Day2

J2SE :

JDK s/w

Java 8 or above

Java developer : JDK

Java : JRE

Demo.java -----

public Demo1

Demo2

Demo3

Demo.java

class Demo{

}

Demo.java -----------> compiler ------Demo.class------>JVM

Developing Java Application in IDE (STS):

=================================

in java all classes should be inside a package.

package name convension is reverse domain name

com.masai.service

100 --- business logic

com.masai.dao

500 ---data base access code

com.masai.controller

400 --- controller flow

com

masai

service

first keyword using which we should start a java application :

package

type few charecter + crtl + spacebar

main + crtl+ spacebar

public static void main(String[] args)

Class and Interface : should start with capital latter . (PascalNamingConvension)

--any thing start with capital letter it would be either class or an interface (99 % )

ArrayList

InputStream

variable name, method name : should start with small letter and follow camelCasing

calculateInterest()

forName()

variable are 2 types:

java does not support global variable ----

it support only instance variable or class variable and local variable

example:

Demo.java:

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

package com.masai;

public class Demo {

//static or class variable

static int x=10;

//non-static or instance variable

int y = 20;

public static void main(String[] args) {

//local variable

int z =100;

System.out.println(x);

}

}

\*\*\*Note: in java, local variable must be initialized before we use it.

//local variable

int z;

z=100;

System.out.println(z);

example :

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

int marks = 500;

// if(marks > 600)

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

// else

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

// String result = "First Div";

//

// if(marks < 600)

// result="Second Div";

//

// System.out.println(result);

String result = (marks > 600) ? "First Div" : "Second Div";

System.out.println(result);

}

}

Day3

int x = 10;

if x value is divisible by 5 then print "Hello" // 10

if x value is divisible by 3 then print "Welcome" // 9

if x value is divisible by 5 and 3 then print "Hello Welcome" // 15,30

if x value is not divisible by 5 and 3 then print "invalid number" //2,11

Datatypes in java:

==============

--java supports 8 primitive datatypes under 4 categories:

A. Integers:

1.byte 8 bit/1 byte 0 -128 to 127

2.short 16bit / 2 byte 0

3.int 32 bit/ 4 byte 0

4.long 64 bit/ 8byte 0

B.Real numbers:

5. float 32 bit or 4 bytes 0.0

6.double 64 bit or 8byte 0.0

C. Charecters:

7.char --- 16 bit or 2 byte ' '

D. Boolean

8. boolean -- 1 bit (true or false) : false

Note: any number with the decimal point value would be accepted by the jvm as double not as a float

\*\*\*local variable must be initialized before we use it.

typecasting:

=========

--the procedure of converting one data type into its equivallent another data type is known as typecasting.

--we have 2 types of typecasting:

1.implicit typecasting: (widening or upcasting)

--storing the smaller datatype value into its equivallent bigger datatype

2.explicit typecasting: (narrowing or downcasting)

--storing the bigger datatype value into its equivallent smaller datatype

byte b =10;

byte c = (byte)(b+1);

Note: if we use arithmatic operator (+, - , \* , /, % ) with two datatypes then the resultent data type will be according to the following rule:

max(int, type1, type2, ....);

int= max(int, byte, byte);

byte to char conversion:

==================

--byte and char both are compatible type, their value can be assignd to each other, implicitly conversion takes places.

char c1 = 'A';

char c2 = 100;

System.out.println(c2);

ex2:

char c1 = 'A';

byte c2 = (byte)c1;

System.out.println(c2);

ex3:

byte b =70;

char c = (char)b;

System.out.println(b);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1) For compute-launch-button-tooltip Error:

https://stackoverflow.com/questions/12529629/eclipse-an-internal-error-occurred-during-compute-launch-button-tooltip

Try setting the launch properties in: Window>Preferences>Run/Debug(Expand)>Launching(Click)

Under launch Operations menu set it to: Always launch the previously launched application Click: Apply>Ok

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

2) For Pushing in Github directly From STS:

https://www.geeksforgeeks.org/how-to-export-eclipse-projects-to-github/#:~:text=Step%25201%253A%2520Open%2520Eclipse%2520IDE,go%2520to%2520Team%252D%253Ecommit.

If still facing issue push it manually for now.

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

3) for jdk error: change version of JDK to 1.7

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class and Object concept:

====================

Java is an Object Orineted Programming language:

OOP supports OOPs concept:

oops concept having some principles:

1.encapsulation :- it is the backbone of OOPs concept.

2.abstraction

3.polymorphism

4.inheritance

Java mostly used to develop Business application:

Business Organization: without computer: manually

Goal : earn profit

---to computerize the services offered by any business organization whatever application we develop is known as Business application.

any program or application common part:

1. data

variables

2.logic

functions/ methods

example:

Banking application:

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

class Account{

private String customerName

private int accno

private double balance = 10000;

public deposit(int amt){

//security

balance = balance+amt;

}

public withdraw(int amt){

//securty logic

balance = balance - amt;

}

checkBalance()

transferAmount()

}

class Authentication{

changePassword()

login()

logout()

}

---untill anything(data) loaded into the RAM (primary memory) it will not be available to the CPU for the execution.

Day4

---untill anything(data) loaded into the RAM (primary memory) it will not be available to the CPU for the execution.

static loading :- a block of code would be loaded into the RAM before it is executed, after being loaded into the RAM it may or may not get executed.

dynamic loading:- a block of code would be loaded into the RAM only when it is required to be executed.

Note: - static loading take place in the execution of structural programing languge, like c program,

where as Java follows dynamic loading.

ex:-

c program:{

1000 variable

10000 methods/function

}

app1.c ---while execution, entire application will be loaded into the RAM

ex2: Java application

class Demo{

1000 variables

10000 methods/function

}

Demo.java -----> compile(java compiler)----->Demo.class---while executing ,

only required part will be loaded into the RAM, remaining part will be there inside

the harddisk in the form bytecode.

--when even it is required , at that time we can load other part of the application into the RAM dynamically and execute them.

--here the required parts are static members.

1. how many members are there in a class:

a. variables

b. methods

--after counting the members inside a class, we need to identify the types of the members

( i.e static or non-static)

--the members where 'static' keyword is applied is known as static member and for which it is not applied is known as non-static.

--while executing a java application, all the static members will be loaded into the RAM and they will be available to the CPU , where as non-static will be there inside the harddisk in the form of byte code.

--to access the non-static members in main method, we need to load the non-static members inside the RAM dynamically by creating object of that class.

requirement of creating an object: -- to load non-static memebers of a class into the RAM dynamically we need to create instance/object of that class.

Ex:

Demo.java:

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

package com.masai;

public class Demo {

//instance variable

int i=100;

//static or class variable

static int j =200;

//non-static

void fun1() {

System.out.println("inside fun1 of Demo");

//10000

}

static void fun2() {

System.out.println("inside fun2 of Demo");

}

public static void main(String[] args) {

//local variable

//int x= 10;

//System.out.println(j);

System.out.println("inside main");

//creating object of a class

//instantiating a class

Demo d1 = new Demo();

System.out.println(d1);

System.out.println(d1.i);

d1.fun1();

//fun1();

//fun2();

}

}

functionality of 'new' operator:

=========================

1. whenever jvm encounters 'new' operator, it will reserve a memeory space inside the heap area of RAM.

2. inside that memory space, it will load all the non-static members of that class

(variable + methods)

\*\*\*while loading the methods, it will load only the method name/signature and its address,

it will not load method body, method body will be loaded at run time when we call that

method.

\*\*\*while loading the non-static variables, inside that reserved memory space, if variable are not initialized then jvm will provide the default value to those non-static variables.

Note: default value will not be given to the local variables, we can not use

uninitialized local variable in our application, it will raise an compilation error.

Note: we can create multiple objects for a perticular class,and modification done in one obj will not reflect another object.

\*\*\*technically, an obj of a class is a memory space inside the heap area where

non-static elements are loaded.

Demo d1 = ?

--for a class ref variable 3 possible values are there:

1. its own class object.

Demo d1 = new Demo();

2. its child class object:-

Demo d1 = new DemoChild(); // it is possible only if DemoChild class is the child class of

Demo class.

--this concept is also known as super class ref and child class object.

3. default value for any ref variable. i.e null.

Demo d1 = null;

example:

Demo.java:

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

package com.masai;

public class Demo {

//instance variable

int i;

//static or class variable

static int j =200;

//non-static

void fun1() {

System.out.println("inside fun1 of Demo");

//10000

}

static void fun2() {

System.out.println("inside fun2 of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();//here d1 is a reference variable, d1 is pointing to the Demo class obj.

d1.i=100;

System.out.println(d1.i);//100

Demo d2 = d1;

System.out.println(d2.i); //100

}

}

Note: one obj(memory space) can be refered by multiple variables simulteniously,

but one variable can not refer multiple object simulteniously.

--the obj which is not referenced by any ref variable will be treated as garbage, and in Java there is a seperate thread running continiusly, called Garbage collector, the duty of this garbage collector is to kill that un-referenced obj and free the RAM.

--if any one variable holding the address of any object then that object is not

treated as garbage.

--from the null, if we try to access any value(non-static) or call any method(non-static), then we will get a runtime exception called NullPointerException.

example:

Demo d1 = new Demo();//here d1 is a reference variable, d1 is pointing to the Demo class obj.

d1.i=100;

Demo d2 = d1;

d1= null;

System.out.println(d2.i);//100

// till this line no any obj is eligible for GC.

primitive variables having thier seperate copies:

ex:

int x = 10;

int y =x;

System.out.println(x); // 10

System.out.println(y); //10

x = 200;

System.out.println(x); // 200

System.out.println(y); // 10

where as object if reassigned to diff variables then all variables will points to the s

same copy.

ex2:

Demo d1 = new Demo();

Demo d2 = d1;

System.out.println(d1.i); //0

System.out.println(d2.i); //0

d1.i=500;

System.out.println(d1.i); //500

System.out.println(d2.i); //500

ex3:

Demo d1 = new Demo();

Demo d2 = d1;

System.out.println(d1.i); //0

System.out.println(d2.i); //0

d2.i=500;

System.out.println(d1.i); //500

System.out.println(d2.i); //500

example :

Demo d1 = new Demo();

d1.fun1();

//another approach of calling a method

new Demo().fun1();

d1.fun1();

d1.fun1();

System.out.println(d1.i);

State of an object: data present inside that object at that instance of time is known(what an obj knows)

as state of the object.

behaviour of an object: functionality that are applicable to that object (what an object

can do) is known as behaviour of an object

Song.java:

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

package com.masai;

public class Song {

String artist;

String title;

void play() {

System.out.println(artist+" is singing "+title);

}

public static void main(String[] args) {

Song track1 = new Song();

track1.artist="Lata";

track1.title="Wande Matram";

track1.play();

Song track2 = new Song();

track2.artist="Sukhwindar";

track2.title="Jai Ho";

track2.play();

}

}

Note: we can have an empty class also. and we can generate the .class file also for this class:

ex:

A.java:

----------

public class A {

}

A.java ---- > A.class

Has-A relationship:

==================

inside a class, as a instance member if we define any other class object then

it is called as a Has-A relationship.

ex:

A.java:

----------

package com.masai;

public class A {

int i = 10;

void funA() {

System.out.println("inside funA of A");

}

}

Demo.java:

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

package com.masai;

public class Demo {

int x = 100;

A a1 = new A();

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1);//address of Demo obj

System.out.println(d1.x);//100

System.out.println(d1.a1);//address of A obj

d1.a1.funA();

}

}

You problem:

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

package com.masai;

public class Demo {

int x = 100;

Demo d1 = new Demo();

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1);

}

}

Day5

Demo.java:

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

package com.masai;

public class Demo {

int x = 100;

Demo d1 = new Demo();

//A a1 = new A();

public static void main(String[] args) {

Demo d1 = new Demo();

}

}

--the above application will throw a runtime exception called StackOverFlowError exception.

Demo.java:

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

package com.masai;

public class Demo {

int x = 100;

Demo d1;

//A a1 = new A();

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1.d1);//null

d1.d1=new Demo();

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

}

}

Accessing static members of a class:

============================

--to access the static members from the main method, we have 3 options:

1. we can access directly without any ref variable (within the same class).

2.By using class name .(dot) // it is most recomended way.

3.By creating an object of that class and through object reference.

example:

Demo.java:

package com.masai;

public class Demo {

static int x = 100;

static void fun1() {

System.out.println("inside fun1 of Demo");

}

public static void main(String[] args) {

// System.out.println(x);

// fun1();

// System.out.println(Demo.x);

// Demo.fun1();

Demo d1 = new Demo();

d1.fun1();

}

}

accessing static member by using class name :-

--because when we use a class name inside our application that class

context will be created.(and static memebers are loaded inside this context area)

accessing static member by using class object:-

example:

package com.masai;

public class Demo {

static int x = 100;

static void fun1() {

System.out.println("inside fun1 of Demo");

}

int y =20;

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1.y);//20

System.out.println(d1.x);//100

d1= null;

System.out.println(d1);

System.out.println(d1.x);// 100

System.out.println(d1.y);// NPE

}

}

Note: each ref variable of a class have 2 parts, one part points to the heap area and another part points to the context area.

--so that we can access the static members with the ref variable of a class also.

