1/11/2019

*\* Non Static Members:*

-class level members which don’t have static keyword

-get memory Location when Object is created

-called as Instance Variable

-access in only one way

ClsName var=new ClsName();

var.non-staticmember;

-where as static can access in 3 ways

1. ClsName.m1();

2. ClsName e=null;

e.m1();

3.ClsName var=new ClsName();

var.staticmember;

Every object create separate Copy of Memory. which is not modified by Another Object. if we modify static variable by using one object its affected to all other objects.

package com.naren.NonStaticvariables;

public class NonstaticEX {

int i=90;

int j=100;

public static void main(String[] args) {

NonstaticEX n= new NonstaticEX();

NonstaticEX n1= new NonstaticEX();

System.out.println(n.i+" "+n.j);

System.out.println(n1.i+" "+n1.j);

n.i=30;

n.j=80;

System.out.println(n.i+" "+n.j);

System.out.println(n1.i+" "+n1.j);

}

}

Op:90 100

90 100

30 80

90 100

-Why non static member cannot access static members?

Non static variables guarantee to get the memory location before execution that’s why can not access non static content from static.

this Keyword:

- current object reference

-this is used to non static final variable ,only non static can allow

- compiler automatically place this keyword in non static members

` -where as constructor argument variable name and local var name same then compiler does not add this keyword, values not change in local variable ,so we can give this Keyword explicitly

Pass By Value And Call By Reference

-Calling a method by a variable is called Call By value

- Calling a method by a address is called Call By address

-java does not support call by address

- method param have Object reference var but it holds the class rep data that’s why java does not support call by reference

-where as java not supported pointers

2/11/2019

CONSTRUCTOR

Constructor is a Special method in java. it not contain return type,

-Constructor role is only initializing object, new key word is creating Object.

-Constructor calling with only new keyword else it leads compile time error

-Where as class is representation of data, constructor is holding the data

- Constructor has allowed all four types of Accessibility Modifiers. But private is used in singleton Class

-types of constructors

1.Default : compiler Given

2.No-arg : programmer given without params

3.Parameterized: Programmer

publicclass ConVsMethod {

public ConVsMethod() {

// TODO Auto-generated constructor stub

System.out.println("constructor");

}

publicstaticvoidConVsMethod() {

// TODO Auto-generated constructor stub

System.out.println("method");

}

publicstaticvoid main(String[] args) {

// TODO Auto-generated method stub

//calling cons

ConVsMethod c=new ConVsMethod();

//calling method

ConVsMethod();

}

}

***Constuctor overloading:***

* Defining multiple constructor with different parameters
* Same Logic execute in all constructors. we can define non-static block is recommended

Non Static Block

* One time execution block at the time of object creation
* Super(),this must be first statement
* Only one allowedin one constructor

package com.naren.NonStaticMembers;

publicclass NonStaticBlock {

staticinti=10;

// by using static and non static variables we can call constructor

publicstatic NonStaticBlock n1=new NonStaticBlock();//1

//NonStaticBlock n2=new NonStaticBlock() stack overflow

static {//4

System.out.println(" 1 block"+i+" "+NonStaticBlock.b);

}

public NonStaticBlock() {

// TODO Auto-generated constructor stub

System.out.println("cons");//3//6

}

{

System.out.println("2 block");//2//5

}

publicstaticvoid main(String[] args) {

// TODO Auto-generated method stub

// NonStaticBlock n=new NonStaticBlock();

}

staticintb=9;

}

5/11/2019

Q) how many ways we can load a Class ?

1. using java command

Java ClassName

Ex: java Example;

2. From another class , by calling Static member

package com.nare.howmanywayscallclass;

publicclass Ex {

static {

System.out.println(" static content is loaded whenever class is loaded");

}

staticvoid m()

{

System.out.println("by calling static");

}

Ex(){

System.out.println("ex constructor");

}

}

package com.nare.howmanywayscallclass;

publicclass Test {

publicstaticvoid main(String[] args) {

// TODO Auto-generated method stub

Ex.m();

}

}

Op: static content is loaded whenever class is loaded

by calling static

3.From another class , by callingconstructor

package com.nare.howmanywayscallclass;

publicclass Test {

publicstaticvoid main(String[] args) {

// TODO Auto-generated method stub

Ex e=new Ex();

}

}

Op: static content is loaded whenever class is loaded

ex constructor

4. By using Reflection Api, Class.forName(“cls name”)

package com.nare.howmanywayscallclass;

publicclass Test {

publicstaticvoid main(String[] args) throws Exception {

// TODO Auto-generated method stub

Class.forName("com.nare.howmanywayscallclass.Ex");

}

}

Op: static content is loaded whenever class is loaded

OOPS FUNDAMENTALS

Oop: Object Oriented Programming

Represents real world Objects

Building blocks of oops are class and object

Class: class is blue print or template of an object

Object:Object is Physical reality of class or An Instance of a class

- A real world object contains three characteristics

1.State: It is Object Properties

Ex:like name,height,weight

2.Behavior:It is Operation of object

Ex:like sleep,walk.run

3.identity: Itis an Identity of Object

Ex: name

Object Relations

Java supports three relations

1. Is-A(Inheritance)

* Student Is-A Person
* Employee Is-A Person

Implement Is- A relation using

1. extends : is used b/w two classes or two interfaces
2. implements: is used b/w class and interface

2. Has-A (Composition)

Creating datatype as a child class and creating variable inside parent class is called Composition. so one object can exist another object

publicclass Test {

private Ex e;

}

3.Uses-A (Aggregation)

Uses-A is oneObject uses another object for performing one operation

publicclass Test {

publicvoid m(Ex e) {

System.out.println(" using Ex Object");

}}

**OOP Principles:**

***1.Encapsulation:***

The process of creating a class by hiding internal data from out side class is called Encapsulation. And accessing it only publicly exposed methods

The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.

-In java we can achieve encapsulation

1.By declaring all variables as private

2.by definining one pair setter and getter methods

Uses:

We can stop unauthorised access

We can stop wrong values

**Ex:**

**package** com.nare.encapsulation;

**publicclass** Beans {

**privatedouble**balance;

**publicdouble** getBalance() {

**return**balance;

}

**publicvoid** setBalance(**double**balance) {

**if**(balance<0) {

System.***out***.println("do not enter negative value");

}

**else** {

**this**.balance=balance;

}

**this**.balance = balance;

}

}

**package** com.nare.encapsulation;

**publicclass** Test {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stubBE

Beans b=**new** Beans();

//b.balance; we cont get this not publicly exposed member

b.setBalance(500);

**double**d=b.getBalance();

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

}

***2.Inheritance:***

The process of creating class to reuse existed class members using our own classname or object reference.

Advantages :

1.Reusability

2.Runtime Polymorphism ( dynamic binding)

Inheritance can be achieve in two ways

1. extends : is used b/w two classes or two interfaces
2. implements: is used b/w class and interface

1.class A extends class B

2.interface I1 extends interfaceI2

3. class C implements interface I

6/11/2019

Q). How can we access super class members in subclass ?

-static members directly access by their name or sub class name

-non-static we can access by sub class object.

Ex:

**package** com.nare.inheritance;

**publicclass** SuperEx{

**staticint***i*=52;

**int**j=10;

}

**package** com.nare.inheritance ;

**publicclass** SubEx **extends** SuperEx {

**publicstaticvoid** main(String[] args) {

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

System.***out***.println(SubEx.*i* +" Static var "+SuperEx.*i*+" we can access inthis 3 ways "+*i*);

System.***out***.println(**new** SuperEx().j+" non static variable " +**new** SubEx().j);

}

}

OP: 52 Static var 52 we can access inthis 3 ways 52

10 non static variable 10

Q) Compiler activities in inheritance ?

Compiler first search members in subclass and then search in super class if not available compiler throws “**cannot find Symbol”,** Jvm also search like this.

* Whenever class loading execution order is super to sub class
* Whenever sub class obj is created super class non static blocks executed or invoked.

***Static Members Flow in inheritance***

* Whenever class loading execution order is super to sub class
* Static variables executed from super to sub class
* Main method is not available in sub class then it will go to super class
* Member search in sub class next it will search super class

***Non Static Members Flow in inheritance***

* Non Static variables executed from super to sub class
* Sub class obj is created then non static members executed from super to sub class
* Main method is not available in sub class then it will go to super class
* Member search in sub class next it will search super class
* Sub class constructor is created memory for super class variables

Ex: Static And Non Static Flow

**package** com.nare.inheritance.staticflow;

**publicclass** StaticFlowsuper {

**staticint***i*=*m*();

**publicstaticint** m() {

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

System.***out***.println("super m");

**return** 1;

}

{

System.***out***.println("Super non staticblock ");

}

**static** {

System.***out***.println(" super static");

}

**publicstaticvoid** main(String[] args) {

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

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

}

}

**package** com.nare.inheritance.staticflow;

**publicclass** FlowSUb **extends** StaticFlowsuper{

{

System.***out***.println(" sub non static block");

}

**static**{

System.***out***.println(" sub static block");

}

**publicstaticvoid** main(String[] args) {

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

}

}

Op:super m

super static

sub static block

Super non staticblock

sub non static block

***super() Working functionality***

-By default compiler will place super() in all zero param constructors

-Every class super class is Object class

- Parameterized constructors will executed by pausing super class matching arguments Constructor

- we can not invoke super class all constructors those are

-private constructors

-default is access only with in the package

- Constructor (super()) call must be the first statement in a constructor

Ex:

**package** com.nare.inheritance.supercall;

**publicclass** SuperEx {

**public** SuperEx() {

//System.out.println("");Constructor call must be the first statement in a constructor

**super**();

System.***out***.println("super constructor");

}

**public** SuperEx(**int**i) {

**this**();// this is used to class constructors within the claass

System.***out***.println("param constructor"+i);

}

}

**package** com.nare.inheritance.supercall;

**publicclass** SubEx **extends** SuperEx {

**public** SubEx() {

**super**(90);

}

**publicstaticvoid** main(String[] args) {

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

SubEx s=**new** SubEx();

}

}

Op:super constructor

param constructor90

**Sub Class object structure**

Q) if we create sub class object , will super class obj created?

