|  |
| --- |
|  |
| diff. bt c & java: |
|  |  |
|  | C: JAVA |
|  | 1) Structural oriented 1)OOPs |
|  | 2)#include<stdio.h> 2)no header files-Packages |
|  | header files import java.io.\*; |
|  | 3) 32 keywords 3)49 keywords |
|  | no - sizeof,extern,register,signed,unsigned |
|  | 4) primitive and user 4)only primitive data type |
|  | defined datatype |
|  | 5) #define A 40 -- macros 5)no global declaration |
|  | 6) Pointers 6)no pointers but we can access by creating object |
|  |  |
|  |  |
|  | c++ java |
|  | 1) OBJECT |
|  | class -- blue print/template 1) No mutilple inheritance instead we have interfaces |
|  | 2) Encapsulation -- wrapping of data into single unit |
|  | (hiding) |
|  | 3) Abstraction -- hiding the info from class |
|  | 4) Inheritance -- accessing properties of one class into |
|  | another(code reusability) |
|  | single,multiple,multilevel,hybrid,hiearchial |
|  | 5) Polymorphism -- one class in many forms |
|  | 1) static/compile polymorphism -- method/constructor/operator |
|  | overloading |
|  | 2)Dynamic/runtime -- method overriding |
|  | 6) Constructor |
|  | ->should not have return type |
|  | 7) Destructor -- deallocation of memory(~) 2) there is no destructor,instead we have |
|  | 1)automatic garbage collection |
|  | 2)System.gc()/Runtime.gc() where it automatically calls |
|  | protected void finalize(){ |
|  | //contains the resources to be deallocated and invoked only once in lifecycle of program |
|  | 8)String is datatype 3)String is class/literal |
|  |  |
|  |  |
|  |  |
|  | jdk 1.0 --oak |
|  | 2.0 -- playground |
|  | 3.0 -- kestrel |
|  | 4.0 -- merlin |
|  | 5.0 -- tiger |
|  | 6.0 -- mustang |
|  | 7.0 -- dolphin |
|  | 8.0 -- spyder |
|  |  |
|  |  |
|  | Environmenr setup: |
|  | set JAVA\_HOME and PATH in environment variables:thispc->rightclick->properties->advanced settings->env. var-> user var.->new |
|  | cmd prompt javac |
|  |  |
|  | JDK: |
|  | java development kit |
|  | --> jre + extra tools to develop java prog. |
|  | JVM: |
|  | java virtual machine |
|  | --> used to read and execute my byte code. |
|  |  |
|  | JIT: |
|  | just intime compiler |
|  | --> convert byte code to native machine code. |
|  | JRE: |
|  | jav runtime Environment |
|  | --> support some library file to develop the prog. |
|  |  |
|  | FUNDAMENTALS OF JAVA: |
|  |  |
|  | package com.pack; //package declaration |
|  |  |
|  | //importing packages |
|  | import java.io; |
|  |  |
|  | //class declaration |
|  | class Sample{ |
|  |  |
|  | public static void main(String args[]) |
|  | System.out.println("Hello World"); |
|  |  |
|  | } |
|  | } |
|  |  |
|  | Coding standard: |
|  |  |
|  | 1) class: |
|  | -->starting letter of each word should be capital |
|  | 2) method: |
|  | -->from second the starting letter should be capital |
|  | 3) variables: |
|  | -->all should be small |
|  | -->constant should be all in capital letters |
|  | 4) Identifiers: |
|  | -->Names given for classes methods variables. |
|  | -->start with alphabets |
|  | --->2 special characters \_ and $ can contain numbers |
|  | 5) keywords: |
|  | -->49 keywords |
|  | --> goto or keywords of java but if we use we get compilation error |
|  | --> instead of goto use continue and for const use final |
|  | -->3 reserved words true ,flase for boolean "null" for object |
|  | 6)DataTypes: |
|  | the type of value it stores. |
|  | 1)only primitive datatypes are there: |
|  | 1) byte |
|  | 2)short |
|  | 3) int 4(byte) |
|  | 4)long 8(byte) |
|  | 5)float 4(byte) |
|  | 6)double 8(byte) |
|  | 7)char 2(byte) unsigned integers |
|  | 8)boolean 1(byte) |
|  | 7)Literal: |
|  | how to store the value |
|  | ->int literal |
|  | 1)DECIMAL LITERAL ex:int a=33; int b=678; |
|  | 2)Octal literal ex:int a=01; int b=02;-->always start with "0". |
|  | 3)Hexa decimal literal ex:int a=0x13 / 0X13; |
|  | -->Float literal: |
|  | ->float a=3.15f; or 3.15F; |
|  | -->double literal: |
|  | ->double a=3.15 or 3.14d or 3.14D; |
|  | -->short literal: |
|  | ->short a=10; |
|  | -->long literal: |
|  | -> long a=10 or 10l or 10L; |
|  | -->byte literal: |
|  | -> byte b=10; |
|  | -->boolean literal: |
|  | ->boolean b1=true/false; |
|  | -->character literal: |
|  | ->char a='c' / a=5 / a='\u0001'(unicode representation); |
|  | -->String literal or class: |
|  | ->String s="hello" / s=null; |
|  | 8)Variables: |
|  | identifier used to store a value 2types: |
|  | -->instance variable or class variable(any variable declared inside the class and outside the method). |
|  | ->no need to initialize it takes default value depending on datatype like , |
|  | int=short=byte=long=0; |
|  | float=double=0.0; |
|  | boolean=false; |
|  | object=null; |
|  | char='\u0000'; |
|  | -->local variable |
|  | -> any var declared inside the method and compulsarly it should be initialized otherwise compilation error. |
|  | 9)access specifiers/access modifiers: |
|  | -->4 access spec. |
|  | 1)default -->as like private but only with in package |
|  | 2)public -->accessed any where |
|  | 3)private -->only with in the class |
|  | 4)protected -->with in the inherited class can also in diff package |
|  | 10)Operators: |
|  | 1)arithmatic operator -> +,-,\*,/ |
|  | 2)modulus operator -> % |
|  | 3)relational operator -> >,<,<=,>= |
|  | 4)equality and unequality op -> ==,!= |
|  | 5)assignment operator -> = |
|  | 6)compound assignment operator -> +=,-=,\*=,/= |
|  | 7)increamnet/decreament ->,++,-- |
|  | 8)bitwise operator -> &,|,^,~(once complement),<<,>> |
|  | 9)logical operator -> (&,|,!) |
|  | 10)short circuit logical operator -> &&,|| |
|  |  |
|  |  |
|  | Execution control statements in java: |
|  |  |
|  | 1) conditional statements |
|  | -->if,if else |
|  | -->switch(exp) -->char,int,byte,short,enum,String |
|  | -->arg should always be final |
|  | -->continue statement should not be used |
|  | 2)Looping statement |
|  | -->for,while,do while. |
|  | 3)flow breaking statement |
|  | -->break(stop entire iteration) |
|  | ->labeled break |
|  | ->unlabeled break |
|  | -->continue(stops the current iteration) |
|  | ->labeled continue |
|  | ->unlabeled continue |
|  | -->return (transfers the control back to the calling program) |
|  |  |
|  | ARRAYS: |
|  |  |
|  | ->int a[]={1,2,3,4}; |
|  |  |
|  | ->int a[4]={1,2,3,4}; //wrong because whenever we give size we should use new keyword |
|  |  |
|  | ->int a[]; |
|  | a[0]=1; a[1]=2; |
|  |  |
|  | ->int[] a=new int[4]; |
|  |  |
|  | ->int a[]=new int[4]; |
|  |  |
|  | ->int a[]=new int[]{1,2,3,4}; //anonymous array |
|  |  |
|  | ->int a[]=new int[-4]; //throws negativeArraySize Exception |
|  |  |
|  | ->for each loop |
|  | for(int a1:a) //var should be declared only in for loop of same datatype |
|  | sop(a1); //for each stmt for only printing and we can't do any logic |
|  |  |
|  | 2D array: |
|  | ->int a[][]={{1,2},{3,4}}; |
|  |  |
|  | ->int[][] a=new int[4][4]; |
|  |  |
|  | ->int a[][]=new int[4][4]; |
|  |  |
|  | ->int[] a[]=new int[4][4]; //also a correct way |
|  |  |
|  | ->int a[][]=new int[][5]; //arrays of array |
|  | a[0]=new int[2]; |
|  | a[1]=new int[1]; //for this only 12 bytes will be allocated |