--the part which is pointing to the heap area will be under the control of

developer(we can assign another object or null ) and the part which is pointing to the context area, will not be under the control of developer.

example:

Demo.java:

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

package com.masai;

public class Demo {

static int x = 100;

int y =20;

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1.x);//100

System.out.println(d1.y);//20

d1.x=200;

d1.y=500;

Demo d2 = new Demo();

System.out.println(d2.x);//200

System.out.println(d2.y);//20

}

}

Note: non-static members are specific to each object, where as static members

are sharable with all the object of same class, (single copy of the static member will be

created.)

System.out.println:-

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

package com.masai;

public class Demo {

static int x =100;

static A a1 = new A();

public static void main(String[] args) {

System.out.println(Demo.x);

Demo.a1.funA();//A

System.out.println();//PrintStream

}

}

--funA() belongs to A class, similarly println() method belongs to PrintStream

class .

--this PrintStream class is statically defined inside the System class with 'out' variable.

//example of System class..

class System {

static PrintStream out = new PrintStream();

}

Same class object as static member :

============================

package com.masai;

public class Demo {

int x =100;

static Demo d1 = new Demo();

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1.x);

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

System.out.println(Demo.d1.x);

}

}

methods in java:

=============

--Java does not support nested method defination, we can only call one method from another method.

--method is having 2 part :

1.method signature

2.method body

public void fun1() // method signature

{ // method body

//10000

//or Empty body

}

--in java we have two type sof methods:

1.normal or concreate method : method with body

2.abstract method : method without body

method with parameters:

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

example:

package com.masai;

public class Demo {

//method declaration

public void fun1(int x) {

System.out.println("inside fun1 of Demo "+x);

}

public static void main(String[] args) {

Demo d1 = new Demo();

byte x=20;

//int x=30;

//method call

d1.fun1(x);

}

}

Example : method with ref type parameter:

Demo.java:

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

package com.masai;

public class Demo {

//method declaration

public void fun1(A a1) {

System.out.println("inside fun1 of Demo "+a1);

}

public static void main(String[] args) {

Demo d1 = new Demo();

A obj = new A();

d1.fun1(obj); // after the method execution A obj will not be eligible for GC

d1.fun1(new A()); //after the method execution A obj will be eligible for GC

}

}

example 2:

==========

package com.masai;

public class Demo {

//method declaration

public void fun1(A a1) {

if(a1 != null) {

System.out.println("inside fun1 of Demo "+a1);

a1.funA();

}else

System.out.println("please don't pass null value");

}

public static void main(String[] args) {

Demo d1 = new Demo();

A obj = null;

d1.fun1(obj);

}

}

Polymorphism:

==============

--defining more than one functionality(method) with the same name inside a class.

there are 2 types of polymorphism:

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

1.static polymorphism (compile time polymorphism):-- more than one method with the same name but with diff parameter (which method will be executed will be decided at compile time), we achive static polymorphism by using method overloading..

2.dynamic polymorphism (runtime poly...): more than one method with same name and same parameter, dynamic polymorphism we achieve through inheritance (by using method overriding) , which method will be executed will be decided at runtime.

disadv of the static polymorphism:

============================

example 1:

package com.masai;

public class Demo {

void fun1() {

System.out.println("inside fun1 of Demo");

}

void fun1(int x) {

System.out.println("inside fun1(int x) of Demo");

}

void fun1(float f) {

System.out.println("inside fun1(float f) of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

d1.fun1(10.55f);

}

}

example 2:

package com.masai;

public class Demo {

void fun1() {

System.out.println("inside fun1 of Demo");

}

void fun1(byte b) {

System.out.println("inside fun1(byte b) of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

d1.fun1((byte)10);

}

}

--the main drawback of the static polymorphism is, compiler may goes to the

ambigious state , and will generate compilation error.

ex1:

package com.masai;

public class Demo {

void fun1(int x, float y) {

System.out.println("inside fun1(int x,float y) of Demo");

}

void fun1(float x,int y) {

System.out.println("inside fun1(float x, int y) of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

d1.fun1(10,10); // CE

}

}

example 2:

package com.masai;

public class Demo {

void fun1(A a1) {

System.out.println("inside fun1(A a1) of Demo");

}

void fun1(B b1) {

System.out.println("inside fun1(B b1) of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

d1.fun1(null);

}

}

method with return type:

====================

example 1:

package com.masai;

public class Demo {

int fun1() {

System.out.println("inside fun1 of Demo");

byte x=10;

return x;

}

public static void main(String[] args) {

Demo d1 = new Demo();

long l= d1.fun1();

int i= d1.fun1();

byte b= (byte)d1.fun1();

System.out.println(i);

}

}

method with Object as return type:

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

package com.masai;

public class Demo {

A fun1() {

System.out.println("inside fun1 of Demo");

A a1=new A();

return a1;

//return null;

//return new A();

}

public static void main(String[] args) {

Demo d1 = new Demo();

A obj= d1.fun1();

obj.funA();

}

}

example 2:

Demo.java:

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

package com.masai;

public class Demo {

A fun1(String username, String password) {

if(username.equals("admin") && password.equals("123")) {

System.out.println("inside fun1 of Demo");

return new A();

}else

return null;

}

public static void main(String[] args) {

Demo d1 = new Demo();

A obj= d1.fun1("admin","123");

if(obj != null)

obj.funA();

else

System.out.println("invalid username or password");

}

}

Note: -- in java there is a class called Object class, it is a predefined class in

java library which is a super class(parent class) of any classes in java world.

--so if a method is returning an object, then we can hold that obj either

in its own class ref variable or its parent class ref variable

\*\*\*\*\*to the Object class variable we can assign/store any class object.

Demo d1 = new Demo();

//Object obj= d1.fun1("admin","123");

A a1= d1.fun1("admin","123");

Day6

--We can overload the main method as well ,

example:

package com.masai;

public class Demo {

public static void main(int i) {

System.out.println("inside main(int i) of Demo");

}

public static void main(String[] args) {

main(10);

Demo.main(20);

new Demo().main(30);

}

}

--we can call the main method of another class inside another class also.

A.java:

---------

package com.masai;

public class A {

int i = 10;

void funA() {

System.out.println("inside funA of A");

}

public static void main(String[] args) {

System.out.println("inside main of A");

}

}

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

A.main(null);

}

}

Constructor in java:

===============

--it is a kind of non-static method which will be executed automatically at the time of creating an object.

ex:

Demo d1= new Demo();

--the meaning of the above statement is "creating a Demo class obj by calling /executing zero argument constructor of Demo class".

Note: when we compile a .java file of a class, Java compiler verifies, is there any constructor in our .java file or not. if we place any constructor manually inside a .java file, Java compiler will place the same constructor inside the .class file , but if we don't place any constructor explicitly inside our .java file then java compiler will place a default constructor inside the .class file.

ex:

class Demo{

}

java compiler

Demo.java -------------------------> Demo.class

Note: - we can have a .java file for a class without a constructor, but we can't have a .class file for a class without a constructor.

--default constrcutor given by the java compiler will always public and zero argument and it is empty body.

ex: compiler given default constructor

public Demo(){

//super();

}

--strikly speaking , it is not a empty body , there is one hidden statement is there

inside the default constructor as a first statement.

--until the last statement of the constructor is not executed, object is not

created completly. so for an object creatiion, constructor execution is mandatory.

Diff bt normal method and constructor:

===============================

Normal method Constructor

1. method name can be any name. 1. constructor name must be the class name

2. a method must have a return type 2. constructor does not have return type

atleast void

3.method can be static 3. static keyword is not applicable with constructor

4.on a single obj we can call a method multiple 4. on a single obj a constructor will be called only one

time.

5.a method can be abstract and can be final also 5. abstract and final keyword is not applicable with constructor.

Similarities

=========

1. both are the code block, we can write multiple executable statements.

2.as we can overload a method, we can overload a constructor also. and all the static polymorphism rules are applicable with the constructor overloading.

constructor overloading:

====================

ex:

Demo.java:-

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

package com.masai;

public class Demo {

int x = 10;

int y;

void fun1() {

System.out.println("inside fun1 of Demo");

}

Demo(){

System.out.println("inside Demo()....");

}

Demo(int i){

System.out.println("inside Demo(int )....");

System.out.println(i);

}

public static void main(String[] args) {

Demo d1 = new Demo();

Demo d2 = new Demo(10);

//Demo d2 = new Demo("hello");//CE

}

}

--if we place 4 overloaded constructor inside our class, then we can create object of our class in 4 ways.

this keyword:

==========

--it will represent the current class object.

--there are 3 uses of 'this' keyword:

1. to represent the current class object.

2.to defrentiate the instance variable and the local variable

3. to call a constructor of a class from the another constructor of the same class.

Example:

Demo.java:

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

package com.masai;

public class Demo {

int x = 10;

void fun1() {

int x=400;

System.out.println("inside fun1 of Demo");

System.out.println(x);//local variable

System.out.println(this.x); // instance variable

System.out.println(this);//current object on which fun1 is called

}

public static void main(String[] args) {

Demo d1= new Demo();

System.out.println(d1);

d1.fun1();

}

}

\*\*\*Note: 'this' keyword we can not use inside static area.

Demo.java:-

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

package com.masai;

public class Demo {

int x = 10;

public static void main(String[] args) {

int x=200;

System.out.println(this.x);

}

}

Note: constructor will be called automatically whenever we create obj of a cass, but

we can also call a constructor explicitly.

--if we want to call a constructor, then that call must be from the another

constructor of the same class (by using 'this' keyword) or from the

constructor of child class (by using 'super' keyword)

--that call of the constructor must be the first statement inside a constructor.

example:

Demo.java:-

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

package com.masai;

public class Demo {

Demo(){

this(10);

System.out.println("inside Demo()....");

}

Demo(int i){

this("hello");

System.out.println("inside Demo(int )....");

System.out.println(i);

}

Demo(String s){

System.out.println("inside Demo(String )....");

System.out.println(s);

}

public static void main(String[] args) {

Demo d1=new Demo();

}

}

Constructor are used for basically for 2 purpose :

1. if we want to execute some statement at the time of our object creation, then

we can keep those statements inside the constructor.

2. to initialize the instance variable (initialize an object).

example:

Student.java:

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

package com.masai;

public class Student {

int roll;

String name;

int marks;

//zero argument constructor

Student(){

}

//parameterized constructor

Student(int roll, String name, int marks){

this.roll=roll;

this.name=name;

this.marks=marks;

}

public void showDetails() {

System.out.println("Roll is :"+roll);

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

System.out.println("Marks is :"+marks);

}

}

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

Student s1= new Student(10,"Ram",780);

Student s2=new Student();

s2.roll=100;

s2.name="Ravi";

s2.marks=900;

s1.showDetails();

s2.showDetails();

}

}

pure encapsulation:

================

--mark our class variable as private and expose them outside the class

through the public getters and setters method.

--don't expose our data directly.

Java Bean class:

============

--it is a reusable, universal component which should have following properties:

1.this class should be public

2. variables/fields should be private

3.for each variable/field there should be corresponding public getter and

setter methods.

4.this class must have zero argument constructor/default constructor

5.this class may have parameterized constructor. (optional)

Student.java: as a Bean class

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

package com.masai;

public class Student {

private int roll;

private String name;

private int marks;

public Student() {

}

public Student(int roll, String name, int marks) {

super();

this.roll = roll;

this.name = name;

this.marks = marks;

}

public int getRoll() {

return roll;

}

public void setRoll(int roll) {

this.roll = roll;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getMarks() {

return marks;

}

public void setMarks(int marks) {

this.marks = marks;

}

}

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

Student s1= new Student(10, "Ram", 780);

Student s2= new Student();

s2.setRoll(20);

s2.setName("Ramesh");

s2.setMarks(780);

System.out.println("Roll is :"+s1.getRoll());

System.out.println("Name is :"+s1.getName());

System.out.println("Marks is :"+s1.getMarks());

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

System.out.println("Roll is :"+s2.getRoll());

System.out.println("Name is :"+s2.getName());

System.out.println("Marks is :"+s2.getMarks());

}

}

Day7

Taking input from the user in java:

==============================

--In java(desktop application) we can take input from the user inside our application in 3 ways:

1. Scanner class (java 1.5 version )

2. using Command Line argument

3.using BufferedReader class

Note: with the help of CLA, and BR class we can take input only in the form of String.

--using Scanner class we can take the input in almost all the primitive datatype also.

--This Scanner class belongs to java.util package.

in java common classes (String, Object, System, etc....) they belongs to java.lang package.

System

fully qualified name :

java.lang.System

java.util.Scanner

com.masai.Student

--if we want to utilize/use any classes which belongs to other than java.lang package , inside our class, then we need to import that class inside our class.

//creating Scanner class object by providing the keyboard address as a source.

Scanner sc = new Scanner(System.in); // System.in represent the address of the keyboard.