No, when sub class object , will super class obj not created only get memory to the super class members

**package** com.nare.inheritance.subobjstructure;

**publicclass** Sup {

**public** Sup() {

System.***out***.println("Super constructor "+ **this**);

}

}

**package** com.nare.inheritance.subobjstructure;

**publicclass** Sub **extends** Sup{

**public** Sub() {

System.***out***.println("Sub constructor "+ **this**);

}

**publicstaticvoid** main(String[] args) {

Sub s=**new** Sub();

}

}

OP:Super constructor com.nare.inheritance.subobjstructure.Sub@6d06d69c

Sub constructor com.nare.inheritance.subobjstructure.Sub@6d06d69c

**this& super**

-this() is used to call this class param constructors

-this.member is called current class members

-super() is call super class constructor

-super.member can access super class memberes

**-cannot use super,this in the staticcontext**

**Q)Why we connot use this and super in static context?**

**static context call at the time of class loading this and super member may or may not not get memory location at the time of class loading thats why we connot this and super in static context. If we are call hidden static method we can call using classname.method().**

Ex :**package** com.nare.inheritance.subobjstructure;

**publicclass** Sup {

**public** Sup(**int**i) {

System.***out***.println("super class constructor param constructor ");

}

**public** Sup() {

**super**();

}

**publicint**m1=20;

**publicvoid** m1(){

System.***out***.println("super class memeber");

}

}

**package** com.nare.inheritance.subobjstructure;

**import**com.nare.inheritance.supercall.SuperEx;

**publicclass** Sub **extends** Sup{

**public** Sub(){

**super**(10);

System.***out***.println("sub class constructor");

}

**public** Sub(**int**i) {

**this**();

System.***out***.println("sub class param constructor" );

}

**publicint**m1=10;

**publicvoid** m1(){

System.***out***.println("sub class member");

System.***out***.println(**super**.m1);

System.***out***.println(**this**.m1);

**super**.m1();

}

**publicstaticvoid** main(String[] args) {

Sub s=**new** Sub(5);

s.m1();

}

}

Op:

super class constructor param constructor

sub class constructor

sub class param constructor

sub class member

20

10

super class member

***Polymaorphism:***

The process of creating a method with different implementation to perform different behaviors . In simple terms creating multiple methods with different behaviours is called.

Wecan achieve polymorphism in two ways

1) By using Method Overriding

2) By using MethodOverloding

**MethodHiding,Overriding and Over Loading**

Redefining Super class **static** method in subclass with same prototype is called method hiding . we cannot hide method in same class

Redefining Super class **non-static** method in subclass with same Prototype is called

Method Overriding.we cannot override method in same class, because we con’t write two method with same prototype

Defining multiple methods with same name with different parameters is called

Method overloading. We can over load method in same class and its sub class, because it has different signatures.

***Execution Flow***

-Jvm executes static and overloaded method from the same reference variable type. thats why java 8 introduced static methods in interface. While as interface reference variable is call static method .Inter face cannot create object but sub class object reference is used to call interfaceimpl methods.

-but where as non –static or overriden methods is executed from current object reference.

-For example servlet container calls service method with our class object reference.

So that our class service method will executed

Ex:

**package** com.nare.polymorphism;

**publicclass** MethodhidingLoadindRidding {

**publicstaticvoid** m() {

System.***out***.println("super static method");

}

**publicvoid** m2() {

System.***out***.println("super non static");

}

**publicvoid** m3() {

System.***out***.println("super m3 no param");

}

**publicvoid** m3(**int**i) {

System.***out***.println("super m3 int param");

}

}

**package** com.nare.polymorphism;

**publicclass** MethodhidingLoadindRiddingImpl **extends** MethodhidingLoadindRidding {

**publicstaticvoid** m() {

System.***out***.println("sub static method");

}

**publicvoid** m2() {

System.***out***.println("sub non static");

}

**publicvoid** m3(**int**i,**float**a) {

System.***out***.println("sub m3 int, float param");

}

**publicstaticvoid** main(String[] args) {

MethodhidingLoadindRiddingImpl i1=**new** MethodhidingLoadindRiddingImpl();

i1.*m*();

i1.m2();

i1.m3();

i1.m3(5);

i1.m3(5,4);

MethodhidingLoadindRidding i=**new** MethodhidingLoadindRiddingImpl();

i.*m*();

i.m2();

i.m3();

i.m3(5);

//i.m3(5,4);

}

}

Op:sub static method

super non static

super m3 no param

super m3 int param

sub m3 int, float param

super static method

super non static

super m3 no param

super m3 int param

**Types Of polymorphism**

*1) Compile time polymorphism:*

The method bind at compile time and executed runtime in same class is called compile time polymorphism.

-static , OverLoaded and private overridden methods are comes under compile time polymorphism.

2*) Run time polymorphism:*

The method bind at compile time at one class and executedruntime at another class is called compile time polymorphism.

* Only non-static visible overridden methods come under run time.
* We must invoke a method with super class reference variable and its stores subclass object reference then only we get run time polymorphism

MethodhidingLoadindRidding i=**new** MethodhidingLoadindRiddingImpl();

**Rules in method overriding /hiding**

1)return type must same as super class method

**package** com.nare.polymorphism.exeflow2;

**publicclass** Ex {

**void** m1() {

}

}

**publicclass** Sample **extends** Ex {

**void**m1() {}// return type must be same otherwise CTE

}

2)static modifier should not addor remove

3)accesibility modifier same as super class or it can not increased

4)throws clause same as super class

**Over Loading methods execution**

1).Method executed based on given Argument type

2).if given argument is not matched search for widening

Same type-> widening->AB->AB widening-> var arg type ->var arg widening->ab var arg type🡪 AB var arg type widening

int🡪double 🡪Integer🡪Double🡪integer..-->double..\_-->Integer..-->Double..

**This flow is search in sub class then its search from super class if same type is matching ok otherwise it goes widening .......**

**final methods**

A method which has final keyword is called final method, we cannot override final method

**-c**an we overload final method ?

Yes we can overload final method override is not possible

Q) can we overload main method?

Yes , jvm is treated as normal method

Q) can we declare main method as final?

Yes,its posiible

A class which has final key word is called final class. Which is stop creating sub class.

**Abstraction:**

Abstraction means Hiding internal object operation implementation details from the user and providing necessary details to the user is called Abstraction

-by providing abstract keyword to method is called abstract method , class or interface.

-abstract method doesn’t havebody if you write its CE

-subclass force to override abstract methods.

-A class has a abstract keyword is called abstract Class. Which is partially implemented class. we can write concrete and abstract methods in abstract class.

- Abstract methods operations implemented by subclass sub class . which separate to every user

-concrete method implement in abstract class it is common to all subtypes.

Link: <https://www.geeksforgeeks.org/abstraction-in-java-2/>

**Encapsulation vs Data Abstraction**

1. [Encapsulation](http://contribute.geeksforgeeks.org/encapsulation-in-java/) is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
2. While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

**Advantages of Abstraction**

1. It reduces the complexity of viewing the things.
2. Avoids code duplication and increases reusability.
3. Helps to increase security of an application or program as only important details are provided to the user.

**package** com.app.abstraction;

**publicabstractclass** ApsrtcBus {

//this is separate to every object

**publicabstractvoid**users() ;

//this is same for all users

**void** cost() {

System.***out***.println(" seat cost is same 300");

}

}

**package** com.app.abstraction;

**publicclass** RedBus **extends** ApsrtcBus {

@Override

**publicvoid** users() {

System.***out***.println("these are red bus users");

}

}

**package** com.app.abstraction;

**publicclass** Abhibus **extends** ApsrtcBus {

@Override

**publicvoid** users() {

System.***out***.println("these are Abhi bus bus users");

}

}

**Q)** Why we can not instantiate to the abstract class ?

Because it s not fully implemented class . assume if we create object to abstract class we con not invoke un implemented methods that’s why we cannot create object tothe abstract class.

Q) can we declare abstract method as final or private ?

No, because it should be override in sub class.

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

**interface**

interface also a class. Which is fully un implemented class

-and also called pure abstract class

-it allows only abstract methods and public static final variables.

- in java 8 onwords we can write default and static methods in interface

-Java does not support **multiple inheritance** with classes only possible with interface

**Rules to develop interface**

**1).** Interface can not create concrete methods.

**publicinterface** Ex {

**void**m1() {//interce methods cannot have a body

}}

2).inteface have only public static final variables. if you create non static or non final compiler converts as final and static.

**int*a***=10; 🡪 public static final int a=10; compiler converts automatically

**3)** only Allowed public members

4) interface variable initialise at the time of creation else it throws cmp error because they are final variables

5)interface cannot be instantiated, But its reference variable is created for storing subclass object reference to develop loosely coupled to get runtime

6) we can not create interface as final

Q) can we apply abstract keyword to interface ?

Yes , But its removed at the time of compiling.

**Q)**Can we declare empty interface in java ?

Yes, Is possible..When you use some **marker interface**(no method implementation ) you want to tell jvm or compiler for adding or checking something during runtime or compile time..

Ex marker interfaces are

1)Serializable

2)Cloneable

Q)Why should we use marker interface in Java?

**Marker interface** is **used** as a tag to inform a message to the **Java** compiler so that it can add special behaviour to the class implementing it. ... When a **Java** class is to be serialized, **you should** intimate the **Java** compiler in some way that there is a possibility of serializing this **java** class.

7) the class should be derived from interface we should implement all abstract methods form interface otherwise class should be declared as **abstract.**

8)interface impl methods default accessibility modifier is public, we should implement as public .

***Main diference b/w abstract and interface is Multiple inheritance we can achieve multiple inheritance with only by using interface and interface is fully un implemented class. similarities is we can not instantiated both classes and both are not declared final because Those are implemented by sub classes***

**Types of inheritance**

1) Single Inheritance

**publicclass/interfaceSuper** {

}

**publicclassSubextends/interface** Super {

}

2) Multi level inheritance

**publicclass** Sub1 **extendsSub** {

}

3) Hierarchical inheritance

**publicclass** Sub **extends** Super {

}

**publicclass** Sub1 **extends**Super {

}

4) Hybrid Inheritance

By combining all types is called . In real time we Used Hybrid.

**5) Multiple Interface inheritance**

-Java does not support multiple inheritance with classes.

**publicclass** Sub1 **extends**Super1class,Super2class {

//Its not supported in java

}

-But its support with only interfaces

**publicinterface** Super {

}

**publicinterface** Super1 {

}

**publicclass** Sub1 **implements** Super,Super1{

}

***By default All classes super class is Object class it is changed by comiler , By default interface reference to call Object class methods***

**Identify Multiple inheritance is possible or not in below cases:**

1)Both interfaces has same methodname with **same prototype**, yes its possible , subclass implements **only once**.

**publicinterface** Super {

**void** m1();

}

**publicinterface** Super1 {

**void** m1();

}

**publicclass** Sub1 **implements** Super,Super1{

@Override

**publicvoid** m1() {

}

}

2)Both interfaces has same method name with **different params**, yes its possible , subclass should **implements both methods**.

**publicinterface** Super {

**void** m1(**int**i);

}

**publicinterface** Super1{

**void** m1(String s);

}

**publicclass** Sub1 **implements** Super,Super1{

@Override

**publicvoid** m1(String s) {

}

**publicvoid** m1(**int**i) {

}

}

3)Both interfaces has same method name with **same prototype but return type different**,Not possibleimplementation , because its violating method overriding rule,

Refer Book 2 page no:124

Q) How can we access interface variable from sub class?

By using variable namedirectly, interface variable cannot changed in sub classbecause it is final variable by default.