|  | a[0][0]=1; a[0][1]=2; a[1][0]=3; |
|  |  |
|  |  |
|  | for each for printing 2d elmts: |
|  | int a[][]={{1,2},{3,4}}; |
|  | for(int a1[]:a) |
|  | sop(a1[0]);1,3 // sop(a1[1]); 2,4 |
|  | for(int a2:a1) |
|  | sop(a2); 1,2,3,4 |
|  |  |
|  | STRING: |
|  |  |
|  | 1)String is a immutable class where we can't increase or decrease its size |
|  |  |
|  | constructor in string class: |
|  | 1)String() |
|  | 2)String(String s); |
|  | 3)String(byte[] b,int start,int end); |
|  | 4)String(char[] c,int start,int end); |
|  | 5)String(byte[] b); |
|  | 6)String(char[] c); |
|  |  |
|  | ex: |
|  | char c[]={'j','a','v','a'}; |
|  | String s=new String(c); |
|  | sop(s); //java |
|  |  |
|  | Methods: |
|  |  |
|  | 1)String to String() |
|  | ->string representation of object. |
|  | ex: |
|  | class A |
|  | { |
|  | String s; |
|  | A(String s1) |
|  | { |
|  | s=s1; |
|  | } |
|  |  |
|  | public String toString() { |
|  | return s; |
|  | } |
|  | } |
|  |  |
|  | public class Strings { |
|  |  |
|  | public static void main(String[] args) { |
|  | // TODO Auto-generated method stub |
|  |  |
|  | A a1=new A("hello"); |
|  | System.out.println(a1); |
|  | } |
|  |  |
|  | } |
|  |  |
|  | output:hello |
|  |  |
|  | 2) char charAt(int loc) |
|  | ->to return a single character from the string |
|  | 3) getChars() |
|  | ->returns group of characters. |
|  | 4) byte[] getBytes() |
|  | ->convert bytes to string array |
|  | 5)char[] toCharArray() |
|  | ->convert string to char array |
|  | String s="hello"; |
|  | char c[]=s.toCharArray(); |
|  | 6)startsWith(arg): |
|  | s.startsWith("he"); //true |
|  |  |
|  | 6)endsWith(arg): |
|  | s.endsWith("he"); //false |
|  | 7)boolean equals() |
|  | ->checks the equality of the content. |
|  | == |
|  | ->checks equality of object reference. |
|  |  |
|  | 8)String substring(int start),String substring(int start,int end) |
|  | s.substring(2,7); |
|  | ->here end means end-1, it will print from 2 to 6. |
|  |  |
|  | 9)int compareTo(String s) |
|  | compare two strings |
|  | if equals return 0,greater +value,lesser -value. |
|  | 10) int compareToIgnoreCase(String s) |
|  | same will be done but ignore case |
|  |  |
|  | 11) ->int indexOf(char c or String), |
|  | ->int lastIndexOf(char c or String) |
|  | ->int indexOf(char c or String,10) ->after the 10th position |
|  | ->int lastIndexOf(char c or String,10) ->after the 10th position |
|  |  |
|  | 12) ->String toUppercase() |
|  | ->String toLowercase() |
|  | ->String trim() |
|  | ->String concat(String s) |
|  | ->String replace(char original,char replace) |
|  | 13) String[] split(String delimiter) |
|  |  |
|  | String s1="one.two.three"; |
|  | String s2[]=s1.split("\\."); //if we want split by using special ncharacter then we have to use "\\" |
|  |  |
|  | for spliting using group of words: |
|  | String s="No con,no col, no give and take maoney"; |
|  | String r="con|col|(give and take)"; |
|  | String s1[]=s.split(r); |
|  | sop(s1); -->No |
|  | ,no |
|  | ,no |
|  | manoey |
|  | 14) static String format(String format,String value) |
|  | ex: sop(String.format("|%5d|",4)) |
|  | + ->add spce before value. |
|  | - ->add space after value. |
|  | 0 ->padding with 0. |
|  | w ->width. |
|  | p -> precision. |
|  |  |
|  | 15) boolean regionMatches(boolena ignorecase,int start,String value,int targetstart, int howmany) |
|  | ex: |
|  | String s="ABC Windows test"; |
|  | s.regionMatches(true,4,"windows",0,7); //true |
|  |  |
|  | Wrapper class: |
|  | ->the class thet supports primitive datatype to perform operation on that datatype. |
|  | ->all wrapper class also immutable. |
|  | Each datatype have seperate wrapper class |
|  | int --->Integer |
|  | byte --->Byte |
|  | char --->Character |
|  | float --->Float |
|  | double -->Double |
|  | boolean --->Boolean |
|  | short --->Short |
|  | long --->Long |
|  |  |
|  | Integer: |
|  | ->Constructors |
|  | 2 constructors |
|  | ->Integer(int a) |
|  | Integer i=new Integer(42); |
|  | sop(i); //it prints 42 because by default INteger override toString() |
|  | ->Integer(String s) |
|  | Integer i=new Integer("42"); |
|  | sop(i); //it prints 42 because by default INteger override toString() |
|  | ->Methods |
|  | 4 important methods |
|  | ->public static int parseInt(String s) |
|  | -> converts String to datatype. |
|  | ->public static int parseInt(String s,int radix) |
|  | ex:Integer.parseInt("42",5); //ouput: 22 expl:2\*5^0 + 4\*5^1 = 22 |
|  | ->int intValue() |
|  | ->coverts Integer wrapper class to int datatype |
|  | ->static Integer valueOf(String s) |
|  | ->converts String to Integer Wrapper class. |
|  | Byte,Short,Long: |
|  | ->Constructors |
|  | 2 constructors |
|  | ->Byte(byte b),.. |
|  | ->Byte(String s),.. |
|  | ->same methods as Integer wrapper class for Byte,Short,Long |
|  |  |
|  | Boolean: |
|  | ->constructors as same above |
|  | ->In methods if we pass string value other than true or false then it takes as false value. |
|  |  |
|  |  |
|  | Float: |
|  | ->constructors |
|  | ->Float(double d) |
|  | Float f=new Float(3.14); |
|  | ->Float(flaot f) |
|  | Float f=new Float(3.14f); |
|  | ->Flaot(String s) |
|  | Float f=new Float("3.14"); |
|  | ->Methods |
|  | ->static int compare(float f1,float f2) -> to compare two float datatype |
|  | float f1=3.14f,f2=3.14f; |
|  | if(f1==f2) //the condition is false so only we r using Float.compare(f1,f2)[in this case condition is true] |
|  | ->int compareTo(Float f) ->to compare two float wrapper class |
|  | Float f1=new Float(3.14); |
|  | Float f2=new Float(3.14); |
|  | f1.compareTo(f2); |
|  | ->static boolean isNaN() |
|  | Float f=new Float(Math.sqrt(-4)); |
|  | f.isNaN(); |
|  |  |
|  | Double: |
|  | ->constructor |
|  | ->Double(double d) |
|  | ->Double(String s) |
|  | ->Methods same as all |
|  | Character: |
|  | ->constructor |
|  | ->Character(char ch) |
|  | ->Methods |
|  | ->static boolean isLetter(char ch); |
|  | ->static boolean isDigit(char ch); |
|  | ->static boolean isLetterOrDigit(char ch); |
|  | ->static boolean isUppercase(char ch); |
|  | ->static boolean isLowerCase(char ch); |
|  | ->static boolean isWhiteSpace(char ch); |
|  | AUTOBOXING UNBOXING: |
|  | ->available only from jdk1.5 |
|  | ->automatic conversion datatype to wrapper class(AUTOBOXING) and wrapper class to datatype(UNBOXING). |
|  |  |
|  | With autoboxing |
|  | 1)Integer i; |
|  | int j; |
|  | i=5; |
|  | j=10; |
|  | j=i; |
|  | Without autoboxing |
|  | 1)Integer i; |
|  | int j; |
|  | i=new Integer(5); |
|  | j=10; |
|  | j=i.intValue(); |
|  |  |
|  |  |
|  | SCANNER: |
|  | Scanner s=new Scanner(System.in); |
|  | String s1=s.nextLine(); //we can get multiple word seperately with space |
|  | String s2=s.next(); |
|  | int i=s.nextInt(); |
|  |  |
|  | OOPs concepts: |
|  |  |
|  | This keyword(this): |
|  | ->refers to the current class reference variable. |
|  | ->used when parameter varibale name instance variable name are same. |
|  | this() constructor: |
|  | ->this() used to invoke different constructor of the same class |
|  | ->and always it should be present in the 1st line. |
|  |  |
|  | ex: |
