer in the memory to be used as an InputStream. There are following forms of constructors to create ByteArrayInputStream objects

ByteArrayInputStream bArray =new ByteArrayInputStream(byte[] a);

Another form takes an array of bytes, and two ints, where off is the first byte to be read and len is the number of bytes to be read.

ByteArrayInputStream bArray =newByteArrayInputStream(byte[]a,int off, int len);

There are some methods which can be applied on the above buffer stream

Here is the example

import java.io.\*;

publicclassByteStreamTest{

publicstaticvoid main(String args[])throwsIOException{

ByteArrayOutputStream bOutput =newByteArrayOutputStream(12);

while( bOutput.size()!=10){

// Gets the inputs from the user

bOutput.write(System.in.read());

}

byte b []= bOutput.toByteArray();

System.out.println("Print the content");

for(int x=0; x < b.length; x++){

// printing the characters

System.out.print((char)b[x]+" ");

}

System.out.println(" ");

int c;

ByteArrayInputStream bInput =newByteArrayInputStream(b);

System.out.println("Converting characters to Upper case ");

for(int y =0; y <1; y++){

while(( c= bInput.read())!=-1){

System.out.println(Character.toUpperCase((char)c));

}

bInput.reset();

}

}

}

DataInputStream :

The DataInputStream is used in the context of DataOutputStream and can be used to read primitives.

Following is the constructor to create an InputStream:

InputStreamin=DataInputStream(InputStreamin);

This is nothing but the file input stream,

The system input and file input streams can be wrapped into this input stream

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | public final int read(byte[] r, int off, int len)throws IOException  Reads up to len bytes of data from the input stream into an array of bytes. Returns the total number of bytes read into the buffer otherwise -1 if it is end of file. |
| 2 | Public final int read(byte [] b)throws IOException  Reads some bytes from the inputstream an stores in to the byte array. Returns the total number of bytes read into the buffer otherwise -1 if it is end of file. |
| 3 | (a) public final Boolean readBooolean()throws IOException,  (b) public final byte readByte()throws IOException,  (c) public final short readShort()throws IOException  (d) public final Int readInt()throws IOException  These methods will read the bytes from the contained InputStream. Returns the next two bytes of the InputStream as the specific primitive type. |
| 4 | public String readLine() throws IOException  Reads the next line of text from the input stream. It reads successive bytes, converting each byte separately into a character, until it encounters a line terminator or end of file; the characters read are then returned as a String. |

File Navigation and I/O:

File Class:

Java file class represents file names and file pathname on an abstract manner.

This class is used to create file, directories,file searching and file deletion.

The file object represents the actual file on the disk

There are many methods to create file object :

File(File parent,String child);

creates a new File instance from a parent abstract pathname and a child pathname string.

File(String pathname);

creates a new File instance by converting the given pathname string into an abstract pathname.

File(String parent,String child)

creates a new File instance from a parent pathname string and a child pathname string.

File(URI uri)

creates a new File instance by converting the given file: URI into an abstract pathname

FileReader Class:

File reader class is inherited from InputStreamRader class .FileReader is used for reading streams of characters

FileReader(File file) :: creates a new FileReader, given the File to read from.

FileReader(FileDescriptor fd) :: creates a new FileReader, given the FileDescriptor to read from.

FileReader(String fileName) :: creates a new FileReader, given the name of the file to read from.

FileWriter Class :

File writer class is inherited from OutputStreamWriter class. This class is used to write stream of characters.

FileWriter(File file) :: creates a FileWriter object given a File object.

FileWriter(File file,boolean append) :: creates a FileWriter object given a File object.

FileWriter(FileDescriptor fd) :: creates a FileWriter object associated with a file descriptor.

FileWriter(String fileName) :: creates a FileWriter object given a file name.