--this Scanner class having multiple methods by using which we can read the data in our application in various primitive data types:

nextInt();

nextFloat();

nextLong();

nextBoolean();

next(); //read the String

--------------------------------------> these methods will read the next token in their respected data type, they will not read the entire line.

nextLine() // read the String // it will read the entire line.

example:

Demo.java:

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

package com.masai;

import java.util.Scanner;

public class Demo {

public static void main(String[] args) {

Scanner sc= new Scanner(System.in);

System.out.println("Enter a number :");

int num= sc.nextInt();

System.out.println("That number is :"+num);

}

}

ex2:

Scanner sc= new Scanner(System.in);

System.out.println("Enter a roll :");

int roll= sc.nextInt();

System.out.println("Enter marks :");

int marks= sc.nextInt();

System.out.println("The Roll is :"+roll);

System.out.println("The Marks is :"+marks);

next() and nextLine():-

==================

Demo.java:

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

Scanner sc= new Scanner(System.in);

System.out.println("Enter a roll :");

int roll= sc.nextInt();

System.out.println("Enter Name :");

String name= sc.next();

System.out.println("Enter marks :");

int marks= sc.nextInt();

System.out.println("The Roll is :"+roll);

System.out.println("The Name is :"+name);

System.out.println("The Marks is :"+marks);

explanation: next();

Enter roll

10 "/n"

Enter name:

Ram Kumar singh "/n" // here kumar is consumed as next token as marks value

Enter Marks

20 "/n"

explanation: nextLine();

Enter roll

10 "/n"

Enter name: // here sc.nextLine() will consume the above "/n"

Ram Kumar singh "/n"

Enter Marks

20 "/n"

---so to solve the above problem we have following solutions:

1. never use nextLine() always read the token by using next();

//with this we can not read full space sepetated name.

2. always use nextLine(); //even to read the primitives also.

ex:

Scanner sc= new Scanner(System.in);

System.out.println("Enter a roll :");

int roll= Integer.parseInt(sc.nextLine());

System.out.println("Enter Name :");

String name= sc.nextLine();

System.out.println("Enter marks :");

int marks= Integer.parseInt(sc.nextLine());

System.out.println("The Roll is :"+roll);

System.out.println("The Name is :"+name);

System.out.println("The Marks is :"+marks);

3. if we need to use nextLine() , then use it at begining.

ex:

Scanner sc= new Scanner(System.in);

System.out.println("Enter Name :");

String name= sc.nextLine();

System.out.println("Enter a roll :");

int roll= sc.nextInt();

System.out.println("Enter marks :");

int marks= sc.nextInt();

System.out.println("The Roll is :"+roll);

System.out.println("The Name is :"+name);

System.out.println("The Marks is :"+marks);

4. call the blank nextLine() after reading the token to consume

the buffered charecter "/n".

Scanner sc= new Scanner(System.in);

System.out.println("Enter a roll :");

int roll= sc.nextInt();

sc.nextLine();

System.out.println("Enter Name :");

String name= sc.nextLine();

System.out.println("Enter marks :");

int marks= sc.nextInt();

System.out.println("The Roll is :"+roll);

System.out.println("The Name is :"+name);

System.out.println("The Marks is :"+marks);

reading the data in the form of matrix:

Scanner sc= new Scanner(System.in);

System.out.println("Enter number in matrix");

int n1= sc.nextInt();

int n2= sc.nextInt();

int n3= sc.nextInt();

sc.nextLine();

int n4= sc.nextInt();

int n5= sc.nextInt();

int n6= sc.nextInt();

System.out.println(n1+" "+n2+" "+n3);

System.out.println(n4+" "+n5+" "+n6);

String class:

==========

--this class belongs to java.lang package.

"java.lang.String"

--in java String is a group of charecter.

String class object we can create in 2 ways:

String s1 = new String("Welcome");

String s2 = "Welcome";

ex:

Demo.java:

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

package com.masai;

public class Demo {

Demo(String s){

}

public static void main(String[] args) {

String s1 = new String("Welcome");

Demo d1 = new Demo("hello");

System.out.println(d1);// Demo@34232

System.out.println(s1);// Welcome

}

}

--there are multiple overloaded println() method is defined inside the PrintStream class :

1.println(){

//printing line break

}

2. println(primitives){

print the primitive

}

3.println(String s){

// it will print the content

}

4.println(Object obj){

//it will print the address

}

5.println(char[] chr)

ex:

String s1 = new String("Welcome");

String s2 = "Welcome";

String s3 = "Welcome";

System.out.println(s1.equals(s2)); // true // compare the content

System.out.println(s1 == s2); // false //compare the ref

diff bt

String s1="Welcome"; // here one object is created

String s2 = new String("Welcome"); // here 2 object is created.

example:

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

String s1 = "Welcome";

String s2 = "welcome";

String s3 = new String("Welcome");

String s4 = new String("Hello");

}

}

reffer String\_scp diagram..

Note: Garbage collector does not have any effect on SCP area,

--SCP area will be sharable in entire application, it is one per JVM.

example:

Demo.java:-

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

package com.masai;

public class Demo {

public static void main(String[] args) {

Demo d1 = new Demo();

d1=null;

Demo d2 = new Demo();

String s1 = "Welcome";

System.out.println(System.identityHashCode(s1));

s1= null;

String s2 = "Welcome";

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

}

}

example2:

A.java:

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

package com.masai;

public class A {

String msg="Welcome";

}

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

String s1="Welcome";

A a1= new A();

System.out.println(s1 == a1.msg);

}

}

Note: String obj is an immutable object, i.e once a string object is created,we can not modify that obj, if we want to modify it by calling its method, those method will return a new String obj. instead of modifying that exixting string object.

example:

String s1="Welcome";

String s2= s1.concat(" to Java");

System.out.println(s1); //Welcome

System.out.println(s2); //Welcome to Java

Since String obj is immutable , we can not modify an existing obj, each modification

will return a new obj, but if we want to get mutability, then we should use

StringBuffer or StringBuilder classes.

--these both classes belongs to java.lang package.

Diff bt StringBuffer and StringBuilder:

================================

--most of the methods of StringBuilder is non-synchronized, i.e not thread-safe

it will give fast performance.

--whereas most of the methods of StringBuffer is synchronized i.e thread safe, and give slow performance compare to StringBuilder.

synchronized void bookTicket(){

--

--

}

synchronized void getAvailibility(){

--

--

}

example:

package com.masai;

public class Demo {

public static String reverseString(String orignalString) {

StringBuilder sb = new StringBuilder("");

char[] chr= orignalString.toCharArray();

for(int i= chr.length-1; i>=0; i--) {

sb.append(chr[i]);

}

return sb.toString();

}

public static void main(String[] args) {

System.out.println(reverseString("welcome"));

}

}

Day8

Arrays in java:--

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

----to hold multiple data of the same type.

--In Java arrays are treated as a special type of object whose class is non-existance.

int[] arr= new int[5];

//5 int variable is created with thier default value as an array obj.

System.out.println(arr);// [I@42343243

Note: array variable is a reference variable which will hold object of an array.

int[] arr= new int[5];

System.out.println(arr);// [I@42343243

System.out.println(arr[0]);//0

System.out.println(arr[3]);//0

System.out.println(arr.length);

System.out.println(arr[6]);//AIOBE

example2:

//declaring an integer array

int[] arr= new int[5];

//reassinging value to the each element of an array

arr[0] =10;

arr[1] =20;

arr[4] =40;

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

//

// System.out.println(arr[i]);

// }

for(int i:arr) {

System.out.println(i);

}

another syntax:

===============

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

for(int i:arr) {

System.out.println(i);

}

difference:

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

int[] arr1;

arr1=new int[5]; //valid

int[] arr2;

arr2= {10,20,30,40,50}; //invalid

example2:

int[] arr= new int[3];

// in this single statement we have created total 4 variables

1. arr --ref variable

2.arr[0] -- primitive variable int type

3.arr[1] -- primitive variable int type

4.arr[2] -- primitive variable int type

Array of object:

============

A[] arr = new A[5];

//A a1, a2, a3;

1.arr --ref variable of array type object

2.arr[0] -- ref variable A type with default value = null

3.arr[1] -- ref variable A type with default value = null

4.arr[2] -- ref variable A type with default value = null

example1:

// A[] arr = new A[3];

//

// arr[0] = new A();

// arr[1] = new A();

// arr[2] = new A();

//

// for(A a1:arr) {

// a1.funA();

// }

//

//shortcut approach

A[] arr = {new A(),new A(), new A(),new A()};

for(A a1:arr) {

a1.funA();

}

exmaple2:

String[] cities= {"Delhi","pune","mumbai","chennai"};

for(String city: cities) {

System.out.println(city.toUpperCase());

}

array as a method parameter:

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

ex1:

package com.masai;

public class Demo {

void fun1(int[] arr) {

System.out.println("inside fun1 of Demo");

}

public static void main(String[] args) {

Demo d1= new Demo();

//here we can pass 2 possible value

//1. null default value for any ref variable

//2. int array object, (loaded or without loaded)

d1.fun1(10);

}

}

example2:

-----------

package com.masai;

public class Demo {

void fun1(int[] arr) {

if(arr != null) {

System.out.println("inside fun1 of Demo");

for(int i:arr) {

System.out.println(i);

}

}

else

System.out.println("null is not allowed");

}

public static void main(String[] args) {

Demo d1= new Demo();

//int[] arr5= new int[3];

int[] arr5= {20,30,40,50};

d1.fun1(arr5);

//d1.fun1(null);

}

}

example3:

package com.masai;

public class Demo {

void fun1(int[] marks) {

if(marks != null) {

System.out.println("inside fun1() of Demo");

int total =0;

for(int m:marks) {

total += m;

}

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

}

else

System.out.println("null is not allowed");

}

public static void main(String[] args) {

Demo d1= new Demo();

int[] nums= {100,120,150,180,200};

d1.fun1(nums);

}

}

Exampl4:

========

package com.masai;

import java.util.Scanner;

public class Demo {

void fun1(Student[] students) {

if(students != null) {

System.out.println("inside fun1() of Demo");

for(Student student:students) {

System.out.println("Roll is :"+student.getRoll());

System.out.println("Name is :"+student.getName());

System.out.println("Marks is :"+student.getMarks());

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

}

}else

System.out.println("null not allowed...");

}

public static void main(String[] args) {

Demo d1= new Demo();

Scanner sc= new Scanner(System.in);

System.out.println("How many Student ?");

int num= sc.nextInt();

Student[] students = new Student[num];//3

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

System.out.println("Enter Details of Student "+(i+1));

System.out.println("Enter roll:");

int roll= sc.nextInt();

System.out.println("Enter Name");

String name= sc.next();

System.out.println("Enter Marks");

int marks= sc.nextInt();

//students[i]= new Student(roll, name, marks);

students[i] = new Student();

students[i].setRoll(roll);

students[i].setName(name);

students[i].setMarks(marks);

}

d1.fun1(students);

}

}

array as return type:

===============

package com.masai;

public class Demo {

Student[] getStudents(String username, String password){

if(username.equals("admin") && password.equals("123")) {

//we can connect with DB and get multiple Student object from the DB and return them

//in the form an array.

//Student array is created with size 3

Student[] students= new Student[3];

//each student variable is initialized with Student class object

students[0]=new Student(10, "N1", 780);

students[1]=new Student(12, "N2", 680);

students[2]=new Student(14, "N3", 880);

return students;

}

else

return null;

}

public static void main(String[] args) {

Demo d1 = new Demo();

//Object obj= d1.getStudents("admin", "123");

Student[] students= d1.getStudents("admin", "123");

if(students !=null) {

for(Student student: students) {

System.out.println("Roll is :"+student.getRoll());

System.out.println("Name is :"+student.getName());

System.out.println("Marks is :"+student.getMarks());

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

}

}else

System.out.println("invalid username or password..");

}

}

A[] arr= new A[10]; // class A 10 variable, with null value

//total 11 variable created 1.arr and 10. A class variable

2d Array:

========

--In Java multi-dimentional array is treated as array of array.

example:

int[][] arr= new int[3][2];

System.out.println(arr);//[[I@

System.out.println(arr.length);//3

System.out.println(arr[0]);//[I@3432432

System.out.println(arr[0].length);

System.out.println(arr[0][1]);//0

example :

package com.masai;

public class Demo {

public static void main(String[] args) {

int[][] arr= new int[4][4];

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

for(int j=0;j<arr[i].length;j++) {

arr[i][j]=j;

}

}

for(int[] ar:arr) {

for(int i:ar) {

System.out.print(i);

}

System.out.println();

}

}

}

Command Line Arguments:

=======================

--JVM will call the main method initialy by suppying empty String array object.

--at the time of running our application though the CLA what ever value we will

supply (space seperated), that value will be loaded into that String array object.

--and that argument is called as CLA, which is available to our application

in the form of String array.

ex:

//loaded string array obj

//String[] names= {"Ram","Amit","Ravi"};

//default value of any ref variable

//String[] names= null;

//empty String array

String[] names= {}; // jvm will pass this obj

or

//String[] names= new String[0];

example:

System.out.println(args);//address of String array obj

System.out.println(args.length);//0

System.out.println(args[0]);//AIOBE

Day9

Inheritance:

==========

--Getting the properties of one object of a class to another Object of another class is known as inhetance.

--with the help of inheritance we achieve IS-A realtionship

--the main adv of inheritance is code reusability and run time polymorphism.

Has-A relationship example:

class A {

B b1 = new B();

}

Animal // parent/super/base

sleep() //1000

walk() //5000

eat() //6000

makeNoise()

//child/sub/derived

Dog Cat Tiger Lion ,20

Type of inheritance:

=================

Note: At class level java supports only multilevel inheritance(single, hirarical).where as with the help of interface we can achieve multiple inheritance.