|  | class A |
|  | { |
|  | A() |
|  | { |
|  | this(10); |
|  | } |
|  | A(int a) |
|  | { |
|  | } |
|  | A(String s) |
|  | { |
|  | A() |
|  | } |
|  | } |
|  | class Main |
|  | { |
|  |  |
|  | psvm() |
|  | { |
|  | A obj=new A("jeeva"); |
|  | } |
|  | } |
|  |  |
|  | Polymorphism:(many forms) |
|  | ->2 types |
|  | ->static or compile time polymorphism. |
|  | ->dynamic or run time polymorphism |
|  | Method overloading: |
|  | ->same method name but different number order of dtatype of arguments present in the same class.(no need to check return type). |
|  | ex: |
|  | class Overload |
|  | { |
|  | void test() |
|  | { |
|  |  |
|  | } |
|  | void test(int a) |
|  | { |
|  |  |
|  | } |
|  | void test(int a, int b) |
|  | { |
|  |  |
|  | } |
|  | void test(String a, int b) |
|  | { |
|  |  |
|  | } |
|  |  |
|  |  |
|  | } |
|  | public class Main2 { |
|  |  |
|  | public static void main(String[] args) { |
|  | // TODO Auto-generated method stub |
|  |  |
|  |  |
|  | Overload obj=new Overload(); |
|  | obj.test(); |
|  | obj.test(10); |
|  | obj.test(10,20); |
|  | obj.test("jeeva",20); |
|  |  |
|  | } |
|  |  |
|  | } |
|  | Constructor overloading: |
|  | ->the same as method but here it is constructor instead of method |
|  |  |
|  | var args: |
|  | ->passing variable number of arguments to the method. |
|  | -> it should always in the last of args. |
|  | ->In combination of normal arg and var args the var args should always at last |
|  | ->And a method can contain only one var args. |
|  | ->indicated with(datatype...var) |
|  | ex: |
|  | class Varargs |
|  | { |
|  | void test(int...a) |
|  | { |
|  |  |
|  | } |
|  | void test(boolean...b) |
|  | { |
|  |  |
|  | } |
|  | void test(int a,String...s) |
|  | { |
|  |  |
|  | } |
|  | } |
|  | public class Main1 { |
|  |  |
|  | public static void main(String[] args) { |
|  | // TODO Auto-generated method stub |
|  | Varargs v=new Varargs(); |
|  |  |
|  | v.test(1,2,3,4,5); |
|  | v.test(true,true,false,true); |
|  | v.test(10,"jeeva","nan","dha","m"); |
|  |  |
|  | } |
|  |  |
|  | } |
|  | non-access specifiers: |
|  | static keyword: |
|  | ->non-access specifiers |
|  | ->it has its own restriction whereas access specifier can be applied for anything |
|  | ->when a class is declared to be static then no need to create object |
|  | ->only inner class can be static outer class should not be static |
|  | ->when a method is declared to be static if it is present in the same class it is invoked by method name,different class means then invoked by classname.methodname |
|  | ->if a variable is declared as static it will be initialized only once and invoked as variable name in the same class,classname.variablename if it is in different class |
|  | ->static method can access only static content otherwise,we have to object an object and access |
|  | ->static block static{} will be executed before main method |
|  | ex: |
|  | public class Main1 { |
|  |  |
|  | static //executed first |
|  | { |
|  |  |
|  | } |
|  | public static void main(String[] args) { |
|  |  |
|  |  |
|  | } |
|  |  |
|  | } |
|  |  |
|  | System.out.println(); |
|  | System -> class |
|  | out -> object of PrintStream class |
|  | println()-> (static)method in PrintStream class |
|  | behind the above statement ->public class System |
|  | { |
|  | static PrintStream out; |
|  | static PrintStream in; |
|  | } |
|  |  |
|  | static import: |
|  | ->available from jdk(1.5) where we can call or invoke static method and ststaic variables directly without using class name. |
|  | ex: |
|  | import static java.lang.Math.\*; |
|  | import static java.lang.System.\*; |
|  | double d=sqrt(16); |
|  | out.println(d); |
|  |  |
|  | INHERITANCE: |
|  | ->accessing the properties of one class in another class |
|  | ->advantage of inheritance is code reusability |
|  | 5 types of inheritance |
|  | ->single |
|  | ->multilevel |
|  | ->hierarchial |
|  | ->hybrid |
|  | ->multiple(not possible in java) |
|  |  |
|  | SINGLE INHERITANCE: |
|  |  |
|  | ex: |
|  | class A |
|  | { |
|  | int a=120; |
|  | } |
|  | class B extends A |
|  | { |
|  | int b=130; |
|  | } |
|  | public class Main1 { |
|  |  |
|  | public static void main(String[] args) { |
|  | B obj=new B(); //creating the object for child or derived class |
|  | System.out.println(obj.a); //prints 120 |
|  | } |
|  |  |
|  | } |
|  |  |
|  | super(): |
|  | ->used to access base or parent class constructor. |
|  | ->only in the case of base class constructor |
|  | ->super() should be used in the first line only |
|  | ex: |
|  | class A |
|  | { |
|  |  |
|  | A() |
|  | { |
|  | System.out.println("invoked using super() method"); |
|  | } |
|  |  |
|  | } |
|  | class B extends A |
|  | { |
|  |  |
|  | B() |
|  | { |
|  | super(); |
|  | } |
|  |  |
|  | } |
|  | public class Main1 { |
|  |  |
|  | public static void main(String[] args) { |
|  | B obj=new B(); //creating the object for child or derived class |
|  | } |
|  |  |
|  | } |
|  | output: invoked using super() method |
|  |  |
|  | MULTI-LEVEL INHERITANCE: |
|  |  |
|  | class A |
|  | { |
|  |  |
|  | } |
|  | class B extends A |
|  | { |
|  |  |
|  | } |
|  | class C extends B |
|  | { |
|  |  |
|  | } |
|  | public class Main1 { |
|  |  |
|  | public static void main(String[] args) { |
|  | C obj=new C(); //creating the object for child or derived class |
|  | } |
|  |  |
|  | } |
|  |  |
|  | super keyword: |
|  | for accessing parent class variables or method. |
|  | class A |
|  | { |
|  | int i; |
|  | } |
|  | class B extends A |
|  | { |
|  |  |
|  | B() |
|  | { |
|  | super.i=2; |
|  | } |
|  |  |
|  | } |
|  | public class Main1 { |
|  |  |
|  | public static void main(String[] args) { |
|  | B obj=new B(); //creating the object for child or derived class |
|  | } |
|  |  |
|  | HOW CONSTRUCTORS ARE INVOKED IN INHERITANCE: |
|  | ->in inheritance the constructors are always invaoked as top-down approach. |
|  | ->first it goes to the related class constructor and check we used this or super otherwise it tries to invoke only the default constructor of the base class. |
|  | ex: |
|  | class A |
|  | { |
|  | A() |
|  | { |
|  | System.out.println("A CLASS CONSTRUCTOR"); |
|  | } |
|  | } |
|  | class B extends A |
|  | { |
|  |  |
|  | B() |
|  | { |
|  | System.out.println("B CLASS CONSTRUCTOR"); |
|  | } |
|  | } |
|  | class C extends B |
|  | { |
|  |  |
|  | C() |
|  | { |
|  | System.out.println("C CLASS CONSTRUCTOR"); |
|  | } |
|  | } |
|  | public class Main1 { |
|  |  |
|  | public static void main(String[] args) { |
|  | C obj=new C(); |
|  | } |
|  |  |
|  | } |
|  | output : A CLASS CONSTRUCTOR |
|  | B CLASS CONSTRUCTOR |
|  | C CLASS CONSTRUCTOR |
|  |  |
|  | IS A RELATIONSHIP: |
|  | ->wherever inheritance comes, that type of rellationship is called "is a relationship" |
|  | ex: |
|  | class Animal |
|  | { |
|  |  |
|  | } |
|  | class Dog extends Animal |
|  | { |
|  |  |
|  | } |
|  | HAS A RELATIONSHIP: |
|  | ex: |
|  | class Wheel |
|  | { |
|  |  |
|  | } |
|  | class Car extends Wheel |
|  | { |
|  | Wheel obj=new Wheel(); //whenever we create an object for the extended class it is called "has a relatioship" |