**publicinterface** Super {

**int*i***=10;

}

**publicclass** Sub1 **implements** Super{

**publicstaticvoid** main(String[] args) {

System.***out***.println("interface varible"+***i***);

//super.i=19; //CE final field can not change

}

}

\*\*\*\*Why java does not supports multiple Inheritance with classes

If we Assume java supports multiple inheritance

1)Compiler execute default constructor with super() call then compiler confused which super class is Executed.Then Ambiguity error will comes

That’s Why java does not supports multiple Inheritance with classes

Possible with interfaces only because interface cannot be instantiated

**publicclass** Super {

**public** Super() {

//constructor

}

}

**publicclass** Super1 {

**public** Super1() {

//constructor

}

}

**publicclass** Sub1 **extends** Super,Super1{

**public** Sub1() {

**super**();// here which super class will invoked by compiler. Then Ambiguity error will comes

}

}

2)**publicclass** Super {

**void** m1(){

Sysout(“Super Class”);

}

}

**publicclass** Super1 {

**void** m1(){

Sysout(“sub class Class”);

}

}

**publicclass** Sub1 **extends(Assume)**Super,Super1{//not possible but assume

p s v main(){

Sub1 s=new Sub1();

s.m1();// both super classes memory allocated by subclass then m1() called jvm get confused which method called so ambiguity error will comes

if you take Interface with multiple inheritance

Both interfaces has same method name with **same prototype**, yes its possible , subclass implements **only once**. And we can call by sub class object reference then it binds super class method and executed by sub class. that’s why java does not support multiple inheritance with classes its possible by only interfaces

}

Q) what is reference variable ?

The variable is created by using reference datatype is called reference variable

Reference variable is created by using Array , class, interface or enum.

-reference variable is pointing or referenced to object’s memory

*Types of reference variables*

**1)**Local reference variables-at the time of its method stack frame will get in JSA

2) static reference variable- at the time of class loading will get memory in HEAP AREA

3) non -static reference variable-at the time of object creation will get memory in METHOD AREA

*Types of Objects*

1)Referenced or Reachable Object

Example e=new Example();

2)un- referenaced or un reachable Onbjects

new Example();

-In java dev is responsible for creating object by using new and available constructor

-jvm is responsible to destroy the objects

Q) what type of Objects jvm destroy

Jvm destroy only unreferenced objects.

*Java.lang.OutOfMemoryError*

If there is no memory to create new object in heap area,jvm terminate program and throwing Error

Solution:Developer is responsible to unreferencing to objects that object after Using.

**Garbage Collection**

The process of destroying Objects unreferenced objects is called Garbage collection.

New Example();// unreferenced Object-destroy GC

Example e=new Example();// referenced objects-Destroy by Dev

Programmer should destroying referenced object By using Following ways

1)String s=null// storing null to destroy the object

2)s=new String();// storing assigning new objct to destroy

Q) How jvm Destroy the objects ?

Jvm internally use a daemon thread to destroy objects.

Daemon thread is service thread

Daemon thread is low priority thread since we cannot guarantee to this thread execution

We can request jvm to start garbage collection Process

System.gc(); or Runtime().getRunTime().gc();

**Arrays:**

In java Array is a reference data type .It is used store fixed number of multiple values and objects of same type Type in continue memory Locations.

Limitation : we cannot increase or decrease its size after creation

Array can allow only same type of data

*Array index starts with 0*

*int a[]=new int[3]*

*Here we pass 3 values in 3 locations those locations are a[0],[1],[2]*

Array Declaration:

**publicint**a[];

**publicstaticint***a1*[];

like c, c++ we cannot array size in declaration part

**publicint** [8]a;// CE

Two dimensional array declarartion:

**publicint** [][][]a;

**publicstaticint***a1*[][];

Array Object Creation:

*1)Array creation with values*

**int**a[]= {2,3,4,5,6,7,8};//primitive data

Customerc[]={**new**Customer(3),**new**Customer(5),**new**Customer(8)};//reference data

If we create array object with primitive data , so values stored directly in those locations

If we create array object with reference type data, so object reference stored directly in those locations

2) *Array creation without explicit values or with default values*

**int**a[]=**newint**[4];//primitive type

four memory locations with default values int 0

String s[]= **new** String[5];// reference type

five memory locations with default values string null

above only created one object . It creates one String Array object with five variables of type string with default value null

Customer c[]=**new** Customer[9];

Nine memory locations with default values customer null

Rules to create object:

1) array size is mandatory

String s[]= **new** String[];//CE

2) we must specify array size value in object creation side

Not declaration side

String s[2]= **new** String[];//CE

3) size should be zero or positive else CE

inta[]=newint[-4];// NegativeArraySizeException

4) while storing and reading and modifying value we must pass array index within range else

CE

**int**a[]=**newint**[7];

a[8]=4;//java.lang.ArrayIndexOutOfBoundsException: 8

5) Source data type and destination datat ype must match else CE

Customer c=**new**Example();//incompatible types

Array casting or Exception in array

For all reference types and array including primitive type java.lang.Object is super class

Array with object reference type super class is Object[]

Object[] is sub class of java.lang.Object

Object o=**newint**[] {1,2,3};

**int**[] some=(**int**[]) o;

**for**(**int**i:some) {

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

}

Op:123

*java.lang.ArrayStoreException:*

Object obj[]=**new** Customer[3] ;

obj[1]=**new** Customer(5);

obj[2]=**new** Test();//RE

*length property:*

length is non static final variable in every array object to store array size it is used find the length of array

**staticint***a*[]=**newint**[2];

**publicstaticvoid** main(String[] args) {

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

}

Op:2

Passing array as argument:

**publicstaticvoid** m1( Customer[] c){

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

}

*m1*(**new** Customer[4]);

Customer[] c=**new** Customer[3];

*m1*(c);

Customer[] c1= {**new** Customer(4),**new** Customer(8)};

*m1*(c1);

Test.*m1*(**new** Customer[] {**new** Customer(4),**new** Customer(6)});

// anonymous array

*instanceof :*

-instance of is a keyword to check whether it is current object instance or not.

-instanceof is a keyword that is used for checking if a reference variable is containing a given type of object reference or not.

-we use instance of keyword in if conditions only\

**publicclass** Test {

**publicstaticvoid** m1(Object o){

**if**(o**instanceof** Customer) {

System.***out***.println(" it is customer object");

}**else** {

System.***out***.println(" it is not a customer object");

}

}

**publicstaticvoid** main(String[] args) {

*m1*(**new** Object());

*m1*(**new** Customer(4));

*m1*(**new** Test());

}

}

Op:

it is not a customer object

it is customer object

it is not a customer object

Declaring array as final:

It is possible to declare array as final

Array is final not array variables

**finalint** [] a=**newint**[4];

Above array is a final not locations as final

**finalint** [] a=**newint**[4];

a[2]=9;

a[2]=8;

System.***out***.println(Arrays.*toString*(a));

//a=**newint**[5];//The final local variable a cannot be assigned

Types of arrays based on Dimensions

1)single dimension array: array of objects or values

**int** [] a=**newint**[4];

1)multi dimension array: arrays of arrays

**int**ia[][]=**newint**[3][3];

base array size is mandatory where as child array size is not mandatory

Var –arg parameter method

*Arary as parameter:*

**publicstaticvoid** m1(**int** [] a) {

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

System.***out***.println(a[i]);

}

}

**publicstaticvoid** main(String[] args) {

*m1*(**newint**[] {3,4,5,6,5});

}

*Var arg as parameter:*

**publicstaticvoid** m1(**int**... a) {

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

System.***out***.println(a[i]);

}

}

**publicstaticvoid** main(String[] args) {

*m1*(**newint**[] {3,4,5,6,5});

*m1*(9,5,3,3,3);

}*:*

what is the difference between next() and nextLine() in java

next() can read the input only till the space. It can’t read two words separated by space. Also, next() places the cursor in the same line after reading the input.

nextLine() reads input including space between the words (that is, it reads till the end of line n). Once the input is read, nextLine() positions the cursor in the next line.

**Some Example on Arrays:**

**1).**Addition with array

**publicclass** ExArray {

**publicstaticvoid**m1(**int**... a) {

**int**sum=0;

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

sum=sum+a[i];

}

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

}

**publicstaticvoid** main(String[] args) {

*m1*(**newint**[] {3,4,5,6,5});

*m1*(9,5,3,3,3);

}}

2) dynamic values in array:

**publicclass** ExArray {

**publicstaticvoid** m1(**int**... a) {

**int**sum=0;

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

sum=sum+a[i];

}

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

}

**publicstaticvoid** main(String[] args) {

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

System.***out***.println(" enter range ");

**int**range=sc.nextInt();

**int**a[]=**newint**[range];

**for**( **int**i=0;i<range;i++) {

System.***out***.println(" enter value "+(i+1));

**int**value=sc.nextInt();

a[i]=value;

}

*m1*(a);

}

3) string to array values

**publicclass** ExArray {

**publicstaticvoid** m1(**int**... a) {

**int**sum=0;

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

sum=sum+a[i];

}

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

}

**publicstaticvoid** main(String[] args) {

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

System.***out***.println(" enter numbers with spaces");

String s=sc.nextLine();

String[] split=s.split(" ");

**int**[] a=**newint**[split.length];

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

a[i]=Integer.*parseInt*(split[i]);

}

*m1*(a);

}

**Arrays class in java**

The **java.util.Arrays** class contains a static factory that allows arrays to be viewed as lists.Following are the important points about Arrays −

* This class contains various methods for manipulating arrays (such as sorting and searching).

**Some Methods in Arrays class**

*1) asList():*

The **java.util.Arrays.asList(T... a)** returns a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.) This method acts as bridge between array-based and collection-based APIs,

**Declaration**

Following is the declaration for **java.util.Arrays.asList()** method

public static <T> List<T> asList(T... a)

**publicclass** Ex {

**publicstaticvoid** main(String[] args) {

String [] s= {"hari","naren"};

Listl=Arrays.*asList*(s);

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

Listl1=Arrays.*asList*("hanuman","salmon","ram");

System.***out***.println(l1);

}

}

Op:

[hari, naren]

[hanuman, salmon, ram]

## *2)binarySearch():*

## Description

The **java.util.Arrays.binarySearch(byte[] a, byte key)** method searches the specified array of bytes for the specified value using the binary search algorithm.The array must be sorted before making this call.If it is not sorted, the results are undefined.

## Declaration

Following is the declaration for **java.util.Arrays.binarySearch()** method

public static int binarySearch(Type a, Type key)

**publicstaticvoid** main(String[] args) {

**int** [] a= {3,2,4,5,6};

**int**index=Arrays.*binarySearch*(a,4);

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

}

Op:2

*3)copyOfRange():* copy the original array to new array

**publicstaticvoid** main(String[] args) {

**int** [] a= {3,2,4,5,6};

System.***out***.println("original array");

**for**(**int**d:a) {

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

}

**int**s[]=Arrays.*copyOf*(a,3);

System.***out***.println("new array");

**for**(**int**t:s) {

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

}

}

}

Op:

original array

3

2

4

5

6

new array

3

2

4

*4)*[*static boolean deepEquals(Object[] a1, Object[] a2)*](https://www.tutorialspoint.com/java/util/arrays_deepequals.htm)

This method returns true if the two specified arrays are deeply equal to one another.