A.java:

---------

package com.masai;

public class A {

int i=10;

void funA() {

System.out.println("inside funA of A");

}

}

Demo.java:

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

package com.masai;

public class Demo extends A{

int x=100;

void funX() {

System.out.println("inside funX of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1.x);

d1.funX();

System.out.println(d1.i);

d1.funA();

}

}

Here A class will become parent/super class and Demo class will become child clas

or subclass.

--the the object of child class we can reffer the properties of parent class also.

With respect to the diagram:

1. Object of the super class is created first completly before object of the child class created completly.

2.super class obj will be created from the constructor of the subclass.(by using super(); )

Demo(){

super(); // create the parent class object by executing parent class constructor.

}

3. super class object will be created in association with the subclass object.

4. Object class is the super most class of any java class.

Note: meaning of super() is to create the parent class object by executing parent class zero arguement constructor.

Note: if inside the parent class we have only parameterized constructor then inside the child class we need to call that parent class constructor explicitly by supplying appropriate argument ex: super(10);

otherwise our inheritance will fail.. here we can not depend upon default constructor of the child class, we need to keep constructor inside the child class explicitly and from there need to call parent class constrcutor,

example:

A.java:

---------

package com.masai;

public class A {

int i=10;

A(int i){

System.out.println("inside A constructor");

}

void funA() {

System.out.println("inside funA of A");

}

}

Demo.java:

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

package com.masai;

public class Demo extends A{

int x=100;

Demo(){

super(10);

System.out.println("inside Demo constructor");

}

void funX() {

System.out.println("inside funX of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1.x);

d1.funX();

System.out.println(d1.i);

d1.funA();

d1.toString();

}

}

Method overriding:--

================

--it is the procedure of defining a method in the sub-class with the same name and same signature which is already defined inside the parent class/super class.

A.java:

---------

package com.masai;

public class A {

int i=10;

void funA() {

System.out.println("inside funA of A");

}

}

Demo.java:

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

package com.masai;

public class Demo extends A{

int x=100;

void funX() {

System.out.println("inside funX of Demo");

}

@Override //this annotation make sure that we have overide correct method.

void funA() {

System.out.println("inside funA of Demo");

System.out.println("This is second statement inside funA of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

System.out.println(d1.x);

d1.funX();

System.out.println(d1.i);

d1.funA(); // overriden method will gets the priority

}

}

method overriding rules:

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

1. while overriding a method inside the child class we can not reduce its accessebility.

but we can increase.

2. related with the execption handling.

super keyword:

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

--super keyword will represent the immediate parent class object.

--this keyword represents the current class object.

--there are 2 uses of the super keyword:

1.to represent the immediate parent class object.

2.to call the parent class constructor.

Note: by super keyword calling the method of parent class and calling the

constructor of the parent class is entirly different.

--constructor call must be the first statement while method call can be any place.

\*\*\*\*Note: we can not use super and this keyword inside the static area.

example:

A.java:

-----------

package com.masai;

public class A {

int i=10;

void funA() {

System.out.println("inside funA of A");

}

}

Demo.java:

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

package com.masai;

public class Demo extends A{

int x=100;

@Override

void funA() {

System.out.println("inside funA of Demo");

System.out.println("This is second statement inside funA of Demo");

}

void funX() {

System.out.println("inside funX of Demo");

funA(); //overriden method will be called

super.funA(); //parent class method will be called

}

public static void main(String[] args) {

Demo d1 = new Demo();

d1.funX();

//super.d1.funA() // ERROR

}

}

\*\*\*\*\*Note: static members will not participate in inheritance.even though through the child class object we can access the static members of the parent class. these members will be accessed from the context area. not from the object.

--if we define same static method inside the child class which is already defined

statically inside the parent class, then this concept is known as method

hinding, it is not a method overriding, it will hide the parent class static

method inside the context area.

example:

A.java:

----------

package com.masai;

public class A {

int i=10;

static void funS() {

System.out.println("inside funS of A");

}

void funA() {

System.out.println("inside funA of A");

}

}

Demo.java:

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

package com.masai;

public class Demo extends A{

int x=100;

@Override

void funA() {

System.out.println("inside funA of Demo");

System.out.println("This is second statement inside funA of Demo");

}

void funX() {

System.out.println("inside funX of Demo");

funA(); //overriden method will be called

super.funA(); //parent class method will be called

}

static void funS() {

System.out.println("inside funS of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

d1.funX();

d1.funS();// we can access the static with the child class object also.

A.funS();

Demo.funS();

}

}

Super class reference and child class object:

===================================

--according to this rule, we can assign any child class object to its parent class

reference variable.

A a1 = new A();

A a1 = null;

A a1 = new Demo(); // this is possible only if Demo class is the child class of A

A.java:

funA()

funB()

Demo.java extends A

funA(); //overrident

funX(); // child specific method

Note: if a super class ref pointing to the child class object, then with that obj we can refer only members of the parent class and, if that method is overriden inside the child class then child class method will gets the priority.

but with the super class reference we can not call the child class specific functionality. will get a CE.

example:

A.java:

-----------

package com.masai;

public class A {

int i=10;

void funA() {

System.out.println("inside funA of A");

}

void funB() {

System.out.println("inside funB of A");

}

}

Demo.java:

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

package com.masai;

public class Demo extends A{

int x=100;

@Override

void funA() {

System.out.println("inside funA of Demo");

}

void funX() {

System.out.println("inside funX of Demo");

}

public static void main(String[] args) {

// Demo d1 = new Demo();

//

// d1.funB();

// d1.funA();//Demo

// d1.funX();

//A a1 = new A();

//a1.funA(); // A

//a1.funX(); //CE

A a1 = new Demo();

a1.funA();

a1.funB();

//a1.funX(); //CE

}

}

Note: in order to call the child class specific functionality from the parent class object, we need to downcast the super class ref to the appropriate child class object.

ex:

//upcasting

A a1= new Demo();

//downcasting

Demo d1= (Demo)a1;

d1.funX();

//object downcasting this is possible only when super class ref points to the child class object

//if A a1 = new A() and then we try to downcast then we will get a runtime exception called ClassCastException

Day10

Inhetritance:

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

extends

IS-A relationship

A extends B

method overriding

--constructor : Object class

Object class belongs to java.lang

Super class ref and child class obj.

--to a parent class variable we can assign its any child class object.

A a1 = new A();

A a1 = new B(); // when B is a child of A class.

A a1= null;

LgOldTv.java:

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

package com.masai;

public class LgOldTv {

public void start() {

System.out.println("Tv Starting");

}

public void increaseVolume() {

System.out.println("volume increased..");

}

public void changeChannel() {

System.out.println("Channel changed in Old Way...");

}

}

LgSmartTv.java:

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

package com.masai;

public class LgSmartTv extends LgOldTv {

@Override

public void changeChannel() {

System.out.println("Channel Changed in smart way..");

}

public void playGame() {

System.out.println("Game Starts...");

}

}

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

// LgOldTv remote = new LgOldTv();

//

// remote.start();

// remote.increaseVolume();

// remote.changeChannel(); // old way

//

// LgSmartTv remote=new LgSmartTv();

//

// remote.start();

// remote.increaseVolume();

// remote.changeChannel();//smart way

// remote.playGame();

//upcasting

LgOldTv remote = new LgSmartTv();

remote.start();

remote.increaseVolume();

remote.changeChannel();// smart way

//LgSmartTv nRemote=new LgSmartTv();

//object downcasting

LgSmartTv sRemote= (LgSmartTv)remote;

sRemote.playGame();

}

}

exmaple2:

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

Demo.java:

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

package com.masai;

public class Demo {

public void fun1(Object obj) {

System.out.println("inside fun1 of Demo");

}

public static void main(String[] args) {

Demo d1 = new Demo();

d1.fun1(new A());

d1.fun1(new B());

d1.fun1(null);

}

}

example3:

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

package com.masai;

public class Demo {

public Object fun1(int x) {

System.out.println("inside fun1 of Demo");

if(x > 10)

return new A();

else

return new B();

}

public static void main(String[] args) {

Demo d1 = new Demo();

//A a1= d1.fun1(20); //CE

Object obj= d1.fun1(20);

A a1= (A)obj;

a1.funA();

}

}

example4:

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

package com.masai;

public class Demo {

public Object fun1(int x) {

System.out.println("inside fun1 of Demo");

if(x > 10)

return new A();

else

return new B();

}

public static void main(String[] args) {

Demo d1 = new Demo();

Object obj= d1.fun1(50);

if(obj instanceof A) {

A a1= (A)obj;

a1.funA();

}else {

B b1= (B)obj;

b1.funB();

}

}

}

example:

//Object obj= d1.fun1(50);

//A a1= (A)obj;

A a1= (A)d1.fun1(10);

toString() method:

================

--this method belongs to Object class.

Note: Object class methods are also called as universal method, we can call

these methods on any class objects.

public String toString();

--if we call this method on any object then it will convert that object

address into the String and return that String.

--the above functionality is written inside the toString() method of the

Object class.

--toString() method internally called from the println() method.

--println--- PrintStream

println(primitives){

//print the content

}

println(){

//extra line

}

println(String s){

print the content of String

}

println(Object obj){

String s= obj.toString();

println(s);

}

\*\*Note: if we print any class object reference then it should print the address of that object

in the form of String , but if it is not printing the address, and it prints somting else

(some message, some content) then, the meaning is, that class has overriden the toString() method from the object class.

--toString() is called by the concept of dynamic polymorphism.

--which method will be called decided at runtime. it is also known as

late binding.

early binding vs late binding:

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

binding : connecting the method body with method call is known as binding.

--if it is decided at compile time then it is known as early binding(method overloading

or static polymorphism).

--if it is decided by the jvm at runtime then it is known as late binding (method overriding or dynamic polymorphism).

Student.java:

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

package com.masai;

public class Student {

private int roll;

private String name;

private int marks;

public Student() {

}

public Student(int roll, String name, int marks) {

super();

this.roll = roll;

this.name = name;

this.marks = marks;

}

public int getRoll() {

return roll;

}

public void setRoll(int roll) {

this.roll = roll;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getMarks() {

return marks;

}

public void setMarks(int marks) {

this.marks = marks;

}

@Override

public String toString() {

return "Student [roll=" + roll + ", name=" + name + ", marks=" + marks + "]";

}

}

final keyword:

===========

--in java every keyword has some specific functionality, but there is a keyword whose functionality differs depending upon the situation where it is used i.e final keyword.

1. if we define a variable as final, then we can not change its value elsewhere in program.

Note: if we define any final variable as a instance variable then it should be initialized at the same place or from the contructor.

ex:

Demo.java:

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

public class Demo {

final int x=10;

final int y;

Demo(int y){

this.y=y;

}

public static void main(String[] args) {

Demo d1= new Demo(20);

}

}

2. if we define a method as final then we can not override that method inside the child class.

3.if we define a class as final then we can not extends/inherit that class.

String class is a final class.

Access modifiers:

==============

--it specifies the accessebility of a class or its members outside that class or package.

we have 4 types of access modifiers:

1.public : accessebiltiy is global (other package or other classes also)

2.default: it is similar to public as long as inside the same package, but outside the package it works as private

3.protected: it is simillar to default, but outside the package we can access it by using inheritance.

4.private : the accessebility is restricted to that class only.

Note: A normal class/outer-class can only be either default or public but that class members (variables, methods, constructors, inner classes) can be private ,public, default, protected.

Note: we can not apply access modifiers to the local variables.

--default constructor given by java compiler is always public.

--if we make constructor of a class a default then we can not create its object or extends this class outside that package.

--if we make constructor of a class a private then we can not create its object or extends this class outide that class even in the same package also.

example:

A.java:

---------

package com.masai;

public class A {

public static A getAObject() {

A a1= new A();

return a1;

}

private A() {

}

void funA() {

System.out.println("inside funA of A");

}

}

Demo.java:

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

package com.masai;

import java.util.Calendar;

public class Demo {

public static void main(String[] args) {

//static factory method

A a1= A.getAObject();

a1.funA();

Calendar cal = Calendar.getInstance();

}

}

Day11

packages:-

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

--there are 3 uses of the packages in java:

1.with the package,we can bind the related concept logically.

com.masai.db :

100: --- with intract with the DB

com.masai.service

200: -- in which we define business logic to process the data

com.masai.presentation

200: -- in which we define the presentation logic.

--in java api we have so many predefined classes and interfaces are there.

they are also categoriesed in diff packages.

java.lang : it contains all the common classes(ex: String, System, Object, etc)

java.io package : contains all the classes and interface which are required to perform i/o operations

java.sql package

java.util package :

java.net package:

2. with the help of package we can avoid the naming conflict.

String

java.lang.String --- fully qualified name

com.masai.String ---

3. to provide some level of security, by using access modifiers.

default access modifier : outside the package we can not access

protected access modifier: outside the package we can access using inheritance.

--Note: in realtime we should not define a class, interface without a package.

--In Java every package is a folder, where as every folder is not a package.