|  | } |
|  | DYNAMIC POLYMORPHISM: |
|  | METHOD OVERRIDING: |
|  | ->same method name,same return type,same order and datatype of the argument present in different class and the class should be inherited. |
|  | class A |
|  | { |
|  | void show() |
|  | { |
|  | System.out.println("base class method"); |
|  | } |
|  | } |
|  | class B extends A |
|  | { |
|  |  |
|  | void show() |
|  | { |
|  | //super.show(); -> we use this inorder to invoke show() in parent class. |
|  | System.out.println("derived class method"); |
|  | } |
|  |  |
|  | } |
|  |  |
|  | public class Main1 { |
|  |  |
|  | public static void main(String[] args) { |
|  | B obj=new B(); |
|  | obj.show(); |
|  | } |
|  |  |
|  | } output:derived class method |
|  |  |
|  | ->in order to have effective method overriding we have "dynamic method dispatch",wher we create object for the base class but we store reference of derived class. |
|  | ->at compilation the compiler thinks object is created for base class only at the time of execution it knows the object contains the reference of derived class in this way we achieve runtime polymorphism |
|  |  |
|  | ex: |
|  | class A |
|  | { |
|  | void show() |
|  | { |
|  | System.out.println("base class method"); |
|  | } |
|  | } |
|  | class B extends A |
|  | { |
|  |  |
|  | void show() |
|  | { |
|  | System.out.println("derived class method"); |
|  | } |
|  |  |
|  | } |
|  | class C extends A |
|  | { |
|  |  |
|  | void show() |
|  | { |
|  | System.out.println("derived class method"); |
|  | } |
|  |  |
|  | } |
|  |  |
|  | public class Main1 { |
|  |  |
|  | public static void main(String[] args) { |
|  | A a=new A(); //object is created for A |
|  | a.show(); |
|  | a=new B(); //now the object a refers to class B |
|  | a.show(); |
|  | a=new C(); //now the object a refers to class C |
|  | a.show(); |
|  | } |
|  | } |
|  | output: |
|  | base class method |
|  | derived class method |
|  | derived class method |
|  |  |
|  | Final: |
|  | ->non access specifier |
|  | ->if a class is declared as final it cannot be inherited |
|  | ->if a method is declared as final |
|  | ->if avariable is declared as final it cannot be changed |
|  |  |
|  | Acessors(getters) and mutators(setters): |
|  | ex: |
|  | class Employee |
|  | { |
|  | private String name; |
|  | private int age; |
|  | public String getName() { |
|  | return name; |
|  | } |
|  | public void setName(String name) { |
|  | this.name = name; |
|  | } |
|  | public int getAge() { |
|  | return age; |
|  | } |
|  | public void setAge(int age) { |
|  | this.age = age; |
|  | } |
|  | } |
|  | public class Main2 { |
|  |  |
|  | public static void main(String[] args) { |
|  | // TODO Auto-generated method stub |
|  | Employee obj=new Employee(); |
|  | obj.setName("xxxx"); |
|  | obj.setAge(21); |
|  | System.out.println(obj.getName()+" "+obj.getAge()); |
|  |  |
|  | } |
|  |  |
|  | } |
|  |  |
|  | OBJECT class: |
|  | ->it is the super class of all the classes |
|  | constructor: |
|  | ->Object() |
|  | Methods: |
|  | ->String toString() |
|  | ->boolean equals(Object o) |
|  | ->int hashCode() |
|  | \*)return internal addrss of the value stored in the object. |
|  | \*)if 2 objects are equal according to equals method hasCode method produces the same integer. |
|  | ->protected void finalize() |
|  | ->final void wait() |
|  | ->final void wait(Long l) |
|  | ->final void notify() |
|  | ->final void notifyAll() |
|  | Abstract keyword: |
|  | ->it is a non-access specifier |
|  | ->when a class is declared to be abstract it cannot be intanciated(Object cannot be created) |
|  | ->when a method is declared to abstract it does not contain definition just it ends with semi-colon(;) |
|  | ->variable cannot be abstract. |
|  | ->A class should be declared as abstract if it contains an abstract method but not neccessarily all the abstract class should have abstract method. |
|  | ->abstract class can also contain some normal methods |
|  | ->the abstract class can also be inherited,at the time we have to compulsarly give the defenition of the abstract method in the inherited class or defined class itself to be abstract |
|  | ->abstract contains the default constructor but it cannot be invoked because the object creation is not possible. |
|  | ex: |
|  | abstract class A2 |
|  | { |
|  | abstract void show(); |
|  | void show2() |
|  | { |
|  | System.out.println("normal method"); |
|  | } |
|  | } |
|  |  |
|  | class B2 extends A2 |
|  | { |
|  | void show() |
|  | { |
|  | System.out.println("Abstract method"); |
|  | } |
|  | } |
|  |  |
|  | public class Strings { |
|  |  |
|  | public static void main(String[] args) { |
|  | // TODO Auto-generated method stub |
|  |  |
|  | B2 obj=new B2(); |
|  | obj.show(); |
|  | obj.show2(); |
|  | } |
|  |  |
|  | } |
|  | output: |
|  | Abstract method |
|  | normal method |
|  | Interface: |
|  | ->instead of multiple inheritance we use interface |
|  | ->interfaces are syntactically similar to classes which contains method declaration and variable declaration and initialization |
|  | ->syntax ->accessspecifier interface interfaceName{ |
|  | method declaration or variable declaration |
|  | } |
|  | ->by default all interfaces are abstract so we cannot create object for interface. |
|  | ->by default all interface methods are public and abstract |
|  | ->by default all interface variables are public,static,and final |
|  | -> so interface variables accessed by interfacename.variablename |
|  | ->Interface are used using implements keyword |
|  | ->If a class implements an interface we have to compulsarily provide the definition of interface methods with public access specifier or abstract |
|  | ->the implemented class can also contain some normal method |
|  | ->interfaces can also be inherited |
|  | ->in java we have |
|  | \*)one class extends one class |
|  | \*)one class implements many class |
|  | \*)one interface extends many interface |
|  | ->An interface without any methods and variables is called "marker interface" |
|  | ex: serializable,clonable,remote..etc,. |
|  | ex: |
|  | Interface: |
|  |  |
|  | public interface Maths { |
|  | void arithmatic(int a,int b); |
|  | int c=20; |
|  | } |
|  |  |
|  | classes: |
|  | class A9 implements Maths |
|  | { |
|  |  |
|  | @Override |
|  | public void arithmatic(int a, int b) { |
|  | // TODO Auto-generated method stub |
|  | System.out.println(a+b+" "+Maths.c); |
|  | } |
|  |  |
|  | } |
|  | class B9 implements Maths |
|  | { |
|  |  |
|  | @Override |
|  | public void arithmatic(int a, int b) { |
|  | // TODO Auto-generated method stub |
|  | System.out.println(a-b+" "+Maths.c); |
|  | } |
|  | void show() |
|  | { |
|  | System.out.println("Non implemented method"); |
|  | } |
|  |  |
|  | } |
|  | public class Main2 { |
|  |  |
|  | public static void main(String[] args) { |
|  | // TODO Auto-generated method stub |
|  | Maths m=new A9(); |
|  | m.arithmatic(10,20); |
|  | m=new B9(); |
|  | m.arithmatic(20, 10); |
|  | ((B9) m).show(); |
|  | } |
|  |  |
|  | } |
|  | output: |
|  | 30 20 |
|  | 10 20 |
|  | Non implemented method |
|  |  |