**publicstaticvoid** main(String[] args) {

**int** a[] = {1,2,3};

**int** b[]= {1,2};

System.***out***.println(Arrays.*equals*(a,b));

String s[]= {"naren","siva"};

String s1[]={"naren","siva"};

System.***out***.println(Arrays.*deepEquals*(s1,s));

}

*5)*[*static int deepHashCode(Object[] a)*](https://www.tutorialspoint.com/java/util/arrays_deephashcode.htm)

This method returns a hash code based on the "deep contents" of the specified array.

method returns a hash code based on the "deep contents" of the specified array.For any two arrays a and b such that Arrays.deepEquals(a, b), it is also the case that Arrays.deepHashCode(a) == Arrays.deepHashCode(b).

**publicstaticvoid** main(String[] args) {

String s[]= {"naren","siva"};

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

Op:-1049150918 }

6) [static boolean equals(Object[] a, Object[] a2)](https://www.tutorialspoint.com/java/util/arrays_equals_object.htm)

This method returns true if the two specified arrays of Objects are equal to one another.

**publicstaticvoid** main(String[] args) {

**int** a[] = {1,2,3};

**int** b[]= {1,2};

**int**c[]= {1,2,3};

System.***out***.println(Arrays.*equals*(a,b));

System.***out***.println(Arrays.*equals*(a,c));

}

Op:

false

true

7)[static void sort(Object[] a)](https://www.tutorialspoint.com/java/util/arrays_sort_object.htm)

This method sorts the specified array of objects into ascending order, according to the natural ordering of its elements.

**publicstaticvoid** main(String[] args) {

**int**a[] = {1,2,6,17,9};

System.***out***.println("before sort"+Arrays.*toString*(a));

Arrays.*sort*(a);

System.***out***.println("After sort"+Arrays.*toString*(a));

}

Op:

before sort[1, 2, 6, 17, 9]

After sort[1, 2, 6, 9, 17]

8)[static String toString(Object[] a)](https://www.tutorialspoint.com/java/util/arrays_tostring_object.htm)

This method returns a string representation of the contents of the specified array of ints.

**publicstaticvoid** main(String[] args) {

**int**a[] = {1,2,6,17,9};

System.***out***.println(Arrays.*toString*(a));

}

Op:[1, 2, 6, 17, 9]

For more About Arraysclass:<https://www.tutorialspoint.com/java/util/java_util_arrays.htm>

**INNER CLASSES**

The class, interface or enum and annotation are defined in another class is called inner classes.

**publicclass** Example {

**class** sample{

}

}

**publicclass** Example {

**interface** sample{

}

}

**publicinterface** Example {

**interface** sample{

}

}

**publicinterface** Example {

**class** sample{

}

}

We have 4 types of inner classes:

1).Nested classes(static inner classes)

2) inner classes (non static inner classes)

3)method local class(local inner classes)

4) anonymous inner classes(Argument inner classes)

**publicclass** Example {

**staticclass** sample{//static inner class

}

**class** B{

//non static inner class

}

**void** m1() {

**class**C{

//method inner class

}

**new** Thread() {

//argument inner class

};

}

}

Here class files generated like below

Example$Sample.class

Example$B.class

Example$1C.class

Example$1.class

*1).Nested classes(static inner classes):*

Inner class defined with static keyword is called static inner class

* All types of8 members allowed
* Accessing outer class members from inner class
* **publicclass** StaticInnerclassEx {
* **staticint***a*=10;
* **int**s=20;
* **privateint**y=85;
* **staticclass** Inner{
* **publicstaticvoid** main(String[] args) {
* System.***out***.println(*a*);
* //System.out.println(s);//cannot make a static reference to the non-static field s
* StaticInnerclassEx e= **new** StaticInnerclassEx();
* System.***out***.println(e.s);
* System.***out***.println(e.y);
* }
* }}
* Accessing inner class members from outer class
* **publicclass** Example1 {
* **staticclass** A{
* **staticint***a*=5;
* **int**b=4;
* **privateint**c=1;
* }
* **publicstaticvoid** main(String[] args) {
* // **TODO** Auto-generated method stub
* System.***out***.println(A.*a*);
* A a =**new** A();
* System.***out***.println(a.b);
* System.***out***.println(a.c);
* }
* }
* Accessing outer side class membersExample1.A a=**new** Example1.A();
* **publicclass** Out {
* **publicstaticvoid** main(String[] args) {
* System.***out***.println(Example1.A.*a*);
* Example1.A a=**new** Example1.A();
* System.***out***.println(a.b);
* }
* *}*

*2)Non Static inner class*

**publicclass** Ex {

**class** B{

}

}

By above 2 .class files are generated

-only non static members allowed

-we should not declare static members in non static content

-we cannot create main method in non static because main is static

-creating object to the non static inner class is

**publicclass** Ex {

**class** B{

**privateint**id;

String name;

}

**publicstaticvoid** main(String[] args) {

Ex.B b=**new** Ex().**new** B();

}}/

*3)Method Local inner class*

The inner class is defined in a method of outer class

**publicclass** Ex {

**publicvoid** m() {

**class**A{

}

}

}

By above 2 .class files are generated

Ex.class

Ex$1A.class

--only non static members allowed

-if you create object to the method inner class is not possible

**publicclass** Ex {

**publicvoid** m() {

**class**A {

**void**m2() {

System.***out***.println("how can we reach");

}

}

}

**publicstaticvoid** main(String[] args) {

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

Ex e =**new** Ex();

e.m();

//A a=new A(); // we can not create object

}

}

Solution of above -then we take special class is super class that is interface with a method name of our inner class .

**interface** Ex1{

**void** m2();

}

**publicclass** Ex {

**public** Ex1 m() {

**class** A **implements** Ex1{

**publicvoid** m2() {

System.***out***.println("how can we reach");

}

}

A a=**new** A();

**return** (Ex1) a;

}

**publicstaticvoid** main(String[] args) {

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

Ex e =**new** Ex();

e.m();

Ex1 ex=e.m(); //create object with m()

ex.m2();//run time polymorphism then it is executed in the sub class

}

}

*4)Anonymous inner class:*

It is sub class of some other existed class or interface.

**publicinterface** A {

}

**publicclass** Ex {

A a=**new** A(){

};

}

We no need to create any object compiler will call

**package** Com.nare.nonstatic;

**publicclass** Ex {

**static** A *a*=**new** A(){

@Override

**publicvoid** m1() {

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

}

};

**publicstaticvoid** main(String[] args) {

*a*.m1();

}

}

Understanding of enum:

Enum is set of constants those menu kind of items.

Enum is internally final class.

We should use enum in below situations .

1. Creating course list .

2. Hotel menu

3. Colours list

4. Months etc.

Before enum( java5) we create static final variables.

**publicclass** Ex {

**staticfinalint*jan***=1;

**staticfinalint*feb***=2;

}

**publicclass** Ex1 {

**publicstaticvoid** main(String[] args) {

System.***out***.println(Ex.***jan***);

System.***out***.println(Ex.***feb***);

}

}

OP:

1

2

Ifwe want get the constant we can use enum.

**publicenum** Ex {

***jan***, ***feb***,***march***;

}

**publicclass** Ex1 {

**publicstaticvoid** main(String[] args) {

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

System.***out***.println(Ex.***jan***);

System.***out***.println(Ex.***feb***);

}

}

Op:

jan

feb

Fundamental Classes:

1) Object : object is super class for all Classes.

2) Class : jvm create this class object and store all the class names.

3) String ,String Buffer ,String Builder : these classes to represent character Strings.

4) Throwble, Exception ,Error : Responsible to handle run time errors.

5) Wrapper Classes : Converting primitive data to object and vice versa.

6) Thread : are responsible to create user defined threads.

7) System , Runtime, ClassLoader : these classes responsible in providing system operations.

Object:

-Object is super class for all predefined and userdefined Classes.

- By using Object class can achieve reusability and loose coupling and run time polymorphism.

-Every class has a 11 common properties those are available in object class.

Class {@code Object} is the root of the class hierarchy.

\* Every class has {@code Object} as a superclass. All objects,

\* including arrays, implement the methods of this class.

1.\* Returns the runtime class of this {@code Object}

**publicfinalnative** Class<?> getClass();

2. Returns a hash code value for the object

**publicnativeint** hashCode();

3.Indicates whether some other object is "equal to" this one.

**publicboolean** equals(Object obj) {

**return** (**this** == obj);

}

4. Creates and returns a copy of this object.

**protectednative** Object clone() **throws**

CloneNotSupportedException;

5.Returns a string representation of the object

**public** String toString() {

**return** getClass().getName() + "@" +

Integer.*toHexString*(hashCode());

}

6. Wakes up a single thread that is waiting on this object's

**publicfinalnativevoid** notify();

7. Wakes up all threads that are waiting on this object's monitor

**publicfinalnativevoid** notifyAll();

8.Causes the current thread to wait until either another thread invokes

**publicfinalnativevoid** wait(**long**timeout) **throws**

InterruptedException;

9.Causes the current thread to wait until either another thread invokes

**publicfinalvoid** wait() **throws** InterruptedException {

wait(0);

}

10.Causes the current thread to wait until either another thread invokes

**publicfinalvoid** wait(**long**timeout, **int**nanos) **throws** InterruptedException

11. Called by the garbage collector on an object when garbage collection

\* determines that there are no more references to the object.system resources or to perform other cleanup.

**protectedvoid** finalize() **throws** Throwable { }

1) Retrieving a hash code value for the object

**publicnativeint** hashCode();

-every object have a unique identity . ie 32 bit unique number.

-hash code id used only Set ,Map collections to storing , searching objects of a class.

Hash code can generate in two ways:

1) By using its reference : generated jvm every object hash code is different.

**publicclass** HashCode {

**publicstaticvoid** main(String[] args) {

HashCode h=**new** HashCode();

HashCode j=**new** HashCode();

System.***out***.println(h.hashCode());//1829164700

System.***out***.println(j.hashCode());//2018699554

}

}

2) By using its state : overriding hash code create our custom code.

It is same to every object.

**publicclass** HashCode {

**int**x=99;

**publicint** hashCode() {

**return**x;

}

**publicstaticvoid** main(String[] args) {

HashCode h=**new** HashCode();

HashCode j=**new** HashCode();

System.***out***.println(h.hashCode());//99

System.***out***.println(j.hashCode());//99

}

}

2. Indicates whether some other object is "equal to" this one.

**publicboolean** equals(Object obj) {

**return** (**this** == obj);

}

Comparing two objects of a class.