--when we keep any classes inside any package, then we can compile that class by following

2 ways:

1. manually: here first we need to compile the class and then we need to create folder with the

name of the package and keep all the .class files inside that folder. after that we can

run our application by using fully qualified name.

>java p1.Demo

2.automatically: here we need to compile our class by using following command:

>javac -d . Demo.java :- this command wll create a folder with the name of packages

automatically.

Abstract class:

===========

abstract class

abstract method

--by using an Abstract class we acheive partial abstraction in java.

abstract Animal

eat()

sleep()

walk()

makeNoise()

DomesticAnimal WildAnimal

Dog Cat Tiger Lion

Dog tommy = new Dog();

Animal tommy = new Dog();

Animal tommy = new Animal(); //CE

--In java, if we want to restrict the user, so that he can not instantiate or create object of a class directly then marks that class as abstract class.

--if we make constructor as private then we can not create its object outside that class

but inside the class we can create its object

Note: an abstract class does not have any meaning, unless it is extended by the child class, and we are allowed to create only child class object.

there are 3 differences between a normal class and an abstract class:

====================================================

1. for an abstract class we can not create its object directly,(new keyword is not applicable)

2.inside an abstract class, we \*\*may have an abstract method also.(it is not mandatory)

Note: we can have an empty abstract class also.

3. final keyword is not applicable with the abstract class (final and abstract both are enimies)

Note: apart from above 3 diff bt normal and abstract class, we can do all the

things with the abstract class whatever we can do with the normal class.

ex:

we can have variables

we can have constructors.

etc..

abstract method:

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

--method with body is also knwon as implemented method or concreate method

--method without body is known as abstract method or unimplemented method. these types of methos should have abstract keyword.

ex:

public abstract void makeNoise();

Note: - inside a normal class/concreate class, we can not have an abstract method.

--if we want to place an abstract method inside our class then we need to mark

that class also as an abstract class.

ex:

A.java:

-----------

package com.masai;

public abstract class A {

//unimplemented method or abstract method

public abstract void funAbs();

//implemented method

void funA() {

System.out.println("inside funA of A");

}

}

rule:

\*\*\*Note: if an abstract class having any abstract method then we must override that method inside the child class otherwise we need to mark that child class also as an abstract class.

A.java:

-----------

package com.masai;

public abstract class A{

//unimplemented method or abstract method

public abstract void funAbs1();

//unimplemented method or abstract method

public abstract void funAbs2();

//unimplemented method or abstract method

public abstract void funAbs3();

//implemented method, inside child class we can not override this

final void funA() {

System.out.println("inside funA of A");

}

}

AChild.java:

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

package com.masai;

public abstract class AChild extends A{

@Override

public void funAbs1() {

// TODO Auto-generated method stub

}

@Override

public void funAbs2() {

// TODO Auto-generated method stub

}

}

AGrandChild.java:

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

package com.masai;

public class AGrandChild extends AChild{

@Override

public void funAbs3() {

// TODO Auto-generated method stub

}

}

Normal class

A a1 = ? // 3 possible values

A a1 = new A(); // same class obj

A a1 = new AChild();// child class obj

A a1=null;

Abstract class:

Abs a1 =? // 2 possible values

Abs a1 = new Abs();// CE

Abs a1 = new AbsChild();

Abs a1 = null;

Abstract object created along with its child class object.

example:

A.java:

-----------

package com.masai;

public abstract class A{

public A() {

System.out.println("inside constructor of A ");

}

//unimplemented method or abstract method

public abstract void funAbs1();

//implemented method

void funA() {

System.out.println("inside funA of A");

}

}

AChild.java:

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

package com.masai;

public class AChild extends A{

public AChild() {

System.out.println("inside the constructor of AChild");

}

@Override

public void funAbs1() {

System.out.println("inside funAbs() of AChild");

}

}

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

A a1 = new AChild();

a1.funA();

a1.funAbs1();

}

}

\*\*\*Note: an abstract method can not be static.

In Java we have 3 types of valid structure:

1. full 100% implemented structure (concrete class) // implemented means, method with body

2.partial implemented structure (abstract class) // partial abstraction

3.full 100% unimplemented structure (interface) // 100% abstraction

Day12

Abstract class :

abstract method..

public abstract void makeNoise();

abstract class Animal {

}

---partial abstraction.

Animal a=new Animal(); //CE

Animal a= new Dog();

Animal a= new Cat();

a.makeNoise();

Dog d =new Dog();

@Autowire

Animal a;

Interface:

=======

--it is a full unimplemented structure in java 100 %

--till Java 1.7 interface use to contains only abstract method and final

variable.

--from Jdk 1.8 onwards we can place method with body also inside an interface

(default method or static method).

\*\*\*Note: With the help of an interface we achive loose coupling in Java.

--inside an interface if we place any method wituout body, that method

will be public and abstract wheter we mention it or not.

X.java:

---------

package com.masai;

public interface X {

public abstract void fun1();

void fun2();

}

class A{

}

default constructor

A.java ----------->compile ----------------> A.class

X.java -------------> compiler --------------> X.class

--constructor concept is not applicable with an interface.

--As a class is extended by another class , an inteface need to be implemented by another class.

rule:

--if a class implements an interface , then that class has to override all the abstract method defined inside that interface otherwise we need to mark that implemented class as an abstract class.

example;

X.java:

--------

package com.masai;

public interface X {

void fun1();

void fun2();

}

XImpl.java:

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

package com.masai;

public class XImpl implements X{

@Override

public void fun1() {

System.out.println("inside fun1 of XImpl");

}

@Override

public void fun2() {

System.out.println("inside fun2 of XImpl");

}

//specific method

public void fun3() {

System.out.println("inside fun3 of XImpl");

}

}

Note: we can not create object of an interface. but we can define a reference variable for an interface.

X x1 = new X(); //CE

X x1 = ? // 2 possible value

X x1 = new XImpl(); // any implemented class object

X x1= null;

Note: we can define variable of any 3 valid structure like (concrete class, abstract class or an interface)

but the object should be created only for the concrete class.

X x1= new XImpl(); // here also super class ref and sub class object rule is applicable .

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

X x1= new XImpl();

x1.fun1();

x1.fun2();

//interface ref downcasted to implemented class obj.

XImpl xx= (XImpl)x1;

xx.fun3();

XImpl x2 = new XImpl();

x2.fun1();

x2.fun2();

x2.fun3();

}

}

--inside an interface, in addition to an abstract method, we can have variables also.

--if we define any variable inside an interface , it will be by default "public static final" wheter we mention it or not.

--that variable must be initialized at time of declaration.

--variable defined inside an interface can be accessed by the implemented class object also.

example:

X.java:

---------

package com.masai;

public interface X {

int i=100;

void fun1();

void fun2();

}

XImpl.java:

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

as previous

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

X x1= new XImpl();

x1.fun1();

x1.fun2();

//interface ref downcasted to implemented class obj.

XImpl xx= (XImpl)x1;

xx.fun3();

System.out.println(xx.i);

System.out.println(x1.i);

System.out.println(X.i);

}

}

Interface as a method parameter:

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

--if a method is defined to take an interface, then we can call that method

by supplying any of its implemented class obj or null.

Another.java:

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

package com.masai;

public class Another implements X{

@Override

public void fun1() {

System.out.println("inside fun1 of Another");

}

@Override

public void fun2() {

System.out.println("inside fun2 of Another");

}

}

Demo.java:

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

package com.masai;

public class Demo {

public void funDemo(X x1) {

if(x1 != null) {

System.out.println("inside funDemo of Demo");

x1.fun1();

x1.fun2();

}

else

System.out.println("null is not allowed..");

}

public static void main(String[] args) {

Demo d1= new Demo();

//XImpl obj = new XImpl();

//d1.funDemo(obj);

//d1.funDemo(new XImpl());

//d1.funDemo(new Another());

d1.funDemo(null);

}

}

\*\*\*Note: with the help of an interface also we achieve IS-A relationship.

--implemented class object Is-A type of an interface.

interface as a method return type:

===========================

--if a method having return type as an interface then that method should

return any of the implementation class obj of that interface or null value.

Demo.java:

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

package com.masai;

public class Demo {

public X funDemo() {

System.out.println("inside funDemo of Demo");

return new XImpl();

//return new Another();

//return null;

}

public static void main(String[] args) {

Demo d1= new Demo();

// X x1= d1.funDemo();

//

// if(x1 != null) {

// x1.fun1();

// x1.fun2();

//

// }else

// System.out.println("returning null value");

Object obj= d1.funDemo();

//first level downcasting

X x1= (X)obj;

x1.fun1();

x1.fun2();

//second level of downcasting

XImpl xx= (XImpl)x1;

xx.fun3();

//directly downcasting obj to XImpl object

XImpl xx2= (XImpl)obj;

xx2.fun1();

xx2.fun2();

xx2.fun3();

}

}

Class is a blueprint for an object, where as Interface is also like blueprint of the class.

Hotel.java:

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

package com.masai;

public interface Hotel {

public void chickenBiryani();

public void masalaDosa();

}

TajHotel.java:

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

package com.masai;

public class TajHotel implements Hotel{

@Override

public void chickenBiryani() {

System.out.println("ChickenBiryani from TajHotel");

}

@Override

public void masalaDosa() {

System.out.println("Masala Dosa from TajHotel");

}

//specific method of TajHotel class

public void paneerMasalaDosa() {

System.out.println("paneer masala dosa from Taj Hotel");

}

}

RoadSideHotel.java:

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

package com.masai;

public class RoadSideHotel implements Hotel{

@Override

public void chickenBiryani() {

System.out.println("ChickenBiryani from RoadSide Hotel");

}

@Override

public void masalaDosa() {

System.out.println("ChickenBiryani from RoadSide Hotel");

}

}

Demo.java:

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

package com.masai;

public class Demo {

public Hotel provideFood(int amount) {

Hotel hotel= null;

if(amount > 500)

hotel = new TajHotel();

else if(amount > 200 && amount <=500)

hotel = new RoadSideHotel();

return hotel;

}

public static void main(String[] args) {

Demo d1= new Demo();

Hotel h= d1.provideFood(800);

if(h != null) {

h.chickenBiryani();

h.masalaDosa();

if(h instanceof TajHotel) {

TajHotel taj= (TajHotel)h;

taj.paneerMasalaDosa();

}

}

else

System.out.println("Amount should be greater than 200");

}

}

--interface can not extends another class and can not implement any interface also.

\*\*\*\*--but one interface can extends more than one interface simulteniously(multiple inheritance)

Intr1.java(I)

funA();

funB();

Intr2.java(I)

funC()

funD()

interface Intr3 extends Intr1,Intr2{

funX();

funY();

}

class Demo implements Intr3{

//it has to override all the methods of Intr1, Intr2 and Intr3 interfaces, otherwise

Demo class has to be marked as abstract class.

}

Demo d1 = new Demo();

d1.funA();

d1.funB();

d1.funC();

d1.funD();

d1.funX();

d1.funY();

Intr1 i1 = new Demo();

i1.funA();

i1.funB();

Intr2 i2= new Demo();

i2.funC();

i2.funD();

Intr3 i3 = new Demo();

i3.funA();

i3.funB();

i3.funC();

i3.funD();

i3.funX();

i3.funY();

--from java 1.8 onwards some new feature introduced in interface.

1.default method

2.static method

--both method should have a body.

1.default method:

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

-- we can define a default method with the body inside an interface.

--this default method need not override inside the implementation classes.

--if we want , we can override this default method inside any implementation classes.

--these default method are bydefault inherited inside the implementation classes.

--we can call these default method from any implementation class object.

Hotel.java:

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

package com.masai;

public interface Hotel {

public void chickenBiryani();

public void masalaDosa();

public default void iceCream() {

System.out.println("iceCream from Hotel");

}

}

TajHotel.java:

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

package com.masai;

public class TajHotel implements Hotel{

@Override

public void chickenBiryani() {

System.out.println("ChickenBiryani from TajHotel");

}

@Override

public void masalaDosa() {

System.out.println("Masala Dosa from TajHotel");

}

//specific method of TajHotel class

public void paneerMasalaDosa() {

System.out.println("paneer masala dosa from Taj Hotel");

}

@Override

public void iceCream() {

System.out.println("Ice cream from TajHotel");

}

}

RoadSideHotel.java:

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

from previous example

Demo.java:

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

package com.masai;

public class Demo {

public Hotel provideFood(int amount) {

Hotel hotel= null;

if(amount > 500)

hotel = new TajHotel();

else if(amount > 200 && amount <=500)

hotel = new RoadSideHotel();

return hotel;

}

public static void main(String[] args) {

Demo d1= new Demo();

Hotel h= d1.provideFood(800);

if(h != null) {

h.chickenBiryani();

h.masalaDosa();

h.iceCream();

if(h instanceof TajHotel) {

TajHotel taj= (TajHotel)h;

taj.paneerMasalaDosa();

}

}

else

System.out.println("Amount should be greater than 200");

}

}

2.static method:

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

--we can define a static method also inside an interface from java 1.8

--this static method must have body.

--static method of an inteface will not be inherited inside the implementation class object.

--so we can not call this static method of an interface by using implementation class

object.

\*\*Note: we can call the static method of an interface only by using Interface name.

we can not call static method of an interface even by using interface variable also.