**sss**

**Exceptional Handling in java:**

2 types of exception:

* **Checked Exception**
* sub classes of exception ,it will insist the programmer to surround the code using try catch or throws otherwise the compiler will not compile the program.
* **Unchecked Exception**
* Sub classes of run time exception the compiler will not insist the programmer to surround the code using try catch or throws but at the time of execution it throws the exception.
* There are different types of unchecked exception.

**Arithmetic exception:**

Any number divided by zero.

**Array index out of bound exception:**

**String index out of exception:**

Applicable only for string array

**Array store exception:**

Ex:

int[] a=new int[4];

a[3]=”hello”;

**Negative array size exception:**

Ex:

int[] a=new int[-4];

**class cast exception:**

ex:

class A

{

}

Class B extends A

{

}

Class C{ psvm{

A a=new A();

B b=new B();

a=b;

b=a //exception occur

}}

**Null pointer exeption:**

Ex:

A a=new A();

a.add();

a=null;

a.add(); //exception occur

**Number format exception:**

**Illegal monitor state exception**

**Illegal thread state exception**

These 2 occur in thread

**Keywords to handle exception:**

**try:**

* Program to be monitored is put inside try block.

**Catch:**

* Used to caught the exception.

**Finally:**

* It is an optional statement used to close the resources, if it is used whether exception occur or not it will be executed after the try block
* Mainly used in file programming , database programming , socket programming

**There are 3 ways of using possiblities of exception:**

try{ } catch{ } finally{}

try{ } catch{ }

try{ } finally{}

**Multi catch statement:**

Single try can have multiple catches.