Comparing objects in two ways

1) By using its reference

By using ‘==’ compare will always state

By using equals compare will state and reference.

**publicclass** Equals {

**publicstaticvoid** main(String[] args) {

**int**x=10;

**int**y=10;

Equals e =**new** Equals();

Equals e1 =**new** Equals();

System.***out***.println(e.equals(e1));//false

System.***out***.println(e==e1); // false

}

}

By defaults equals () will works on reference if we want to state we can ovwerride equals()

2) By using its state

If we want compare objects with state we can override equals()

If we overrides equal () we can check reference and state.

**package** Equals;

**publicclass** Student {

String name;

**int**no;

**public** Student(String name, **int**no) {

**super**();

**this**.name = name;

**this**.no = no;

}

@Override

**publicboolean** equals(Object obj) {

**if**(**this** == obj) {

**returntrue**; // comparing objects referece

}**else** {

**if**(obj**instanceof** Student) {

Student s=(Student)obj;

**return** (**this**.no==s.no&&**this**.name==s.name);

// comparing objects state

}

**else** {

**returnfalse**;

}

}

}

**publicstaticvoid** main(String[] args) {

Student s=**new** Student("naren",7);

Student s2=s;

System.***out***.println(s.equals(s2));//true// its reference

Student ss = **new** Student("har",4);

System.***out***.println(s.equals(ss));// false it comes from state

}

}

3. \* Returns the runtime class of this {@code Object}

**finalnative** Class<?> getClass();

- Retrieving run time class object reference

- The class is loaded in jvm at execution time is called run time class.

**publicclass** GetName {

**publicstaticvoid** main(String[] args) {

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

GetName g=**new** GetName();

System.***out***.println(g.getClass());

System.***out***.println(g.getClass().getName());

}

}

Op:

class GetName

GetName

4. Returns a string representation of the object

**public** String toString() {

**return** getClass().getName() + "@" +

Integer.*toHexString*(hashCode());

}

- it generates classname@hexadecimal of hash code.

- when we print object println method calls toString() internally.

-we override the toString to generate object state in string format.

**publicclass** ToString {

**int** x=0;

String s;

**public** ToString(**int** x,String s) {

**this**.x=x;

**this**.s=s;

}

**publicstaticvoid** main(String[] args) {

String s="narendra";

ToString s1=**new** ToString(8," narendra");

System.out.println(s1);

}

}

Op:

ToString@6d06d69c

**publicclass** ToString {

**int**x=0;

String s;

**public** ToString(**int**x,String s) {

**this**.x=x;

**this**.s=s;

}

**public** String toString(){

**return**"no "+x+" "+

"name "+s;

}

**publicstaticvoid** main(String[] args) {

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

String s="narendra";

ToString s1=**new** ToString(8," narendra");

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

}

}

Op:no 8 name narendra

5. Creates and returns a copy of this object.

**protectednative** Object clone() **throws**

CloneNotSupportedException;

* Clone is used to creating duplicate copy with current object state .
* To execute clone() we must implement cloneable interface

-A class implements the Cloneable interface toindicate to the method that it

is legal for that method to make afield-for-field copy of instances of that class.

**publicclass** Clone **implements** Cloneable {

**int**x;

@Override

**public** String toString() {

**return** Integer.*toBinaryString*(x);

}

**publicstaticvoid** main(String[] args) **throws** CloneNotSupportedException {

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

Clone c=**new** Clone();

c.x=30;

Clone c1=(Clone)c.clone();

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

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

}

}

Op:

11110

11110

11. finalize():Called by the garbage collector on an object when garbage collection

determines that there are no more references to the object.

system resources or to perform other cleanup.

**protectedvoid** finalize() **throws** Throwable { }

-destructing all parts of the object.

-garbage collector is responsible for destroying objects

-finalise method automatically called by garbage collector.

**class**F

{ **double**c=1000;}

**publicclass** Finalize {

Ff;

Finalize(Ff){

**this**.f=f;

}

**publicvoid** finalize() {

System.***out***.println("in finalize method");

f=**null**;

}

}

**publicclass** FinalizeDemo{

**publicstaticvoid** main(String[] args) **throws** InterruptedException {

**for**(**int**i=1;i<=10;i++) {

**new** Finalize(**new** F());

}

System.*gc*();

Thread.*sleep*(1000);

}

final & finally & finalisze:

final : final member cannot be changed.

Final class cannot overridden ,variable cannot changed ,

finalize(): finalize is used destroying the objects.

Finalize() calls by gc internally.

finally {}: is executed definitely whether exception is handled or not.

Finally is mostly used to un referencing the objects

*The remaining methods will discuss in the multithreading*

String Handling

String:

* String is sequence a of characters placed inside “ “
* “ “ is called String literal
* “ “ internally creates new object of java.lang.String
* String internally uses char[] object.
* Strings="narendra"; here s is not a object “narendra” is object. S is variable.

*What are the possible ways to create a string ?*

String object crete by string literal and one of String constructor

String class have 15 constructors .among them we discuss some

**publicclass** CreteStringPossibleWays {

**publicstaticvoid** main(String[] args) {

//1)String()

String s1= **new** String();

//2)String(String original)

String s2 = **new** String(" narendra dasara");

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

//3)String(char value[])

**char**v[]= {'n','a','r','e','n','d','r','a'};

String s3= **new** String(v);

System.***out***.println(s3);

//4) String(StringBuffer buffer)

StringBuffer b =**new** StringBuffer(" narendra");

String s4 =**new** String(b);

System.***out***.println(s4);

//5)String(StringBuilder builder)

StringBuilder b1= **new** StringBuilder("narendra");

String s5= **new** String(b1);

System.***out***.println(s5);

//6) public String(char value[], int offset, int count) // create new strig of given char

**char**v1[]= {'n','a','r','e','n','d','r','a'};

String s6= **new** String(v1,2,6);

System.***out***.println(s6);//rendra

//7)String(byte bytes[], int offset, int length)

**byte**byt[]= {'n','a','r','e','n','d','r','a'};

String s7= **new** String(byt,1,7);

System.***out***.println(s7);//arendra

//8)String(byte bytes[])

**byte**by[]= {'n','a','r','e','n','d','r','a'};

String s8= **new** String(by);

System.***out***.println(s8);//narendra

}

}

String Operations:

<https://github.com/narendradasara99/StringHandling>

what is the difference b/w split and StringTokenizer..?

String.**split**( ) VS **stringTokenizer**. For one thing, **StringTokenizer** returns one substring at a time whereas the **split** method returns an array of substrings.

Best to use String's split(...) method. The StringTokenizer class is a legacy class (NOT deprecated).

By default, StringTokenizer uses whitespace to delimit tokensand if there are two or more delimiter characters, in a row, it collapses them into one delimiter. See the difference in below example.

**package** com;

**import**java.util.Arrays;

**import** java.util.StringTokenizer;

**publicclass** StringTokenizerEx {

**publicstaticvoid** main(String[] args) {

StringTokenizer s = **new** StringTokenizer(" i am naren”,"am");

//delimiting where available a and m as taken separate tokens

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

**while**(s.hasMoreElements()) {

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

}

String s1= **new** String(" i am naren");

String []a=s1.split("am");// split is delimiting as a one string

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

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

System.***out***.println(a[i]);

}

}

}

Op:

3

i

n

ren

2

i

naren

StringBuffer:

StringBuffer is thread safe , mutable of sequence of characters . StringBuffer is same like String but can be modified in the same memory location.

Where as StringBuilder is also same but it is not thread safe .

StringBuffer s=**new** StringBuffer();//create empty string with defaultsize is 16

StringBuffer s=**new** StringBuffer(8);//create empty string with size is 8

StringBuffer s=**new** StringBuffer("abc");// converting string to string buffer object with default size is 16.

***StringBufferOperation: stingbuffer is mutable we can perform operations***

1)append() : adding the given String

2) insert() : insert the data in given location.

3)replace() : replacing the data of given locations.

4) delete ():deleting the data of given locations.

5) reverse(): reversing the whole string.

**publicclass** StBuff {

**publicstaticvoid** main(String[] args) {

StringBuffer s=**new** StringBuffer("abc");

s.append(" is Sequence of characters ");

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

s.insert( 2,"efghjklmnopqrstuvwxyz");

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

s.replace(2, 6, " narendra is good boy");

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

s.delete(2, 9);

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

s.reverse();

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

}

}

Op:

abc is Sequence of characters

abefghjklmnopqrstuvwxyzc is Sequence of characters

ab narendra is good boyjklmnopqrstuvwxyzc is Sequence of characters

abra is good boyjklmnopqrstuvwxyzc is Sequence of characters

sretcarahc fo ecneuqeS si czyxwvutsrqponmlkjyob doog si arba

\*\*Why String is immutable...?

1). End user send data in the form String. If string is mutable .if you want to concat the some string to the original value then original value will be gone from the memory . next you want do another operation on original value original value gone from the memory location on the first operation its not possible then string is immutable.

2) In the Map collection we store the value in the form of key value pair .

Key is string, if string is mutable then after storing the value in the Map,

Key will be changed with any operation ,in the map also changed then we get the value of map we get wrong answers that’s why string is immutable.

**package** com;

**import** java.util.HashMap;

**import** java.util.Map;

**publicclass** WhyStringisimmutable {

**publicstaticvoid** main(String[] args) {

Mapm=**new** HashMap<>();

String s= **new** String("naren");

m.put(s,8);

s.toUpperCase();

System.***out***.println(m.get("naren"));

StringBuffer s1=**new** StringBuffer("d");

m.put(s1,10);

s1.append("narendra");

System.***out***.println(m.get("d"));

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

}

}

Op:

8

null

**Exception Handling:**

Exception handling mechanism is used to solving runtime errors due to the program execution time.

Throwable class is super class to all errors. They are two types

1)errors : we cannot catch error type exception. Those are handled by jvm

2) exception :we can catch error type exception.Those are handled by programmer.

Exception have a subclasses those are

**publicclass** Exception **extends** Throwable

Runtime exception sub classes : those are logical mistake due program execution time.

**publicclass** RuntimeException **extends** Exception

and direct exception type sub classes :are logical mistake due program execution time due to

condition failed by given wrong input values. These exceptions are mandatory to handle by using throw keyword other wise throws error

Based on exception handling all exception are divided into two categories

1) checked exceptions : when exception is handled by programmer if that exception is checked by compiler is called checked exception .

Ex: all direct sub classes of exception class.

ClassNotFoundException

CloneNotSupportedException

InterruptedException

NoSuchFieldException

NoSuchMethodException..... etc

2) un checked exceptions: when exception is handled by programmer if that exception is not checked by compiler is called un checked exception .

Ex: Error and RunTimeExceptions

ArithmeticException

ArrayIndexOutOfBoundsException

NullPointerException

ClassCastException ... etc

How can we handled exceptions ..?

1) by using throws for reporting exception to method handler

2) using try,catch for catching an exception to stop abnormal condions.

Try : may or may not exception raised statements we can write.