Note:- we can define same static method as static or non-static method inside the implementation class also.

which is already defined statically inside the interface. (this concept is called as method hinding)

Hotel.java:

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

package com.masai;

public interface Hotel {

public void chickenBiryani();

public void masalaDosa();

public default void iceCream() {

System.out.println("iceCream from Hotel");

}

public static void drinkingWater() {

System.out.println("drinking water from Hotel");

}

}

Demo.java:

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

package com.masai;

public class Demo {

public Hotel provideFood(int amount) {

Hotel hotel= null;

if(amount > 500)

hotel = new TajHotel();

else if(amount > 200 && amount <=500)

hotel = new RoadSideHotel();

return hotel;

}

public static void main(String[] args) {

Demo d1= new Demo();

Hotel h= d1.provideFood(800);

if(h != null) {

h.chickenBiryani();

h.masalaDosa();

h.iceCream();

Hotel.drinkingWater();

//h.drinkingWater(); //CE

if(h instanceof TajHotel) {

TajHotel taj= (TajHotel)h;

taj.paneerMasalaDosa();

}

}

else

System.out.println("Amount should be greater than 200");

}

}

Day13

interface StudentService{

public void registerStudent(Student student);

public Student getStudentByRoll(int roll);

public boolean updateStudent(Student student);

}

class StudentServiceImpl1 implements StudentService{

public void registerStudent(Student student){

//using JDBC approach

}

public Student getStudentByRoll(int roll){

//using JDBC approach

}

public boolean updateStudent(Student student);

//using JDBC approach

}

class StudentServiceImpl2 implements StudentService{

public void registerStudent(Student student){

//using ORM approach

}

public Student getStudentByRoll(int roll){

//using ORM approach

}

public boolean updateStudent(Student student);

//using ORM approach

}

@Autowire

StudentService service;

service.registerStudent(student);

service.getStudentByRoll(10);

service.updateStudent(student);

example 2:

interface X {

void funX();

}

interface Y {

void funY();

}

Note: one class can extends another class and implements multiple interfaces simulteniously.

example

class A extends B implements X,Y{

}

Note: we can have an empty interface also.

public interface X{

}

--this type of empty interface is also known as tag or marker interface.

--some of the predefined marker interfaces in Java are:

java.io.Serializable

java.lang.Cloanable

--these kind of marker interfaces used to specify certain type of special behaviour of our object.

class Student implements Serializable{

}

Student s1=new Student(10,"Ram",780); // moving out this Student obj from the RAM to outside the RAM is known as serialization. and reverse is known as deserialization.

--if we want to group multiple objects to a special type then also we can make use Marker interface.

example:

Special.java:-

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

package com.masai;

//marker interface

public interface Special {

}

A.java:

----------

package com.masai;

public class A implements Special{

void funA() {

System.out.println("inside funA of A");

}

}

Demo.java:

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

package com.masai;

public class Demo {

public void fun1(Special special) {

//want to specify some specail behaviour

System.out.println("here special behaviour is applied");

}

public static void main(String[] args) {

Demo d1= new Demo();

d1.fun1(new A());

}

}

Collection framework:

==================

Collection: if we want to represent a group of objects as a single unit (single object)

then we should use collection.

--Collection of objects..

Framework:

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

--the main objective of a f/w is ease developer work.

--it is semimplemented architecture.

--A f/w comprises some abstract design with built-in behaviour(functionality),

in order to use it, we need to insert our functionality in various places of f/w.

--A s/w f/w is a universal, resuable s/w platform to develop s/w applications,

products or solutions.

Data structure:

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

--array

--stack

--queue

--LinkedList

--searching

--sorting

Collection f/w:

============

--it defines several classes and interfaces which can be used to represent/arrange

group of objects as single unit(object)

--each collection classes are like a container or a bag, where we can hold multiple

objects.

--it is basically readymade implemented DS.

---we can group/hold/store multiple object as a single object by using array.

--in Java array is also one type of object

--5 student object as a single object.

Student[] students = new Student[5]; // here one array object is created and 6

variables are created.

students // ref type array variable

student[0] // ref type Student class variable

student[1] // ref type Student class variable

student[2] // ref type Student class variable

student[3] // ref type Student class variable

student[4] // ref type Student class variable

student[0] = new Student(10,"N1",780);

student[1] = new Student(12,"N2",880);

student[2] = new Student(13,"N3",680);

student[3] = new Student(14,"N4",580);

student[4] = new Student(15,"N5",680);

limitations of array:

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

1. size is fixed, we can not increase or decrease it dynamically.

2.it supports homogenious type of Data/element.

--this limitation we can overcome by taking Object class array.

Object[] objs= new Object[3];

objs[0]=new Student();

objs[1]=new Employee();

objs[2]=new Product();

3. array concept is not implemented based on readymade method support.

for each and every requirement we need to write our own logic,.

even for printing the elements from an array we need to write for loop.

--in each array object we have only one non-static variable i.e 'length'.

--to overcome the above limitation we need to use collection f/w.

feature of Collection f/w:

====================

--collections are growable and shinkable in nature.

--collection can hold both homogenoius and heterogenious type of element.

--every collection classes are implemented based on some standard data-structure, hence readymade method

support is available for most of the requirements.

\*\*\*All(99%) the collection f/w related classes and interfaces belongs to java.util package.

--collection f/w having 2 section:

1. normal collection (here we manage object uniformally/individually/ singular manner).

2. Map (here we manage objects in key-value pair)

List: ---> when we need to arrange the elements in a sequence (index based manner) and duplicate elements are allowed.

Set: -- when we need uniqueness (duplicate elements are not allowed)

Queue:-- when we need to arrange the elements prior to processing(FIFO order is default, but we can manipulate)

\*\*\*\*Note:- from Java 1.5 LinkedList class also implements Deque interface.

Collection Interface:

================

--It is the foundation upon which the collection f/w is built.

--it declares some of the core methods that all collection classes will have.

--these methods we can call on any collection f/w related classes objects.

some of methods of collection interface:

================================

int size(); // how many elements are there in that collection object.

boolean isEmpty();

boolean contains(Object obj); //for searching any element.

Iterator iterator(); // it is inherited from Iterable interface

Object[] toArray();//to convert any collection object to normal array

T[] toArray(T[]); //convert only limited size element to array

boolean add(Object obj); // it is a most commonly used method, to add any element insdie any tupe of

collection.

boolean remove(Object obj);

boolean containsAll(Collection col);

boolean addAll(Collection col); copy all the element from one collection to our collection object

boolean removeAll(Collection col);

boolean retainAll(Collection col); // except supplied collection elements all other will be removed.

void clear(); // remove all the elements from the collectioj

// these following 2 methods belongs to Object class

boolean equals(Object obj); //compare two collection object

int hashCode(); // return the hashCode of collection obj

//these methods introduced in java 1.8 to perform functional programming.

public boolean removeIf(java.util.function.Predicate<? super E>);

public java.util.Spliterator<E> spliterator();

public java.util.stream.Stream<E> stream();

public java.util.stream.Stream<E> parallelStream();

Day14

List:

=====

--It is the child interface of Collection interface and declares the behavour of a collection to

preserve the sequence of an element.

--elements can inserted and accessed by thier position using zero based index.

--here insertion order will be preserved and duplicate elements are allowed.

--in addition to the Collection inteface methods ,List interface defines some of its own

methods also.

public Object get(int index);

public Object set(int index, Object newObject); it will return the overriden object

public Object remove(int index); // will return the removed object.

public int indexOf(Object obj);

etc..

\*\*\*Note: Collection f/w does not support primitives, it only supports objects.

add(Object obj);

ArrayList:

========

--it is the implementation of the List interface.

--it is basically a dynamic array (it dynamically increase and decrease in size).

--ArrayList class is the best choice if our frequent operation is retrieval based on index.

--duplicate are allowed.

--null insertion is possible (multiple time also)

example:

ArrayList al=new ArrayList();

System.out.println(al);//[]

Note: all the collection classes has overriden the toString() method internally. so they will print the elements inside that collection in [] square bracket.

--all the collection classes are like a container or bag which holds multiple objects.

--in the above statement we have created an empty AL object with the default initial

capacity 10.

--once AL reaches to its max capacity then a new AL object will be created in the memory automatically with the new capacity using following formula:

newCapacity = (currentCapacity \* 3/2) +1;

ArrayList al=new ArrayList(1000); // AL created with the initial capacity 1000;

Autoboxing and Autounboxing:

==========================

this concept comes in java 1.5 version

boxing: converting primitives into the object(box) it is known as boxing and reverse

is called unboxing.

--8 primitive datatype.

--for each primitive data types we have corresponding wrapper classes are there.

--int --- java.lang.Integer

--byte --- java.lang.Byte

--char ---> java.lang.Charecter

--boolean --> java.lang.Boolean

--

--

--before java 1.5 inorder to add the primitives in the collection we need to wrap that primitives to their corresponding wrapper class object.

int i=10;

Integer i1= Integer.valueOf(i); //boxing

int x= i1.intValue(); //unboxing

from java 1.5 onwords we have a concept called autoboxing and autounboxing

int i =10;

Integer i1 = i; //autoboxing

int x=i1; // autounboxing

example:

ArrayList al=new ArrayList();

al.add("delhi");

al.add("mumbai");

al.add("chennai");

al.add("kolkata");

al.add(new A());

al.add(new Student(10, "Amit", 780));

al.add(null);

al.add(null);

al.add("delhi");

al.add(10); // Integer

al.add(true); //Boolean

al.add(10.55); //Double

System.out.println(al);

System.out.println(al.size());

// Object obj= al.get(1);

// String city= (String)obj;

int x= (Integer)al.get(9);

System.out.println(x);

---in the above application our ArrayList object is not a type safe Collection object.

--if our collection is not type safe collection then we can add any type of object at

any position inside our collection.

--here while getting the elements from the type unsafe collection every time we

need to downcast the element, which is not feasiable. there might be a change of

ClassCastException.

--so in realtime, our collection should be type safe collection.

--type safe collection means making our collection homogenious.

benifit of type safe collection:

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

1. if we try to add any other type of element then compiler will stop at compile time.

2.we will get rid of downcasting problem.

ArrayList<Object> al=new ArrayList<>();

--taking the type of Object is simmillar to creating type-unsafe collection.

example

package com.masai;

import java.util.ArrayList;

public class Demo {

public static void main(String[] args) {

ArrayList<String> al=new ArrayList<>();

al.add("delhi");

al.add("mumbai");

al.add("chennai");

al.add("kolkata");

al.add("delhi");

al.add(10);

al.add(null);

String s= al.get(2);

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

}

}

--the above type safe collection concept is called Genrics concept.

--Generics concept also introduced in java 1.5 version

//ArrayList class sudo code before generics

class ArrayList implements List{

public boolean add(Object obj){

//adding the object obj to the AL.

}

public Object get(int index){

it will return the obj to the specified index

}

//remaining methods.

}

//ArrayList class sudo code after generics

class ArrayList<T> implements List{

public boolean add(T t){

//adding the object t to the AL.

}

public T get(int index){

it will return the obj to the specified index

}

//remaining methods.

}

example:

ArrayList<String> al=new ArrayList<>(); // it is the List of String object

ArrayList<Integer> al=new ArrayList<>(); // it is the List of Integer object

ArrayList<Student> al=new ArrayList<>(); // it is the List of Student object

Demo.java:

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

package com.masai;

import java.util.ArrayList;

import java.util.Scanner;

public class Demo {

public static void main(String[] args) {

Scanner sc= new Scanner(System.in);

ArrayList<Student> students = new ArrayList<>();

int count = 1;

while(true) {

System.out.println("Enter details of Student "+(count++));

System.out.println("Enter Roll");

int roll= sc.nextInt();

System.out.println("Enter Name");

String name= sc.next();

System.out.println("Enter Marks");

int marks= sc.nextInt();

Student student = new Student(roll, name, marks);

students.add(student);

System.out.println("Student object added sucessfully...");

System.out.println("Want more(y/n) ?");

String choice= sc.next();

if(choice.equalsIgnoreCase("n"))

break;

}

for(Student student: students) {

System.out.println("Roll is :"+student.getRoll());

System.out.println("Name is :"+student.getName());

System.out.println("Marks is :"+student.getMarks());

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

}

}

}

Day15

using normal for loop to iterate the AL object:

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

for(int i=0;i< students.size();i++) {

Student student= students.get(i);

System.out.println("Roll is :"+student.getRoll());

System.out.println("Name is :"+student.getName());

System.out.println("Marks is :"+student.getMarks());

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

}

Note : from the List object we can get elements one by one by using following approaches also:

1.by using Iterator obj

2.by using ListIterator obj

3.by using forEach() method

4.by using Stream api

--in addition to the normal and enhanced for loop.

where as from Set and Queue:

--we can not use Normal for loop we can only use :

1.enhanced for loop

2.by using Iterator

3.by using forEach() method

4.by using Stream api

LinkedList:

========

--it is also one of the implementation of List interface.

--LinkedList class from Java 1.5 onwards also implements Deque interface.

--LinkedList class is the best choice if our frequent opeation is insertion or deletion from the middle.

--LinkedList class also follows the properties of List and Deque(preserve the sequence and index concept)

10 20 30 50 60

0 1 2 3 4 5

With the LinkedList if we delete or insert elements then too much siffting operation is not required.