Ex:

**package** exception;

**public** **class** Arithmetic {

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

// **TODO** Auto-generated method stub

**try**

{

**int** d=42/0;

**int** c[]= {1};

c[33]=42;

}

**catch**(ArrayIndexOutOfBoundsException e)

{

System.***out***.println(e);

}

**catch**(ArithmeticException e)

{

System.***out***.println(e);

}

**finally**

{

}

}

}

**Multi catch in a single catch:**

**catch**(ArrayIndexOutOfBoundsException | ArithmeticException e)

{

System.***out***.println(e);

}

**Throw:**

Used to manually throw an exception

**Syntax:**

throw new arithmeticException();

**ex:**

**package** exception;

**public** **class** Arithmetic {

**static** **void** demoproc()

{

**try**

{

**throw** **new** NullPointerException();

}

**catch**(NullPointerException e)

{

System.***out***.println("caught "+e);

**throw** e;

}

}

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

// **TODO** Auto-generated method stub

**try**

{

*demoproc*();

}

**catch**(NullPointerException e)

{

System.***out***.println("recaught "+e);

}

}

}

**Output:**

caught java.lang.NullPointerException

recaught java.lang.NullPointerException

**throws:**

* used to throw the exception and executes as usual ,it can be used only in methods

**User defined exception/custom exception:**

Ex:

**package** exception;

**class** MyException **extends** Exception

{

**private** **int** detail;

**public** MyException(**int** a) {

// **TODO** Auto-generated constructor stub

detail=a;

}

@Override

**public** String toString() {

**return** "MyException [detail=" + detail + "]";

}

}

**public** **class** Main{

**static** **void** compute(**int** a) **throws** MyException

{

System.***out***.println("called compute " +a);

**if**(a>10)

**throw** **new** MyException(a);

System.***out***.println("Normal exit");

}

**public** **static** **void** main(String[] args)

{

**try**

{

*compute*(1);

*compute*(20);

}

**catch**(Exception e)

{

System.***out***.println("caught "+e);

}

}

}

**Output:**

called compute 1

Normal exit

called compute 20

caught MyException [detail=20]

**COLLECTION:**

**Arraylist class:**

* It is the dynamic array used to increase or decreese its size at run time.
* Used for faster selection and slower insertion and deletion.
* Default capacity is 10.

**Constructor:**

* ArrayList()
* ArrayList(int capacity)
* ArrayList(Collection c)

**Method:**

* void add(Object obj)
* void add(int index,Object obj)
* void addAll(Collection c)
* void addAll(int index,Collection c)
* Object get(int index)
* void remove(int index)
* Object set(int index,Object obj)
* int indexOf(Object obj)
* int lastIndexOf(Object obj)
* boolean contains(Object obj)
* int size()
* boolean isEmpty()

**ex:**

List<Employee> l=new ArrayList<Employee>(); //generics from jdk1.5

List<Employee> l1=new ArrayList<>(); //jdk1.7

for accessing the elements in the list we have 4 ways:

* naormal for loop
* for each

**for**(String s:l1)

{

System.***out***.println(s);

}

* iterator

Iterator<String> itr=l1.iterator();

**while**(itr.hasNext()) //boolean hasNext() --- check wheather it contains next element

{

System.***out***.println(itr.next()); //Object next() --- returns single element

}

* list iterator

->ListIterator interface used to access individual elmts of collection both forward and backward

ListIterator<String> litr=l1.listIterator();

**while**(litr.hasPrevious())

{

System.***out***.println(litr.previous());

}

**LinkedList:**

* similar to arraylist
* faster insertion and deletion and slower selection
* because linkedlist implements list,deque(double ended queue)

List<Integer> ll=**new** LinkedList<>();

**Vector:**

* similar to arraylist
* but it is synchronized or thread safe
* it is the legacy(old) class
* default capacity is 10

**constructor:**

* Vector()
* Vector(int capacity)
* Vector(int capacity,int increament)
* Vector(Collection c)

**Set Interface:**

3 classes

**HashSet:**

* used to print the elements in the random order
* default capacity is 16.
* whenever we add the elements into the hashset internally it creates the map which contains the inserted as a key and a dummy object for that value.

syntax:

Set<String> s=new HashSet();

constructors:

* HashSet();
* HashSet(int capacity);
* HashSet(int capacity,float fillRatio);
  + fillRatio ranges from 0.0 to 1.0
  + if the capacity is less than the no. of elmt it multiplied by fillRtaio in order to increase the capacity.
* HasSet(Collection c);

LinkedHashSet:

* prints in the sameorder as we give.

syntax:

Set<String> s=new HashSet();

TreeSet:

* Used to sort the elmts in alphabetical and ascending order,there is no descending.

Map Interface:

* Used to store the collection of object as a unique key value pair.
* Map is unordered.

Map.Entry Interface:

* Used to access the key and value separately

Methods:

* void put(Object key,Object value)
* void putAll(Map m)
* Object get(Object key)
* void remove(Object key)
* boolean containsKey(Object key)
* boolean containsValue(Object value)
* Set entrySet() 🡪converts both key and values to Set interface
* Set keyset() 🡪converts only key to Set interface

HashMap:

* contains unique key value pair and prints in random order.

Constructors:

4 constructors:

1)HashMap()

2)HashMap(int capacity)

3)HashMap(int capacity,float fillRatio)

4)HashMap(Map m)

ex:

**package** exception;

**import** java.util.\*;

**public** **class** map {

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

// **TODO** Auto-generated method stub

Map<String,Double> hm=**new** HashMap<>();

hm.put("john", 235.88);

hm.put("jane baker",777.99);

hm.put("john", 235.88);

hm.put("jjjj",77997.99);

System.***out***.println(hm.size());

System.***out***.println(hm);

//printing using entrySet()

Set s=hm.entrySet();

Iterator i=s.iterator();

**while**(i.hasNext())

{

Map.Entry me=(Map.Entry)i.next();

System.***out***.println(me.getKey()+" "+me.getValue());

}

//printing using keySet()

s=hm.keySet();

i=s.iterator();

**while**(i.hasNext())

{

String s1=(String)i.next();

System.***out***.println(s1+" "+hm.get(s1));

}

}

}

output:

3

{jjjj=77997.99, john=235.88, jane baker=777.99}

jjjj 77997.99

john 235.88

jane baker 777.99

jjjj 77997.99

john 235.88

jane baker 777.99

Linked HashMap:

* contains key value pair and prints in the order that we have inserted
* Map<String,Double> hm=**new** LinkedHashMap<>();

TreeMap:

* used to sort the elmts based on key.
* Map<String,Double> hm=**new** TreeMap<>();

HashTable:

* it is a legacy class and similar to hashmap but it is synchronized or thread safe.
* it uses enumeration

ex:

Enumeration e=hm.keys();

**while**(e.hasMoreElements())

{

String s2=(String)e.nextElement();

System.***out***.println(s2+" "+hm.get(s2));

}

Properties class:

* it is the sub calss of hashtable contains unique key value pair both should be in the form of string

Date class:

* used to print date and time

constructors:

1)Date() 🡪prints current date and time

2)Date(long ms) 🡪prints date and time from jan 1st 1970.

methods:

1)int compare(date d)

2)boolean before(date d)

3)boolean after(date d)

4)void setTime(long ms)

5)long getTime()

ex:

**package** exception;

**import** java.util.Date;

**public** **class** Dat {

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

// **TODO** Auto-generated method stub

Date d1=**new** Date();