Catch: catching the exception send user understanble message we can write.

Finally: compulsory executable code when control come out from try block.

By using finally block to write return statement we can suppress the exception:

**publicclass** Ex1 {

**publicstaticint** m() {

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

**try** {

System.***out***.println(10/0);

}

**catch**(NullPointerException e) {

System.***out***.println(" its not reached");

}

**finally**{

**return** 5;

}

}

**publicstaticvoid** main(String[] args) {

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

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

}

}

Exception Propagation: the process of sending an exception from called method to caller method is called Exception Propagation.

We can handle exception either called or caller method

public class Ex1 {

public static void m() {

// TODO Auto-generated method stub

try {

System.out.println(10/0);

}

**catch**(ArithmeticException e) {

System.***out***.println(" by /zero is wrong");

}

}

**publicstaticvoid** main(String[] args) {

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

*m*();

}}

**publicclass** Ex1 {

**publicstaticvoid** m() {

System.***out***.println(10/0);

}

**publicstaticvoid** main(String[] args) {

**try** {

*m*();

}

**catch**(ArithmeticException e) {

System.***out***.println(" arthimetic exception");

}

}}

Throw: throw is used to throw an exception by manually from a method or constructor.

In most of the cases we use it from throwing checked excetion.

Throws:throw is used to report the exception to method caller.

Multithreading:

Thread : A thread is sequential flow of execution path .it executes methods in sequential flow( one after another) . in single jvm to create multiple threads for multiple independent flows of execution paths at a time concurrently.

Mutlithreading: the process of creating multiple threads for executing multiple independent tasks concurrently to complete their execution in less time by using cpu ideal time.

Inside jvm run two threads they are

1)main

2) Garbage collector

We can create custom thread in two ways

1. By extending java.lang.Thread class
2. By implementing java.lang.Runnable interface
3. By using anonymous inner class
4. By implementing Callable interface.

**Thread Class vs Runnable Interface**  
  
1. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.  
  
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

1)By extending java.lang.Thread class

**publicclass** ByUsingThread **extends** Thread {

@Override

**publicvoid** run() {

System.***out***.println("by using thread class");

}

**publicstaticvoid** main(String[] args) {

Thread b=**new** ByUsingThread();

b.start();

}

}

2) By implementing java.lang.Runnable interface

**publicclass** ByUsingRunnableInterface **implements** Runnable {

@Override

**publicvoid** run() {

System.***out***.println("by using Runable inteface");

}

**publicstaticvoid** main(String[] args) {

ByUsingRunnableInterface b=**new** ByUsingRunnableInterface();

Thread t=**new** Thread(b);

t.start();

}

}

3) By using anonymous inner class

**publicclass** ByUsingAnonymousclass {

**publicstaticvoid** main(String[] args) {

//1)using anonymous class by using runnable

Runnable r=**new** Runnable() {

@Override

**publicvoid** run() {

System.***out***.println(" by using anonymous class by using runnable");

}

};

Thread t = **new** Thread(r);

t.start();

//2) by using anonymous class using thead

**new** Thread() {

**publicvoid** run() {

System.***out***.println(" by using anonymous class using thead ");

};

}.start();

}

}

4)By implementing Callable interface.

What is callable interface in Java with example?

**Callable interface** was added in **Java** 5 to complement existing Runnable **interface**, which is used to wrap a task and pass it to a Thread or thread pool for asynchronous execution. **Callable** actually represent an asynchronous computation, whose value is available via Future object.

To create the thread, a Runnable is required. To obtain the result, a Future is required.

The Java library has the concrete type FutureTask, which implements Runnable and Future, combining both functionality conveniently.  
A FutureTask can be created by providing its constructor with a Callable. Then the FutureTask object is provided to the constructor of Thread to create the Thread object. Thus, indirectly, the thread is created with a Callable. For further emphasis, note that there is no way to create the thread directly with a Callable.

Link: <https://www.geeksforgeeks.org/callable-future-java/>

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutionException;

**import**java.util.concurrent.FutureTask;

**publicclass** ByusingCallableInterface **implements** Callable<Employee>{

@Override

**public** Employee call() **throws** Exception {

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

Employee e= **new** Employee("narendra");

**return**e;

}

**publicstaticvoid** main(String[] args) **throws** InterruptedException, ExecutionException {

Callable<Employee>e=**new** ByusingCallableInterface();

FutureTask<Employee>f=**new**FutureTask<>(e);

Thread t =**new** Thread(f);

t.start();

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

}

}

Thread priority:

Every thread is created with a priority. It is b/w 1 to 10.

1🡪 is min priority

5🡪is normal priority

10🡪is max priority

Set and get priority, name:

If we set highest priority to the thread thenit will completed first.

**publicclass** PriorityEx **extends** Thread {

@Override

**publicvoid** run() {

System.***out***.println("run"+getName());

}

**publicstaticvoid** main(String[] args) **throws** InterruptedException {

Thread t=**new** PriorityEx();

t.setPriority(7);

t.setName("lowest priority thread");

t.start();

Thread t1=**new** PriorityEx();

t1.setName("highest priority thread");

t1.setPriority(10);

t1.start();

}

}

Op:

run lowest priority thread

run highest priority thread

Types of threads

1)non deamon:a thread that executes main logic of the program is called daemon thread

2) daemon:a thread that executes background based on scheduled the program is called daemon thread. It is service thread. Garbage collector is daemon thread.`

Synchronization:

The process of executing multiple threads using same objects is called synchronization .when multiple threads are modifying same object we must develop synchronization

Dead lock: deadlock is situation if there are two threads are waiting to each other to complete theirexecution. Wrong usage of synchronization keyword then dead lock is occurred.

Inner thread communication : the process of executing multiple threads that threads communicating each other internally and completing their execution is called inter thread communication. Below methods to develop inner thread communication

wait(): the thread tells to another thread this is in waiting sate

notify(): notify method is notice this thread mode is convert waiting to wait for lock.

notifyall(): same like notify.

Collection:

Collection is a frame work is used to store heterogeneous and homogeneous , unique and duplicate objects without worrying about the size limitation for carrying multiple objects at a time for one application to another application.

\*) how many ways can we collect objects?

1) Array format🡪 without identity

2) key and value pair format🡪with identity

In java 1.0 two classes we have to collect objects in above formats

1) Vector 🡪 Array format

2) Hashtable🡪key and value format

These two classes are thread safe to solve this problem in java 1.2 introduced more classes...

Those are ArrayList and HashMap respectively.

***Types of collection hierarchy:***

1) Collection hierarchy classes-Array format

2)Map hierarchy classes- key and value format

***Common terminologies for collection framework***

*Collection of objects:* the collection object contains other class objects.

*Collection of Collectoin:*the collection object contains other Collection objects.

*Collection of Maps:*the collection object contains other Map objects.

*Element :* means object

*Entry:*  means (key value)pair

*Homogeneous elements*: same class objects

*Heterogeneous elements*: different class objects

*Unique elements:*different class objects or same class objects with different state/reference

*Duplicate elements :*same class objects with same state/reference

Objects are decided unique or duplicate by ‘==’ operator and equals()

== returns true then objects are duplicate

== and equals return false then objects are unique

***Collection Hierarchy****:*

Collection classes are divided into categories 🡪 Set, List and Queue

Set implemented classes:- HashSet, TreeSet, LinkedHashSet.

List implemented classes:- ArrayList, Vector ,LinkedList, Stack.

Queue implemented classes:- PriorityQueue.

***Set implemented classes*** *:* for storing only unique elements (different class objects or same class objects with different state/reference)

***HashSet:***  for storing heterogeneous unique elements without worrying sorting order

***LinkedHashSet:***for storing heterogeneous unique elements with insertion order

***TreeSet:*** for storing homogeneous unique elements with sorting order. TreeSet only allows homogeneous unique elements.TreeSet not allowed heterogeneous unique elements because they cannot be compared.

***List implemented classes:***  for storing unique and/or different elements in insertion order we must choose list classes.

***ArrayList and Vector:***  for storing and retrieve duplicate in random access(insertion and deletion in middle),these two classes functionality is same except ,Vector is Thread safe. Single thread model we must choose ArrayList.

The draw back of ArrayList and Vector gives less performance if we perform insertion and deletion in the middle all remaimg elements moved side .

***LinkedList:*** to solve above problem we must choose LinkedList . It internally uses linked list data structure.

***Stack:*** for storing and retrieveing in LIFO order we choose Stack.

***Map Hierarchy***

Map hierarchy classes collect the values in the form of key value pair

***HashMap***: for storing unique entries we must choose HashMap

***LinkedHashMap***:for storing unique entries in insertion order we must choose LinkedHashMap.

***TreeMap***:storing unique entries in sorting order we must choose HashMap ,TreeMap Allows only comaparable unique entries

***Hashtable*** : HashMap and hashTable is same but hashTable is thread safe

***Properties***: class is used to store properties permanently in file.

***HashSet:***  for storing heterogeneous unique elements without worrying sorting order.

* Duplicates not allowed
* HashMap is backed to HashSet
* It is a growable array or incremental array
* Default capacity is 16
* Incremental capacity is double
* Load factor 0.75( when array size reached by 0.75.. array size will be increased)
* Unique elements only added.
* Null insertion is possible, only one allowed

**import**java.util.HashSet;

**publicclass** HashSetEx {

**publicstaticvoid** main(String[] args) {

HashSeth =**new**HashSet<>();

h.add(**new** Employee("narendra"));

h.add(**null**);

h.add(**null**);// only one null is allowed

h.add(**new** Employee("narendra"));

h.add(80);

h.add("narendra");

h.add("narendra");//duplictes not allowed

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

}

}

Op:[null, Employee [name=narendra], 80, narendra, Employee [name=narendra]]

***LinkedHashSet:***for storing heterogeneous unique elements with insertion order.

* Duplicates not allowed
* LinkedHashMap is backed to HashSet
* It is a growable array or incremental array
* Default capacity is 16
* Incremental capacity is double
* Load factor 0.75( when array size reached by 0.75.. array size will be increased)
* Unique elements only added.
* Null insertion is possible, only one allowed

**import**java.util.HashSet;

**import** java.util.LinkedHashSet;

**publicclass** HashSetEx {

**publicstaticvoid** main(String[] args) {

LinkedHashSeth=**new** LinkedHashSet<>();

h.add(**new** Employee("narendra"));

h.add(**null**);

h.add(**null**);// only one null is allowed

h.add(**new** Employee("narendra"));

h.add(80);

h.add("narendra");

h.add("narendra");//duplictes not allowed

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

}

}

Op:[Employee [name=narendra], null, Employee [name=narendra], 80, narendra]

***TreeSet:*** for storing homogeneous unique elements with sorting order

* Duplicates not allowed
* HashMap is backed to TreeSet
* It is a growable array or incremental array
* Default capacity is 16
* Incremental capacity is double
* Load factor 0.75( when array size reached by 0.75.. array size will be increased)
* Homogeneous Comparable Unique elements only added.
* Null insertion is not possible, only one allowed