--In Java LinkedList is implemented using Doubly linked list data structure.

example:

----------

package com.masai;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Scanner;

public class Demo {

public static void main(String[] args) {

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

ll.add(10);

ll.add(20);

ll.add(30);

ll.add(40);

ll.add(50);

ll.add(null);

ll.add(null);

ll.add(60);

ll.add(60);

System.out.println(ll);

// for(int x:ll) { // here the chance of NPE

// System.out.println(x);

// }

for(Integer x:ll) {

System.out.println(x);

}

}

}

Vector:

=======

--it is also similar to ArrayList class, with following differences.

1.AL introduced in java 1.2 v whereas Vector class is a legecy collection class introduced in java1.0 v

2.where AL reaches to the max capacity the new AL obj will be created internally with

newCapacity = (currentCapacity \* 3/2)+1

where as when Vector class reaches to its max capacity then a new vector obj is created in the memory by

double capacity.

3. most of the methods of AL class is non-synchronized (not thread safe) where as most of the methods of Vector class is synchronized, i.e thread-safe.

---AL will give better and fast performance compare to Vector class.

totalticket = 10;

8

9

public synchronized void bookTicket(int numberOfTicket){

//

if(numberOfTicket < = totalTicket)

{

totalTicket -= numberOfTicket;

}

}

public synchronized void viewAvailability(){

}

--until we have specific requierement we should not use synchoronized keyword.

---we have an option to make our AL objects methods as synchronized.

java.util.Collection(I) : root interface of Collection f/w

java.util.Collections(c) : utility class.

Collections.sychronizedList(al); // it will convert the AL to the synchronized List(thread safe obj)

Vector --- black & white tv

AL ---> Color TV -----> reduce the color

Stack class:

==========

Stack<Integer> st = new Stack<>();

st.add(10);

st.add(20);

st.add(30);

st.add(40);

st.add(50);

st.add(null);

st.add(null);

st.add(60);

st.add(60);

System.out.println(st);

System.out.println(st.pop());

System.out.println(st);

}

}

--All the opeation of the Stack we can perform with the help of LinkedList class also.

ArrayList<String> al = new ArrayList<>(); /// too specific

List<String> al= new ArrayList<>(); // recommended way.

Collection<String> al = new ArrayList<>();//

Object al = new ArrayList<>(); // too generic

example

Demo.java:

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

package com.masai;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.List;

import java.util.Scanner;

import java.util.Stack;

import java.util.Vector;

public class Demo {

public List<Student> getStudents(){

LinkedList<Student> students = new LinkedList<>();

students.add(new Student(10, "n1", 780));

students.add(new Student(12, "n2", 580));

students.add(new Student(13, "n3", 780));

students.add(new Student(14, "n4", 680));

return students;

}

public static void main(String[] args) {

Demo d1= new Demo();

List<Student> students= d1.getStudents();

for(Student student:students) {

System.out.println(student);

}

}

}

Set:

====

--it is the child interface of the Collection interface.

--it defines the behaviour of a collection that it does not allows the duplicate elements.

--here index concept is not applicable. so we can not use get(int index) method.

HashSet(c):

=========

--it is the first implementation of the Set interface.

--here insertion order is not preserved,

--elements will be added based on their hashCode.

--duplicate elements are not allowed.

--if we try to add any duplicate element, it does not throw any exception, simply add() method return false.

--null insertion is possible , but only one null value.

\*\*\*\*HashSet class is the best choice, if our frequent operation is searching.

--searching a paerticular element based on hashCode will have time complexity O(1).

HashSet<Integer> hs= new HashSet<>();

--here an empty HS obj is created with the initial capacity 16 and the default load factor(fill ratio) is 0.75

--here fill ratio means after completion of 75% the new HS object will be created in the memory.

HashSet<Integer> hs= new HashSet<>(1000,0.8f);

--here initial capacity is 1000 and and once reaches to the 80% then a new HS object is created in the memory.

example:

HashSet<Integer> hs= new HashSet<>();

hs.add(10);

hs.add(20);

hs.add(30);

hs.add(10);

hs.add(10);

hs.add(null);

hs.add(null);

System.out.println(hs.size());

System.out.println(hs);

output:

4

[null, 20, 10, 30]

Note: to access the elements one by one from the HS class we can not normal for loop,

but we can use enahanced forloop

example

for(Integer i1:hs) {

System.out.println(i1);

}

Note: HashSet is very much related with HashMap, it internally uses the HashMap to store the element.

Object equality:

=============

--equals() method belongs to Object class,

public boolean equals(Object obj);

--this method is implemented inside the Object class as follows:

public boolean equals(Object obj){

/\*

if(obj == this)

return true;

else

return false;

\*/

return obj == this;

}

--in order to make our class objects logically equal we need to override the above equals() method from the Object class to our Student class.

ex:

Inside Student.java:

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

@Override

public boolean equals(Object obj) {

Student s1= this;

Student s2= (Student)obj;

//if(s1.getRoll() == s2.getRoll() && s1.getName().equals(s2.getName()) && s1.getMarks() == s2.getMarks() )

// return true;

// else

// return false;

return (s1.getRoll() == s2.getRoll() && s1.getName().equals(s2.getName()) && s1.getMarks() == s2.getMarks() );

}

Note: this equals() method has a best friend called hashCode() method, it is also defiend inside the Object class:

public int hashCode();

---equals() and hashCode method is like a contract, if we override the equals() method to make our objects logically equal then we have to override the hashCode() method also;

--if we call equals() method to compare two object and if it returns true then those objects hashCode value should also be same.

Student.java:

==========

package com.masai;

import java.util.Objects;

public class Student {

private int roll;

private String name;

private int marks;

public Student() {

// TODO Auto-generated constructor stub

}

@Override

public boolean equals(Object obj) {

Student s1= this;

Student s2= (Student)obj;

// if(s1.getRoll() == s2.getRoll() && s1.getName().equals(s2.getName()) && s1.getMarks() == s2.getMarks() )

// return true;

// else

// return false;

return (s1.getRoll() == s2.getRoll() && s1.getName().equals(s2.getName()) && s1.getMarks() == s2.getMarks() );

}

@Override

public int hashCode() {

return Objects.hash(roll,name,marks);

}

public Student(int roll, String name, int marks) {

super();

this.roll = roll;

this.name = name;

this.marks = marks;

}

public int getRoll() {

return roll;

}

public void setRoll(int roll) {

this.roll = roll;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getMarks() {

return marks;

}

public void setMarks(int marks) {

this.marks = marks;

}

@Override

public String toString() {

return "Student [roll=" + roll + ", name=" + name + ", marks=" + marks + "]";

}

}

Demo.java:

========

package com.masai;

import java.util.HashSet;

public class Demo {

public static void main(String[] args) {

HashSet<Student> hs=new HashSet<>();

hs.add(new Student(10, "n1", 780));

hs.add(new Student(12, "n2", 880));

hs.add(new Student(13, "n3", 980));

hs.add(new Student(10, "n1", 780));

hs.add(new Student(10, "n1", 780));

System.out.println(hs.size());

}

}

Day16

from the add() method of the HashSet class following 2 method is called to identify the object uniqueness:

equals()

hashCode()

--if for 2 objects equals will returns true then their hashCode value should

return the same value for both object.

basically 3 cases where we need to override equals and hashCode method inside our class.

1. whenever we try to make our objects logically equal.

2. whenever we try to add our objects inside the HashSet or LinkedHashSet class.

3.whenever we try to add our object as a key inside the HashMap class.

LinkedHashSet:-

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

--it is the child class of HashSet class, it also does not allow duplicate, but

it maintains the insertion order.

Note: In collection f/w all the collection classes are mutually inclusive . i.e we can convert any collection classes to any other collection class very easily.

Removing duplicates from the ArrayList:

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

package com.masai;

import java.util.ArrayList;

import java.util.HashSet;

import java.util.LinkedHashSet;

public class Demo {

public static void main(String[] args) {

ArrayList<Integer> al = new ArrayList<>();

al.add(10);

al.add(20);

al.add(10);

al.add(10);

al.add(30);

al.add(20);

System.out.println(al);

LinkedHashSet<Integer> lhs=new LinkedHashSet<>(al);

System.out.println(lhs);

al=new ArrayList<>(lhs);

System.out.println(al);

}

}

removing duplicate from the String:

============================

package com.masai;

import java.util.ArrayList;

import java.util.LinkedHashSet;

public class Demo {

public static String removeDuplicateFromString(String original) {

char[] chr= original.toCharArray();

LinkedHashSet<Character> lhs=new LinkedHashSet<>();

for(char ch:chr) {

lhs.add(ch);

}

StringBuilder result= new StringBuilder("");

for(char ch: lhs) {

result.append(ch);

}

return result.toString();

}

public static void main(String[] args) {

String s= Demo.removeDuplicateFromString("ratan");

System.out.println(s);

}

}

Note: String class and all the wrapper classes has already overriden the

equals and hashCode method internally.

removing duplicate from the List:

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

Demo.java:

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

package com.masai;

import java.util.ArrayList;

import java.util.LinkedHashSet;

import java.util.List;

public class Demo {

public static List<String> removeDuplicateFromList(List<String> cities) {

LinkedHashSet<String> lhs= new LinkedHashSet<>(cities);

return new ArrayList<>(lhs);

}

public static void main(String[] args) {

List<String> cities= new ArrayList<>();

cities.add("delhi");

cities.add("delhi");

cities.add("mumbai");

cities.add("chennai");

System.out.println(cities);

List<String> resultList= removeDuplicateFromList(cities);

System.out.println(resultList);

}

}

TreeSet:

=======

--it has the nature of Collection, Set, SortedSet.

--In Java TreeSet class implemented by using balanced tree data structure.

--We use the TreeSet class, when we want to arrange our object in sorted order(natural sorting order)

--duplicates are not allowed.

--insertion order is not preserved.

--even a single null also not allowed, if we try to add a null value then at runtime we get a NPE.

example1:

TreeSet<Integer> ts= new TreeSet<>();

ts.add(10);

ts.add(2);

ts.add(5);

ts.add(12);

ts.add(6);

ts.add(15);

System.out.println(ts);

example2:

TreeSet<String> ts= new TreeSet<>();

ts.add("delhi");

ts.add("mumbai");

ts.add("kolkata");

ts.add("chennai");

ts.add("chandigarh");

System.out.println(ts);

example3:

Note: if we try to add any element inside the TreeSet object, then those elements should be comparable.

i.e That object class should implements "java.lang.Comparable" interface. otherwise we will get a ClassCastException at runtime.

Note: - all the Wrapper classes and String class implements Comparable interface.

Comparable interface:

=================

--this interface belongs to java.lang package.

--by using this interface, we will specify the sorting rules/technique inside the class.

--this interface is having only one abstract method:

public int compareTo(Object obj);

--if we try to add any class object inside the TreeSet, then that class must implement Comparable interface and override this compareTo method , and inside this method we need to specify the sorting technique of that class object.

Note: in TreeSet hashCode and equals() method is not used.

@Override

public int compareTo(Object o) {

//in this method, we need to specify the sorting rules.

//this method internally called by the add() method of TreeSet obj,

//when we try to add any element.

//s1.compareTo(s2);

//if s1 is bigger than s2 it returns +1

//if s2 is bigger than s1 it retunes -1

//if both s1 and s2 are equal then it returns 0;

}

example

Student.java:

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

package com.masai;

public class Student implements Comparable{

private int roll;

private String name;

private int marks;

public Student() {

// TODO Auto-generated constructor stub

}

public Student(int roll, String name, int marks) {

super();

this.roll = roll;

this.name = name;

this.marks = marks;

}

public int getRoll() {

return roll;

}

public void setRoll(int roll) {

this.roll = roll;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getMarks() {

return marks;

}

public void setMarks(int marks) {

this.marks = marks;

}

@Override

public String toString() {

return "Student [roll=" + roll + ", name=" + name + ", marks=" + marks + "]";

}

@Override

public int compareTo(Object o) {

Student s1= this;

Student s2= (Student)o;

if(s1.getMarks() > s2.getMarks())

return +1;

else if(s1.getMarks() < s2.getMarks())

return -1;

else

return 0;

}

}

--the above sorting logic is based on the marks..

Note: in order to check the object uniqueness

HashSet and LinkedHashSet class uses equals() and hashCode() metho

where as TreeSet class uses compareTo() method. if compareTo() returns 0.

above example using generics:

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

public class Student implements Comparable<Student>

{

--

--

@Override

public int compareTo(Student s) {

if(this.getMarks() > s.getMarks())

return +1;

else if(this.getMarks() < s.getMarks())

return -1;

else

return 0;

}

}

not removing duplicate:

===================

@Override

public int compareTo(Student s) {

if(this.getMarks() > s.getMarks())

return +1;

else

return -1;

//return this.getMarks() > s.getMarks() ? +1:-1;

}

example sort the student based on name:

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

@Override

public int compareTo(Student s) {

return this.getName().compareTo(s.getName());

}

reverse order:

===========

@Override

public int compareTo(Student s) {

return s.getName().compareTo(this.getName());

}

if Marks are same then sort them according to thier name:

================================================

@Override

public int compareTo(Student s) {

if(this.getMarks() > s.getMarks())

return +1;

else if(this.getMarks() < s.getMarks())

return -1;

else

return this.getName().compareTo(s.getName());

}

java.util.Comparator(I):

===================

--by using interface, we can define the sorting technique for a class objects from outside of that class.