System.***out***.println(d1);

Date d2=**new** Date(100000);

System.***out***.println(d1.compareTo(d2));

System.***out***.println(d1.before(d2));

System.***out***.println(d1.after(d2));

System.***out***.println(d1.getTime());

}

}

output:

Tue Jul 23 10:39:24 IST 2019

1

false

true

1563858564138

Calendar:

* claendar is an abstract class used to extract the useful information from date and time component

constructor:

1)Calendar()

Method:

1)public static Calendar getInstance()

2)public int get(int field)

3)public int add(int field,int value)

4)public int set(int field,int value) 🡪replace

5)public boolean before(Calendar c)

6) public boolean afetr(Calendar c)

7)public int getActualMaximum(int field)

8) public int getActualMiniimum(int field)

9)public int getMaximum(int field)

10) public int getMinimum(int field)

11)final Date getTime() 🡪converts calendar to date

12) final Date setTime() 🡪converts date to calendar

Gregorian calendar:

* concrete implementation(we can create an object) of calendar class used to extract information from date time component

constructor:

1)GregorianCalendar() 🡪prints current date and time

2) GregorianCalendar(int year,int month,int day)

3) GregorianCalendar(int year,int month,int day,int hh,int min,int sec)

methods:

1)boolean isLeapYear(int year)

same as above calendar class methods….

Constants:

Ex:

**package** exception;

**import** java.util.Calendar;

**import** java.util.Date;

**public** **class** Cal {

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

// **TODO** Auto-generated method stub

Calendar c= Calendar.*getInstance*();

System.***out***.println(c.getTime());

Date d1= **new** Date(20000);

Calendar c1=Calendar.*getInstance*();

c1.setTime(d1);

System.***out***.println(c1.getTime());

System.***out***.println(c.get(Calendar.***YEAR***));

System.***out***.println(c.get(Calendar.***MONTH***));

System.***out***.println(c.get(Calendar.***DATE***));

c.add(Calendar.***YEAR***,-4);

System.***out***.println(c.get(Calendar.***YEAR***));

c.set(Calendar.***MONTH***,4);

System.***out***.println(c.get(Calendar.***YEAR***));

System.***out***.println(c.get(Calendar.***MONTH***));

System.***out***.println(c.getMaximum(Calendar.***YEAR***));

System.***out***.println(c.getActualMaximum(Calendar.***YEAR***));

System.***out***.println(c.getActualMinimum(Calendar.***YEAR***));

}

}

Output:

Tue Jul 23 11:47:07 IST 2019

Thu Jan 01 05:30:20 IST 1970

2019

6

23

2015

2015

4

292278994

292278994

1

simpleDateFormat class:

* present in java.text package used for formatting and parsing the date

constructor:

1)simpleDateFormat()

2) simpleDateFormat(String pattern)

Types of pattern:

1)G

🡪represents the era A.D or B.C

2)y

🡪represents year in no.

3)Y

🡪represents the week of year

4)M

🡪month in year

5)W

🡪week in month

6)D

🡪day in year.

7)d

🡪day in month

8)E

🡪day name in week.

9)a

🡪A.M or P.M

10)H

🡪hour in day 0 to 23

11)k

🡪hour in day 1 to 24.

12)K

🡪hour in A.M or P.M 0 to 11

13)h

🡪hour in A.M or P.M 1 to 12

14)m

🡪minutes

15)s

🡪seconds

16)S

🡪millin seconds

Methods:

1)void setLeaniant(boolean on)

2)Date parse(String s)

3)String format(Date d)

Ex:

**package** exception;

**import** java.text.SimpleDateFormat;

**import** java.util.Date;

**import** java.util.Scanner;

**public** **class** StringFor {

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

// **TODO** Auto-generated method stub

Date d=**new** Date();

System.***out***.println(d);

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter the dob:");

String dat=sc.nextLine();

SimpleDateFormat sd=**new** SimpleDateFormat("dd/mm/yy");

sd.setLenient(**false**);

**try**

{

Date d1=sd.parse(dat);

System.***out***.println(d1);

}

**catch**(Exception e)

{

System.***out***.println(e);

}

SimpleDateFormat sd1=**new** SimpleDateFormat("dd-MM-yy");

String s2=sd1.format(d);

System.***out***.println(s2);

}

}

**Java.io.\*:**

2 classes:

1)File

🡪access info about the existing file.

2)Stream

🡪perform operation on file.

Stream divided into 2 classes:

1)ByteStream ->read and write in bytes

🡪InputStream ->abstract class and read in bytes

->FileInputStream

🡪OutputStream ->write in bytes and abstract class

->FileOutputStream

2)CharacterStream ->read and write in char

🡪Reader ->read in char and abstract class

->FileReader

🡪Writer ->write in char

->Writer

Ex: for file

**package** exception;

**import** java.io.File;

**public** **class** File1 {

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

// **TODO** Auto-generated method stub

File f=**new** File("C:\\file\\text.txt");

System.***out***.println(f.getName());

System.***out***.println(f.getPath());

System.***out***.println(f.getAbsolutePath());

System.***out***.println(f.getParent());

System.***out***.println(f.isFile());

System.***out***.println(f.isDirectory());

System.***out***.println(f.length());

System.***out***.println(f.lastModified());

System.***out***.println(f.canWrite());

System.***out***.println(f.canRead());

System.***out***.println(f.isHidden());

}

}

Reader:

* used to read in the form of characters
* int read() 🡪used to read single character
  + - return -1 at end of file
* int read(char[] c) 🡪used to read array of characters and returns no.of bytes
* int read(char[] c,int offset ,int numchar)

File Reader:

* extends Reader class
* used to read the data from file in the form of characters
* if the file does not exist it throws fileNotFound exception

Constructors:

1)FileReader(File path)

2)FileReader(String directorypath,String filename)

3)FileReader(File f)

Ex:

**package** exception;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** Readerclass {

**public** **static** **void** main(String[] args) **throws** IOException{

// **TODO** Auto-generated method stub

FileReader f=**new** FileReader("C:\\file\\text.txt");

**int** i;

**while**((i=f.read())!=-1)

{

System.***out***.print((**char**)i);

}

f.close();

}

}

Output:

hi welcome.hhg

Writer class:

* abstract class used to write in the form of characters.