**import**java.util.HashSet;

**import**java.util.LinkedHashSet;

**import** java.util.TreeSet;

**publicclass** HashSetEx {

**publicstaticvoid** main(String[] args) {

TreeSeth =**new** TreeSet<>();

h.add("narendra");

h.add("narendra");// duplicates not allowed

h.add("narez");

//h.add(5); only comparable objects allowed

//h.add(null); null not allowed

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

}

}

Op:[narendra, narez]

***TreeSet, Comparable&Comparator:***

Treeset add elements in ascending and descending order

* Tree set does not have any sorting order, we can define in our class based on our requirement.
* TreeSet for storing elements in sorting order internally uses two interfaces Comparable& Comparator
* Treeset add() method call internally these interface methods.
* Our class must be implement by one of these interfaces otherwise ce

Exception in thread "main" java.lang.ClassCastException: Employee cannot be cast to java.lang.Comparable

at java.util.TreeMap.compare(Unknown Source)

at java.util.TreeMap.put(Unknown Source)

at java.util.TreeSet.add(Unknown Source)

at HashSetEx.main(HashSetEx.java:8)

* Add () method store the objects based on the return type values
* If returns 0🡪 this object is not added
* If returns -ve🡪 this object is added left to treeset element.
* If returns +ve🡪 this object is added Right to treeset element.
* Treemap put method logic

cmp = cpr.compare(key, t.key);

**if** (cmp< 0)

t = t.left;

**elseif** (cmp> 0)

t = t.right;

**else**

***return****t.setValue(value);*

*Comparable compareTo():*

Implement compareTo() in our class

**publicclass** Employee **implements**Comparable {

**private** String name;

**public** Employee(String name) {

**super**();

**this**.name = name;

}

**public** String getName() {

**return**name;

}

**publicvoid** setName(String name) {

**this**.name = name;

}

@Override

**public** String toString() {

**return**"Employee [name=" + name + "]";

}

@Override

**publicint** compareTo(Object o) {

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

**return** write logic if u want;

}

}

import java.util.HashSet;

import java.util.LinkedHashSet;

import java.util.TreeMap;

import java.util.TreeSet;

public class HashSetEx {

public static void main(String[] args) {

TreeSet h =new TreeSet<>();

h.add(new Employee("znarendra"));

h.add(new Employee("narendr"));

h.add(new Employee("nare"));

h.add(new Employee("nare"));

System.out.println(h);

}

}

We have 5 test cases to implement compareTo()

1) if return 0

@Override

**publicint** compareTo(Object o) {

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

**return** 0;

}

Op:[Employee [name=znarendra]]

Here first element only stored.

2)if return -ve

**publicint** compareTo(Object o) {

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

**return** -1;

}

Op:[Employee [name=nare], Employee [name=nare], Employee [name=narendr], Employee [name=znarendra]]

Here all elements added in insertion order including duplicates

3) if return +ve

**publicint** compareTo(Object o) {

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

**return** 4;

}

Op:[Employee [name=znarendra], Employee [name=narendr], Employee [name=nare], Employee [name=nare]]

Here all elements added in reverse insertion order including duplicates

4) if return (arg.field-this.field)

**publicint** compareTo(Object o) {

// **TODO** Auto-generated method stubm

**if**(o**instanceof** Employee) {

Employee e=(Employee) o;

**returnthis**.name.compareTo(e.name);

}

**elsereturn** 0;

}

Op:[Employee [name=nare], Employee [name=narendr], Employee [name=znarendra]]

Here unique+

e elements added in ascending order

4) if return (this.field-arg.field)

@Override

**publicint** compareTo(Object o) {

// **TODO** Auto-generated method stubm

**if**(o**instanceof** Employee) {

Employee e=(Employee) o;

**return**e.name.compareTo(**this**.name);

}

**elsereturn** 0;

}

Op:[Employee [name=znarendra], Employee [name=narendr], Employee [name=nare]]

Here unique elements added in descending order

*Comparator compare():*

Comparator is used for custom sorting order.

**import** java.util.Comparator;

**publicclass** Employee **implements** Comparator<Employee> {

**private** String name;

**public** Employee(String name) {

**super**();

**this**.name = name;

}

**public** Employee() {

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

}

**public** String getName() {

**return**name;

}

**publicvoid** setName(String name) {

**this**.name = name;

}

@Override

**public** String toString() {

**return**"Employee [name=" + name + "]";

}

@Override

**Public int** compare(Employee o1, Employee o2) {

Employee e1=(Employee)o1;

Employee e2=(Employee)o2;

**int**diff=e1.name.compareTo(e2.name);

**return**diff;

}

}

import java.util.Comparator;

import java.util.HashSet;

import java.util.LinkedHashSet;

import java.util.TreeMap;

import java.util.TreeSet;

public class HashSetEx {

public static void main(String[] args) {

TreeSet h =new TreeSet<>(new Employee());

h.add(new Employee("znarendra"));

h.add(new Employee("narendr"));

h.add(new Employee("nare"));

h.add(new Employee("nare"));

System.out.println(h);

}

}

Op:[Employee [name=nare], Employee [name=narendr], Employee [name=znarendra]]

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price. | The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. |
| 2) Comparable **affects the original class**, i.e., the actual class is modified. | Comparator **doesn't affect the original class**, i.e., the actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is present in **java.lang** package. | A Comparator is present in the **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method. |

***ArrayList and Vector:***  for storing and retrieve duplicate in random

ArrayList and Vector functionloties are same but ArryList is not a thread safe where as Vector is a thread safe(synchronized).

* Duplicates also allowed
* It is a growable array or incremental array
* Default capacity is 10
* Incremental capacity is (current capacity \*3)/2
* Load factor not applicable
* All types of elements are allowed
* Null insertion is possible, more than one allowed
* Storing in Insertion order

**import** java.util.Collections;

**import** java.util.Enumeration;

**import** java.util.Vector;

**publicclass** ArrayListEx {

**publicstaticvoid** main(String[] args) {

//ArrayList a=new ArrayList();

Vectora=**new**Vector();

System.***out***.println(a.isEmpty());

a.add("jdjd");

a.add(9);

a.add(**null**);//null allowed

a.add(**new** Exception());

a.add(**null**);

a.add(**new** Employee("narendra"));

a.add(**new** Employee("narendra"));

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

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

System.***out***.println(a.get(3));

System.***out***.println(a.isEmpty());

Enumeratione=Collections.*enumeration*(a);

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

Object b=e.nextElement();

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

}

}}

Op:true

[jdjd, 9, null, java.lang.Exception, null, Employee [name=narendra], Employee [name=narendra]]

7

java.lang.Exception

false

jdjd

9

null

java.lang.Exception

null

Employee [name=narendra]

Employee [name=narendra]

***Stack:*** for storing and retrieveing in LIFO order we choose Stack.

* Duplicates also allowed
* It is a growable array or incremental array
* Default capacity is 10
* Incremental capacity double
* Load factor not applicable
* All types of elements are allowed
* Null insertion is possible
* Storing in Insertion order

import java.util.HashSet;

import java.util.LinkedHashSet;

import java.util.LinkedList;

import java.util.Stack;

import java.util.TreeSet;

public class StackEx {

public static void main(String[] args) {

Stack a=new Stack<>();

a.add("a");

a.push("na");

a.add(new Employee("narendra"));

System.out.println(a);

a.pop();// [removing element from stack]

System.out.println(a);

Object b=a.peek();// [return element not remove from stack]

System.out.println(a);

System.out.println(b);

int s=a.search("na");

System.out.println(s);// returning searching object index

}

}

Op:[a, na, Employee [name=narendra]]

[a, na]

[a, na]

na

1

**Map Hierarchy:**

When we want store objects in key and value pairs we must choose Map Hierarchy classes. Map interface is root interface for all Map classes.

***HashMap***: for storing unique entries we must choose HashMap without worring insertion order we use HashMap.

-Here put() method returns object previous object if it is duplicate. Otherwise it returns null.

-Here object key is must unique not object value

-The initial capacity of the HashMap is **24**, i.e., **16**.

-The default load factor of HashMap is **0.75f** (75% of the map size).

**import** java.util.HashMap;

**publicclass** HashMapEx {

**publicstaticvoid** main(String[] args) {

HashMaph=**new** HashMap<>();

Object o=h.put("name","nara");

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

h.put("emp", **new** Employee("narendra "));

Object o1=h.put("name","narasimha");// if we add duplicate object to the map last value will be stored and previous object will be returned

System.***out***.println(o1);

h.put("add", "vja");

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

}

}

Op:

null

nara

{add=vja, name=narasimha, emp=Employee [name=narendra ]}

***LinkedHashMap***:for storing unique entries in insertion order we must choose LinkedHashMap.

***TreeMap***:storing unique entries in sorting order we must choose HashMap ,TreeMap Allows only comaparable unique entries

***Hashtable*** : HashMap and hashTable is same but hashTable is thread safe

***Properties***: class is used to store properties permanently in file.

**Internals of HashSet,LinkedHashSet, TreeSet, HashMap,LinkedHashMap :**

**1)** All above 5 methods internally uses HashTable Structure for storing objects.

2) HashTable internally stores Another object called Bucket.

3) Bucket is collection of elements those have same hashcode. This Bucket is also called collection of Object.

4) Bucket is storing same hash code objects as one group. And others as one group and others as one group...........For comparing and searching is easy.

5)if bucket is already available then add() method will check hashcode() and equals() methods and == operator.

6) if == returns true then it is duplicate it is not added to the collection.

7) == is false it is unique based on the reference then equals() method is called by the add() method.

8) equals returns false it will stored into the collection.

How can we implement our code to this 5 classes

EX:

**publicclass** Employee {

**privateint**id;

**private** String name;

@Override

**publicint** hashCode() {

**returnthis**.id;

}

@Override

**public** String toString() {

**return**"Employee [id=" + id + ", name=" + name + "]";

}

@Override

**publicboolean** equals(Object obj) {

**if**( obj**instanceof** Employee) {

Employee e=(Employee)obj;

**if**(**this**.name==e.name&&**this**.id==e.id) {

**returntrue**;

}

**returntrue**;

}

**returnfalse**;

}

**public** Employee(**int**id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

}

import java.util.HashSet;

import java.util.LinkedHashSet;

import java.util.TreeSet;

public class HashSetEx {

public static void main(String[] args) {

HashSet h= new HashSet<>();

h.add(new Employee(5,"narendra"));

h.add(new Employee(6,"narendra"));

h.add(new Employee(6,"narendra"));// it does not allowed because same id is checked by hashcode

System.out.println(h);

}

}

Retrieving Elements from List and set implemented classes:

Actually in list collection have get method to retrieve elements ,but Set implemented classes has no get method to retrieve elements.

* get: getting object only from list implemented classesa.get(2);

So retrieving elements from collection object without using index, we have 3 implicit cursor objects.