Note: if we want to define a sotring rule of a class objects inside the same class then we should use

java.lang.Comparable interface

--whereas if we want to define the sorting logic outside of that class then we need to use

java.util.Comparator interface.

--with the help of Comparable we can define only one sorting logic, where as with the help of Comparator we can define multiple sorting logic.

--this Comparator interface has also only one abstract method:

public int compare(Object obj1, Object obj2);

--to use the Comparator :-

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

step 1: here we need not polute the Student class(java bean class) by implementing Comparable interface.

step 2: create a seperate class by any name and implements the Comparator interface and define the sorting logic by overriding the compare(--) method.

ex:

StudentMarksComp.java:

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

package com.masai;

import java.util.Comparator;

public class StudentMarksComp implements Comparator{

@Override

public int compare(Object o1, Object o2) {

Student s1= (Student)o1;

Student s2= (Student)o2;

if(s1.getMarks() > s2.getMarks())

return +1;

else if(s1.getMarks() < s2.getMarks())

return -1;

else

return 0;

}

}

by using generics:

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

package com.masai;

import java.util.Comparator;

public class StudentMarksComp implements Comparator<Student>{

@Override

public int compare(Student s1, Student s2) {

if(s1.getMarks() > s2.getMarks())

return +1;

else if(s1.getMarks() < s2.getMarks())

return -1;

else

return 0;

}

}

step 3:

--create the above StudentMarksComp class object and pass that object to the constructor of the TreeSet class.

example

package com.masai;

import java.util.ArrayList;

import java.util.LinkedHashSet;

import java.util.LinkedList;

import java.util.List;

import java.util.TreeSet;

public class Demo {

public static List<String> removeDuplicateFromList(List<String> cities) {

LinkedHashSet<String> lhs= new LinkedHashSet<>(cities);

return new ArrayList<>(lhs);

}

public static void main(String[] args) {

// StudentMarksComp mcomp = new StudentMarksComp();

// TreeSet<Student> ts= new TreeSet<>(mcomp);

TreeSet<Student> ts= new TreeSet<>(new StudentMarksComp());

ts.add(new Student(10, "n1", 780)); //s1

ts.add(new Student(12, "n2", 680)); //s2

ts.add(new Student(13, "n5", 480));

ts.add(new Student(14, "n4", 480));

ts.add(new Student(15, "n3", 480));

System.out.println(ts.size());

System.out.println(ts);

}

}

sorting according to the name:

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

package com.masai;

import java.util.Comparator;

public class StudentNameComp implements Comparator<Student>{

@Override

public int compare(Student s1, Student s2) {

return s1.getName().compareTo(s2.getName());

}

}

utility or tools classes in collection f/w:

==============================

1.java.util.Arrays class: it is mostly used for performing unility operation on the normal arrays

for examaple : sorting, searching, reversing, converting to list, printing the arrays elements. etc,

2.java.util.Collections class: it is mostly used to perform utility operations for the collection f/w related classes.

java.util.Arrays:

============

--inside this class we have variaous static methods are there by using

which we can perform various operations on the normal array object.

ex1:

printing the elements from array:

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

int[] arr= {12,23,34,45,55,12,22};

// String result= Arrays.toString(arr);

//

// System.out.println(result);

System.out.println(Arrays.toString(arr));

sorting an array:

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

int[] arr= {12,23,34,45,55,12,22};

System.out.println(Arrays.toString(arr));

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));

searching an element in array:

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

--first of all our array should be sorted, otherwise we may not get

correct result.

--if the value is found then we get index value otherwise we get the

negetive value.

example

int[] arr= {12,23,34,45,55,12,22};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));

int index= Arrays.binarySearch(arr, 24);

System.out.println(index);

creating List in easiest manner:

=========================

List<String> cities= Arrays.asList("Delhi","Mumbai","Chennai","kolkata");

List<Integer> list= Arrays.asList(10,20,30,40,50,20,60);

List<Student> students= Arrays.asList(

new Student(10, "n1", 780),

new Student(10, "n1", 780),

new Student(10, "n1", 780),

new Student(10, "n1", 780));

}

Collections class:

=============

--this class also defines some of the static methods to perform some

utility operations on the Collection f/w related classes.

Collections.sort:

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

List<String> cities= Arrays.asList("Delhi","Mumbai","Chennai","kolkata");

Collections.sort(cities);

System.out.println(cities);

example 2:

List<Student> students= Arrays.asList(

new Student(12, "n1", 780) new Student(13, "n2", 580), new Student(14, "n3", 480),

new Student(15, "n4", 980));

Collections.sort(students, new StudentMarksComp());

System.out.println(students);

}

getting number of occuerrence in a List:

================================

List<Integer> list= Arrays.asList(10,12,10,10,12,45,55);

int result= Collections.frequency(list, 10);

System.out.println(result);

converting List to the synchronized list:

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

List<Integer> list= Arrays.asList(10,12,10,10,12,45,55);

List<Integer> sList= Collections.synchronizedList(list);

additional

Var-args:

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

--variable number of argumetns.

--introduced in JAva 1.5 v

--this concept is developed based on array concept only but in

much more simplified way.

example:

package com.masai;

public class Demo {

public static int add(int... arr) {

int result=0;

for(int i:arr) {

result+=i;

}

return result;

}

public static void main(String[] args) {

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

System.out.println(add(10,20,30,40));

System.out.println(add());

//System.out.println(add(null));

System.out.println(add(nums));

}

}

Note: we can access var-args elements using zero based index, simillar to normal array.

rules of using var-args:

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

1. if we want to declare any other parameters along with the var-args then this var-arg parameter must be the last parameter.

example:

package com.masai;

public class Demo {

public static int add(String name, int... arr) {

int result=0;

for(int i:arr) {

result+=i;

}

return result;

}

public static void main(String[] args) {

System.out.println(add("Hello"));

System.out.println(add("Hello",10));

System.out.println(add("Hello",20,30));

System.out.println(add("Hello",null));

}

}

2. inside a single method we can not have 2 var-args as a parameter.even for different types also.

3. if multiple overloaded methods are there then var-args gets the least priority.

example:

package com.masai;

public class Demo {

public static int add(int... arr) {

int result=0;

for(int i:arr) {

result+=i;

}

return result;

}

public static int add(int i) {

System.out.println("inside add(int i)");

return 0;

}

public static void main(String[] args) {

System.out.println(add(10));// second add metthod will be called

}

}

4. along with var-args we can not define another overloaded method which

takes array of the same type parameter. it will become duplicate method

defination.

example2:

package com.masai;

public class Demo {

public static void fun1(Student... students) {

for(Student s:students) {

System.out.println(s);

}

}

public static void main(String[] args) {

Student[] stduetns= {

new Student(10, "n1", 780),

new Student(12, "n2", 780),

new Student(13, "n3", 780),

new Student(14, "n4", 780),

};

fun1(stduetns);

}

}

Enum:

======

--with the help of enum we can create our own datatype (enumerated data type)

--in java we have 8 primitive data types are their.

boolean b= true/ false

byte b = -128 to 127 = 256 // here 256 values are allowed.

--if we want to create a datatype where we allow only certain set of values

then we should use enum.

enum of Month :

Month.java

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

package com.masai;

public enum Month {

JAN,FEB,MAR,APR,MAY;

}

--enum can be created in 2 ways:

1.as a seperate .java file like a class or interface (we can compile this

enum and can generate a .class file also)

2.as inner enum inside a class(like a inner class.)

--enum contants should be in upper case (naming convension)

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

Month m=Month.JAN;

System.out.println(m);

}

}

---enum concept is introduced in java 1.5v

--every enum contant is implemented by using class concept.

--every enum constant is always "public static final".

--every enum constant represented an object of type enum.

--the above enum Month will converted internally as follows:

public final class Month extends Enum{

public static final JAN= new Month();

public static final FEB= new Month();

public static final MAR= new Month();

public static final APR= new Month();

public static final MAY= new Month();

}

java.lang.Enum class is a predefiend an abstract class.

--this Enum class already implements Comparable and Serializable

interface internally.

Note: inheritance concept is not applicable with an enum.

but an enum can implements any number of interfaces.

--inlike other languages enum in java is more powerfull because,

we can have variables, methods, constructors inside an enum also.

--even we can place main method inside an enum and execute that

as similar to class.

example:

package com.masai;

public enum Month {

JAN,FEB,MAR,APR,MAY;

public static void main(String[] args) {

System.out.println("inside main of enum");

}

}

values() and ordinal() method:

=========================

--Every enum implicitly contains values() method that returns all the values presents inside an enum.

--values() method will return all the values in the form of array .

this value() method is a static method which we can call on any enum.

--withing the enum every constant placed based on the order, and we can

find the Ordinal values of enum constant by uisng ordinal() method.

--this ordinal values are zero based index value.

--ordinal() method is non-static method.

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

Month[] months= Month.values();

for(Month month:months) {

System.out.println(month+"=========="+month.ordinal());

}

}

}

methods and variables inside an enum:

================================

package com.masai;

public enum Month {

JAN,FEB,MAR,APR,MAY; // here if we place any other elements then

//this semicolon is mandatory otherwise it is optional

int i=10;

void fun1() {

System.out.println("inside fun1");

}

}

constructor in enum:

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

--we can place constructor also inside an enum, that constructor can only be

private whether we mension it or not.

--a constructor of an enum will be executed seperatly for every enum constant at the

time of enum class loaded into the memory.

Month.java:

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

package com.masai;

public enum Month {

JAN,FEB,MAR,APR,MAY;

private Month() {

System.out.println("isnide constructor of Month");

}

}

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

Month m= Month.JAN;

}

}

output:

isnide constructor of Month

isnide constructor of Month

isnide constructor of Month

isnide constructor of Month

isnide constructor of Month

Example :

Month.java:

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

package com.masai;

public enum Month {

JAN,FEB(28),MAR(31),APR,MAY;

private Month() {

System.out.println("isnide constructor of Month");

}

private Month(int num) {

System.out.println("isnide constructor of Month(int num)");

}

}

--here for FEB and MAR, paramterized constructor and for remaining zero

argument constructor will be executed.

Item.java:

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

package com.masai;

public enum Item {

SUGER,SALT,PENCIL(5),PEN(10),SHARPNER(30);

int price;

private Item() {

this.price=20;

}

private Item(int price) {

this.price=price;

}

public int getPrice() {

return this.price;

}

}

Demo.java:

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

package com.masai;

public class Demo {

public static void main(String[] args) {

Item[] items= Item.values();

for(Item item:items) {

System.out.println(item+" price is : "+item.getPrice());

}

}

}

exmaple 2:

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

City.java:

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

package com.masai;

public enum City {

DELHI{

@Override

public void message() {

System.out.println("Welcome User you are in Capital");

System.out.println("Tower in your area is :"+numberOfTowers);

}

},MUMBAI,KOLKATA,CHENNAI("50 towers");

String numberOfTowers;

private City() {

this.numberOfTowers="100 towers";

}

private City(String numberOfTowers) {

this.numberOfTowers=numberOfTowers;

}

public void message() {

System.out.println("Welcome User");

System.out.println("Tower in your area is :"+numberOfTowers);

}

}

Demo.java:

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

package com.masai;

import java.util.Scanner;

public class Demo {

public void printCity(City city) {

if(city != null) {

city.message();

}

else

System.out.println("invalid city");

}

public static void main(String[] args) {

Demo d1= new Demo();

Scanner sc=new Scanner(System.in);

System.out.println("Enter the City name");

String cityName= sc.next();

//converting String to the appropriate enum type.

City city= City.valueOf(cityName.toUpperCase());

//for any invalid city , it will throw an exception

d1.printCity(city);

}

}

Dayextra

Date and Time api:

================

Prior to Java 1.8 we use following classes to handle data and time in java:

--these classes are from

java.util

java.text package

the classes are :

java.util.Date(c)

java.util.Calander(c)

java.text.SimpleDateFormat

from java 1.8 onwards a new Date and time api introcuded in

java.time package.

Demo.java:

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

package com.masai;

import java.time.LocalDate;

import java.time.LocalDateTime;

import java.time.LocalTime;

import java.time.ZonedDateTime;

public class Demo {

public static void main(String[] args) {

LocalDate ld=LocalDate.now();

System.out.println(ld);

LocalTime lt=LocalTime.now();

System.out.println(lt);

LocalDateTime ldt=LocalDateTime.now();

System.out.println(ldt);

ZonedDateTime zdt=ZonedDateTime.now();

System.out.println(zdt);

}

}

G - Era(AD BC)

y - year( yy(18) or yyyy(2018))

M - Month(M(9) or MM(09) or MMM(Sep))(MMMMM--September)

d - day(d(23) or dd(23) or ddd(023))

E - day in a weak(E (sun))(EEEE--Sunday)

a - am pm

h - hour in am or pm (1-12)

hh - hour in am or pm (01-12)

H - hour of day in 24 hour form (0-23)

HH - hour of day in 24 hour form (00-23)

m - minute (4)

mm - minute (04)

s - second (4)

ss - second(04)