Methods:

1)void write(int b)

2)void write(char[] c,int offset, int numchars)

3)void close()

4)void flush()

File writer:

* file writer extends writer class.
* Used to write in the file in the form of characters
* If the file does not exist it creates new file
* If it exists it overwrites the content

Constructor:

1)FileWriter(String filepath)

2) FileWriter(String filepath,boolean append)

Ex:

**package** exception;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** FileWriterclass {

**public** **static** **void** main(String[] args) **throws** IOException {

// **TODO** Auto-generated method stub

FileWriter fw=**new** FileWriter("C:\\file\\text.txt",**true**);

fw.write("welcome to java file");

fw.close();

}

}

Output:

Content will be appended in the text.txt file

StringBuffer:

* Present in java.lang.\*
* It is a mutable class with variable number of characters
* It is synchronized or thread safe and we cannot override the equals method in string buffer

Constructors:

1)StringBuffer()

2)StringBuffer(String s)

3)StringBuffer(int capacity) 🡪defauly capacity is 16

Methods:

🡪void append(String s);

🡪void append(int a)

🡪void append(Object o)

🡪void delete(int start,int end)

🡪void deleteCharAt(int loc)

🡪void replace(int start,int end,String s)

Ex:

**package** exception;

**public** **class** StrgBuffer {

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

// **TODO** Auto-generated method stub

StringBuffer sb=**new** StringBuffer("hello");

System.***out***.println(sb.capacity());

System.***out***.println(sb.length());

System.***out***.println(sb.charAt(1));

sb.setCharAt(1,'i');

System.***out***.println(sb);

sb.setLength(2);

System.***out***.println(sb);

System.***out***.println("----------------------------------");

**int** a=42;

StringBuffer sb2=**new** StringBuffer(40);

String s=sb2.append("a = ").append(a).append("!").toString();

System.***out***.println(s);

sb2.reverse();

System.***out***.println(sb2);

}

}

Output:

21

5

e

hillo

hi

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

a = 42!

!24 = a

StringBuilder:

* Similar to string buffer but is not synchronized but it gives better performance.
* All others are same as stringBuffer.

Threading:

* Used to run multiple lines of code in a single program simultaneously using thread class

Syntax:

Public class Thread extends Object implements Runnable

{

}

Constructors:

1)Thread()

2)Thread(String threadName)

3)Thread(Runnable r)

4) Thread(String threadName,Runnable r)

5)Thread(threadgroup tg,String threadName)

6) Thread(threadgroup tg,Runnable r)

7) Thread(threadgroup tg,Runnable r,String threadName)

Constants:

threadPriority.MIN=1

thread.Priority.MAX=10

thread.Priority.NORM=5

methods:

public void start()

public static void sleep(long ms)

2 types of thread:

1)Main thread

🡪wherever the psvm method presents by default the thread will be running called Main thread.

🡪It will be executed first and blocked to invoke the child thread.

2)Child thread:

🡪We create n number of child thread

Life cycle of child thread:  
5 stages in child thread

Creation of child thread:☹(2 ways)

* By implementing runnable interface

Ex:

**package** exception;

**class** NewThread **implements** Runnable

{

Thread t;

NewThread()

{

t=**new** Thread(**this**,"Demo thread");

System.***out***.println(t);

t.start(); // create new thread and put it in runnable stage

}

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

**try**

{

**for**(**int** i=5;i>0;i--)

{

System.***out***.println("Child Thread"+i);

Thread.*sleep*(500);

}

}

**catch**(Exception e)

{

System.***out***.println("Child thread interrupted");

}

}

}

**public** **class** Threadclass {

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

// **TODO** Auto-generated method stub

**new** NewThread();

**try**

{

**for**(**int** i=5;i>0;i--)

{

System.***out***.println("Main Thread"+i);

Thread.*sleep*(1000);

}

}

**catch**(Exception e)

{

System.***out***.println("Main thread interrupted");

}

System.***out***.println("Main thread exiting");

}

}

Output:

Thread[Demo thread,5,main]

Main Thread5

Child Thread5

Child Thread4

Main Thread4

Child Thread3

Child Thread2

Main Thread3

Child Thread1

Main Thread2

Main Thread1

Main thread exiting

* It just acts like a child thread to create a thread any one of 7 constructor

Runnable contains one method

Public void run() 🡪which contains the operation or the processing of child thread.

Ex:

**class** NewThread1 **extends** Thread

{

Thread t;

NewThread1()

{

**super**("demo thread");

System.***out***.println(**this**);

start();

}

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

**try**

{

**for**(**int** i=5;i>0;i--)

{

System.***out***.println("Child Thread"+i);

Thread.*sleep*(500);

}

}

**catch**(Exception e)

{

System.***out***.println("Child thread interrupted");

}

}

}



Synchronization:

* Whem multiple thread access the single resource the thread has to utilize the resource completely or fullfledged.
* It can be achieved used synchronized keyword.
* It is a non-access specifier.
* It can be applied only to methods or synchronized block of objects.

🡪synchronized(Objects){}

Annotation:

* Available from jdk1.5
* Annotations are like meta data means u r free add ur code and also apply them to variables parameters methods constructors and class.
* Annotations are used instead of xml file.
* Annotation contain 2 things

🡪1)Annotation type ->creation of annotation

Syntax:

Public @Interface annotation\_name()

* Method declaration should not through any exception
* Method declarartion should not contain any parameter
* Method declaration should have a return type either String,datatype,object
* 🡪2)Annotaion:Applying the created annotation in the java program either class level or method level

🡪2 type of annotation

Simple annotation:

* Present in java.lang.\* package and it can be applied only on the java program
* 3 types,

🡪@override

Ensures the annotated method is used to override the method in the super class

🡪@deprecated

This type of annotation the compiler warns u when u r using deprecated elmts of the program.

🡪@SuppressWarning

This type of annotation ensures that the compiler will shield the warning message in the annotated element.

Meta Annoataion:

* Annotation about annotation and at the time of creating a new annotation present in java.lang.annotation.\* package

🡪@Target

Specifies where the annotation should be applied in the java program like

🡪 @target(ElementType.METHOD)

🡪@target(ElementType.CONSTRUCTOR)

🡪@target(ElementType.VARIABLE)

🡪@target(ElementType.TYPE)

🡪@retention:specifies where and how long annotation with this type are to be retained like

🡪 @retention(RetentionPolicy.SOURCE)

🡪@retention(RetentionPolicy.CLASS)

🡪 @retention(RetentionPolicy.ANNOTATION)

🡪@documentation

->Javadoc is used to document only methods variable and constructors of a class in html file

->@documented is used to document the annotation also using java doc tool

Testing:

JUNIT:

* It is a unit testing opensource framework for java programming language used to test the individual classes
* Configure the Junit jar inside the project
* Org.junit.assertclass provides set of assertion methods useful for writing test

1)Void assertEquals(boolean expected,boolean actual)

2)Void assertFalse(boolean condition)

3)void assertNotNull(Object obj)

4)void assertNull(Object obj)

5)void assertTrue(boolean condition)

Junit annotation:

1)@Test 🡪this is the test method to run

2)@before 🡪run before @test

3)@after 🡪run after @test

4)@beforeClass 🡪run once before any of the test method

5)@afterClass 🡪run once after any of the test method

Ex:

package exception;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

class JunitExample {

@Test

void test() {

String str1=new String("hello");

String str2=new String("hello");

String str3=null;

String str4="hello";

String str5="hello";

int a=5,b=6;

assertEquals(str1, str2);

assertTrue(a<b);

assertFalse(a>b);

assertNotNull(str2);

assertNull(str3);

assertSame(str4, str5);

}

}