1) Enumeration

2) Iterator

3) ListIterator

All above three are interfaces are implemented in inner classes inside Collection classes.

These sub class factory methods to retrieve elements. So these factory methods must be called with collection object.

The factory methods are:

* To retrieve Enumeration object.

**public** Enumeration<E> elements()

* **This method is defined in Vector and HashTable**
* WE can get elements and keys also from
* Enumeratione=v.elements();
* Enumeratione1=h.keys();

**import** java.util.Enumeration;

**import** java.util.Hashtable;

**import**java.util.LinkedHashMap;

**import** java.util.Vector;

**publicclass** HashMapEx {

**publicstaticvoid** main(String[] args) {

Vectorv= **new** Vector<>();

v.add("narendra");

v.add("narendra");

v.add("narendra");

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

Hashtableh=**new** Hashtable<>();

h.put("name", "ram");

h.put("add", "vja");

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

Enumeratione=v.elements();

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

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

}

Enumeratione1=h.elements();

**while**(e1.hasMoreElements()) {

System.***out***.println(e1.nextElement());

}

}

}

Op:

[narendra, narendra, narendra]

{name=ram, add=vja}

narendra

narendra

narendra

ram

vja

How can we get all collection class elements from Enumeration interface.

We have a utility class Collections

ArrayLista= **new** ArrayList<>();

a.add("narendra");

a.add("narendra1");

a.add("narendra2");

a.add("narendra3");

Enumeratione3=Collections.*enumeration*(a);

**while**(e3.hasMoreElements()) {

System.***out***.println(e3.nextElement());

* }
* To retrieve Iterator object.

-**public** Iterator<E> iterator()

This method is defined all collection implement classes

Including Vector not hashtable is not a collection type

ArrayLista= **new** ArrayList<>();

a.add("narendra");

a.add("narendra1");

a.add("narendra2");

a.add("narendra3");

Iteratori =a.iterator();

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

System.***out***.println(i.next());

}

Op:narendra

narendra1

narendra2

narendra3

Iterator Rules

1) while retrieving element from the collection ,collection object should not be modified by either adding element or remove element using collection object methods. after this we call itr.next() it leads tojava.util.ConcurrentModificationException. this behaviour of Iterator is called fail-fast

2) remove (): method must be called only after next() else java.lang.IllegalStateException.

3) if u call next() method with empty collection we get java.util.NoSuchElementException

Iteratori =a.iterator();

//i.remove(); //must be called after next() ava.lang.IllegalStateException

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

System.***out***.println(i.next());

}

i.remove();

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

a.add("ram");

//System.out.println(i.next());//java.util.ConcurrentModificationException

Both Enumertation and iterator is same but Enumeration is legacy class.and both are fail-fast,so we can not modify collection while iterating

* To retrieve ListIterator object.

It is bidirectional cursor to retrieve elements only from list implemented collection

Objects- ArrayList, Vector, Stack, LinkedList.

ListIteratorli=a.listIterator();

**while**(li.hasNext()) {

System.***out***.println(li.next());

}

**while**(li.hasPrevious()) {

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

}

Op:narendra

narendra1

narendra2

ram

ram

narendra2

narendra1

narendra

For each loop: it can beused from java 5 onwords inplace of using iterator

* For loop syntax is enhanced for retrieving objects from arry and collection objects without using regular for loop and iterator

**for**(Object o:a) {

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

}

Op:

narendra

narendra1

narendra2

ram

*3 rules on for each loop:*

1) Object type should be array or iterable type

**int**b[] =**newint**[5];

b[2]=4;

**for**(Object o:b) {

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

}

2) Variable should declare inside condition

**for**(Object o:b)

3)Variable should be same type or super type of the objects retrieving from collection.

**for**(Object o:b) {

String s=(String) o;

}

Retrieving Elements from implemented classes:

* Using get (key )

**public** V get(Object key)

HashMaphs=**new** HashMap<>();

hs.put("name", "narendra");

hs.put("nam", "narendra");

System.***out***.println(hs.get("name"));

Op:narendra

If key is not available then null returned

* Using m.values()

**public** Collection<V> values(): it returns collection of values.

Collectionc= hs.values();

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

Op:[narendra, narendra]

* Using m.keyset()

**public** Set<K> keySet() : it returns set(key) values

Sets= hs.keySet();

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

Op:{name=narendra, nam=narendra}

* Using m.entryset()

**public** Set<Map.Entry<K,V>> entrySet()

Setd=hs.entrySet();

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

Op:[name=narendra, nam=narendra]

Introduction to generics:

* This feature is introduced in java5.0
* This is provide the type of method return type and method parameter dynamically achieving at compile time checking for solving ClasscastException
* Mainly generics concept is introduced the purpose of Collection for storing homogeneous objects
* ArrayLista3 =**new** ArrayList<>();
* ArrayLista3 =**new** ArrayList<>();
* a3.add("narendra");
* a3.add("kjjf");
* a3.add(9);
* a3.add(8);
* a3.add("gowri");
* **for**(Object o4:a3) {
* **if**(o4**instanceof** String) {
* System.***out***.println(o4);
* }
* Op:narendra
* kjjf
* gowri
* It allows heterogeneous and homogeneous collection . allows any type of objects if you want set generics. In retrieving we must check instanceof of object.
* ArrayList<String>a3 =**new** ArrayList<>();
* It allows only string type of objects. It is homogeneous collection or generic collection
* ArrayList<String>a3 =**new** ArrayList<>();
* a3.add("narendra");
* a3.add("kjjf");
* //a3.add(5);// its allow only string type it is generic type of string
* a3.add("gowri");
* **for**(Object o4:a3) {
* System.***out***.println(o4);
* }
* We no need to write instanceof operator because it allows only string type

If we define class/ interface/ enum with generic type

Class is called parameter type or generic type.

Method is called generic type method.

Constructor is called generic type constructor.

**publicclass** GenericEX<T> {

**public** GenericEX(T t){

}

**public** GenericEX(){

}

**public**<T>**int** m1() {

System.***out***.println(" m1 method ");

**return** 90;

}

**public**<T>**int** m2(T t) {

System.***out***.println(" m2 method ");

**return** 90;

}

**publicstaticvoid** main(String[] args) {

GenericEX<String>s=**new** GenericEX<>();

s.m1();

s.m2(**new** Integer(0));

s.m2(**new** String());

}

}

Op:

m1 method

m2 method

m2 methods

**How is the transient keyword used in Java?**

*Answer*: The transient keyword is used to indicate that a field in class should not be serialized (used with the Serializable interface)

**What is the difference between final, finalize and finally?**

*Answer*: Final is a modifier which you can apply to variable, methods, and classes. If you create a variable final, this means its value cannot be changed once initialised.

Finalise is a method, which is called just before an object is a garbage collected, allowing it a final chance to save itself, but the call to finalise is not definite.

Finally is a keyword which is used in exception handling, along with try and catch. The finally block is always implemented regardless of whether an exception is thrown from try block or not.

**How do you check if two given String are anagrams?**

*Answer*: Anagrams are a mix-up of characters in String e.g. army and mary, stop and pots etc. To identify if Strings are anagram, you will need to get their character array and identify if they are equal or not.

You are able to use indexOf(), substring() and StringBuffer or StringBuilder class to solve this question.

**What is the difference between throw and throws in Java?**

*Answer*: The throw is used to actually throw an instance of java.lang.throwable class, meaning you can throw both Error and Exception using throw keyword.

However, throws is used as part of method declaration and indicate which kind of exceptions are thrown by this method, so that its caller can handle them.

It is compulsory to assert any unhandled checked exception in throws clause in Java.

**What is the difference between Serializable and Externalizable in Java?**

*Answer*: Serializable interface is used to make Java classes serializable so that they can be transmitted over the network or their state can be kept on disk. However, it influences default serialization built-in JVM, which is pricey, fragile, and unsecured.

Externalizable lets you fully control the Serialization process, identify a customer binary format and enhance security measure.

**Java 8:**

**1)Lamda**

**interface** Sub{

**public** **int** sub(**int** a,**int** b);

}

**public** **class** SubtractionImpl {

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

Sub s=(a, b)-> b-a;

System.***out***.println(s.sub(10,15));

}

**public** List<Book> getSortingBooks() {

List<Book> books = **new** BookDao().getBooks();

Collections.*sort*(books,(o1,o2)-> {

**return** o2.getName().compareTo(o1.getName());

}

**2)Consumer**

**public** **interface** Consumer<T> {

/\*\*

\* Performs this operation on the given argument.

\*

\* **@param** t the input argument

\*/

**void** accept(T t);

**}**

**class** Demo **implements** Consumer<Integer>{

@Override

**public** **void** accept(Integer t) {

System.***out***.println( " print "+t);

}

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

**new** Demo().accept(9);

}

Later it simplifies like this

**public** **class** Demo {

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

Consumer<Integer> c=(t)->System.***out***.println("print"+t);

c.accept(6);

List<Integer> l=Arrays.*asList*(1,5,98,6,5,8);

l.forEach(c);

}

**3)Predicate :**

@FunctionalInterface

**public** **interface** Predicate<T> {

/\*\*

\* Evaluates this predicate on the given argument.

\*

\* **@param** t the input argument

\* **@return** {@code true} if the input argument matches the predicate,

\* otherwise {@code false}

\*/

**boolean** test(T t);

}

**public** **class** Demo {

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

Predicate<Integer> p=(t)->t%2==0;

**boolean** test = p.test(7);

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

List<Integer> l=Arrays.*asList*(1,2,3,4,5,6);

l.stream().filter(p).forEach(t->System.***out***.println("print "+t));

}

}

**4)Supplier :**

@FunctionalInterface

**public** **interface** Supplier<T> {

/\*\*

\* Gets a result.

\*

\* **@return** a result

\*/

T get();

}

**public** **class** Demo {

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

Supplier<String> s = () -> "hi";

String string = s.get();

//System.out.println(string);

List<String> l = Arrays.*asList*("a", "b");

String orElseGet = l.stream().findAny().orElseGet(s);

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

}

STREAM API:

* 1. Using foreach and filter

**public** **class** Demo {

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

List<String> l=**new** ArrayList<String>();

l.add("janu");

l.add("yatheesh");

l.add("narendra");

l.add("sudha");

l.add("balu");

**for**(String s:l) {

**if**(s.startsWith("j")) {

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

}

}

l.stream().filter(t->t.startsWith("j")).forEach(t->System.***out***.println(t));

l.stream().forEach(t->System.***out***.println(t));

Map<Integer,String> m=**new** HashMap<>();

m.put(1, "ramya");

m.put(2, "laxmi");

m.put(3, "sree");

m.forEach((keyd,value)->System.***out***.println(keyd+" "+value));

m.entrySet().stream().forEach(object->System.***out***.println(object));

m.entrySet().stream().filter(k->k.getKey()==1).forEach(t->System.***out***.println(t));

}