FileWriter(String fileName,boolean append) :: creates a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

import java.io.\*;

publicclassFileRead{

publicstaticvoid main(String args[])throwsIOException{

File file =new File("Hello1.txt");

// creates the file

file.createNewFile();

// creates a File Writer Object

FileWriter writer =new FileWriter(file);

// Writes the content to the file

writer.write("This\n is\n an\n example\n");

writer.flush();

writer.close();

FileReader fr =new FileReader(file);

char[] a =new char[50];

fr.read(a);// reads the content to the array

for(char c : a)

System.out.print(c);//prints the characters one by one

fr.close();

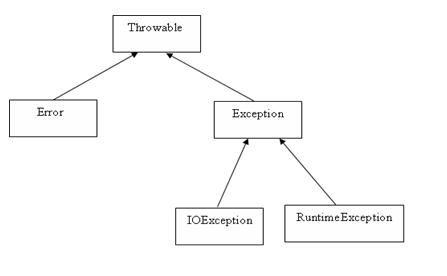
}

}

# Java - Exceptions Handling :

* **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
* **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
* **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

## Exception Hierarchy:



A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

try

{

//Protected code

}catch(ExceptionName e1)

{

//Catch block

}

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Each try block can have multiple catch blocks like below

Try {

//do something

}

Catch 1{

}

Catch 2{

}

Catch 3{

}

## The throws/throw Keywords:

## If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

## You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword. Try to understand the different in throws and throw keywords.

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

import java.io.\*;

public class className

{

public void withdraw(double amount) throws RemoteException,

InsufficientFundsException

{

// Method implementation

}

//Remainder of class definition

}

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

import java.io.\*;

public class className

{

public void withdraw(double amount) throws RemoteException,

InsufficientFundsException

{

// Method implementation

}

//Remainder of class definition

}

## The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

The syntax is

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}finally

{

//The finally block always executes.

}

In the above snippet the code will catch the exception and run the appropriate catch block. Once the catch block is executed then the iteration will come over to finally block.

## Declaring you own Exception:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

* All exceptions must be a child of Throwable.
* If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
* If you want to write a runtime exception, you need to extend the RuntimeException class.

**Java Object oriented**

# Java – Inheritance:

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order.

When we talk about inheritance, the most commonly used keyword would be **extends** and **implements**. These words would determine whether one object IS-A type of another. By using these keywords we can make one object acquire the properties of another object.

## IS-A Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

public class Animal{

}

public class Mammal extends Animal{

}

public class Reptile extends Animal{

}

public class Dog extends Mammal{

}

Now, based on the above example, In Object Oriented terms, the following are true:

* Animal is the superclass of Mammal class.
* Animal is the superclass of Reptile class.
* Mammal and Reptile are subclasses of Animal class.
* Dog is the subclass of both Mammal and Animal classes.

The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.

Example :

public interface Animal {}

public class Mammal implements Animal{

}

public class Dog extends Mammal{

}

## HAS-A relationship:

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

For example :

public class Vehicle{}

public class Speed{}

public class Van extends Vehicle{

private Speed sp;

}

Now the van has speed .

We do not have to put entire code of speed into van but we can reuse the content again and again by using has a relationship .

What happens in real scenario is the van class will hide the implementation of speed from van class user .so the user may ask van class to do some operation and van class may do it on its own or recomond other class to do the operation

A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:

public class extends Animal, Mammal{}

# Java - Overriding

we talked about super classes and sub classes. If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement.In object-oriented terms, overriding means to override the functionality of an existing method.

Example :

class Animal{

public void move(){

System.out.println("Animals can move");

}

}

class Dog extends Animal{

public void move(){

System.out.println("Dogs can walk and run");

}

}

public class TestDog{

public static void main(String args[]){

Animal a = new Animal(); // Animal reference and object

Animal b = new Dog(); // Animal reference but Dog object

a.move();// runs the method in Animal class

b.move();//Runs the method in Dog class

}

}

This would produce the following result:

Animals can move

Dogs can walk and run

In the above example, you can see that the even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.

Now consider the following program

class Animal{

public void move(){

System.out.println("Animals can move");

}

}

class Dog extends Animal{

public void move(){

System.out.println("Dogs can walk and run");

}

public void bark(){

System.out.println("Dogs can bark");

}

}

public class TestDog{

public static void main(String args[]){

Animal a = new Animal(); // Animal reference and object

Animal b = new Dog(); // Animal reference but Dog object

a.move();// runs the method in Animal class

b.move();//Runs the method in Dog class

b.bark();

}

}

The above code will generate the following result

TestDog.java:30: cannot find symbol

symbol : method bark()

location: class Animal

b.bark();

^

## Rules for method overriding:

* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example: if the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* Constructors cannot be overridden.