To Download a Selenium stand Alone Server, we need NPM(Node Package Manager).

Just like, Maven is used in Java World to manage dependencies,

NPM is used in Java Script World to manage dependencies.

Selenium Grid – Instead of one server executing the tests, grid makes use of multiple servers for execution.

Selenium Grid uses HUB and NODES to execute the tests.

The tests are submitted to the HUB, which then distributes to nodes.

Nodes Execute and run the tests. 10 to 15 nodes available.

1 HUB controls all the nodes, which tells which test should run where.

**TestNg vs Junit:**

TestNg and Junit are testing framworks for Selenium Automation Tests..

Junit tests can be eaily rewritten as TestNg tests by just changing the annotations and import them,along with the testing version in pom.xml file.

**Adv of testing over Junit framework for Selenium Automation Tests:**

1.Testng framework can have a xml file, when all the tests under a package can be run as a suite.

2.Testng creates test reports under the System Explorer of the directory name-> test-output->index.html

3.A Test can run in multiple Browsers using the ‘parameter’ field in testing.xml file, where you can set the parameter to “chrome” or “firefox” or “IE”.

4.Can run all the tests from testing suite in parallel by setting parallel = “true” in the testing.xml file.

5. ”thread-count” attribute is to set how many tests can run in parallel at the same time.

CLass and Objects:

~~~~~~~~~~~~~~~~~~

Class:

~~~~~

A prototype or template that helps us design an object's state and behavior.

Object:

~~~~~~~

It is a real-world entity of a class.

The class defines the state and the behavior of an object.

The state is defined by the member variable/attribute balance.

The behavior is defined by the member function/method getBalance().

public class Account

{

private double balance = 500.00;

public double getBalance(int accountId) {

return balance;

}

public static void main(String[] args) {

Account accnt = new Account();

double value = accnt.getBalance(123456);

System.out.println("The balance is: " + value);

}

}

1.public access modifier has been used for the main method as it is invoked using the Java tool by JVM.

2.The static method has been used for the main method as JVM cannot invoke the main method using the object of the class.

3.The method does not return any value, so the return type is void.

4.The parameter of the main method is a String array. Any number of inputs can be passed to the Java program during runtime using command-line arguments. A String array is used to store these inputs which can be used in our code.

/\*Program to understand classes and Objects with get set methods\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

public class Student{

private int studentId;

private String name;

private float qualifyingExamMarks;

private char residentialStatus;

private int yearOfEngg;

public int getStudentId(){

return studentId;

}

public void setStudentId(int studentId){

this.studentId = studentId;

}

public String getName(){

return name;

}

public void setName(String name){

this.name = name;

}

public float getqualifyingExamMarks(){

return qualifyingExamMarks;

}

public void setqualifyingExamMarks(float qualifyingExamMarks){

this.qualifyingExamMarks = qualifyingExamMarks;

}

public char getresidentialStatus(){

return residentialStatus;

}

public void setresidentialStatus(char residentialStatus){

this.residentialStatus = residentialStatus;

}

public int getyearOfEngg(){

return yearOfEngg;

}

public void setyearOfEngg(int yearOfEngg){

this.yearOfEngg = yearOfEngg;

}

public static void main(String[] args){

Student student = new Student();

student.setStudentId(1001);

student.setName("Jacob");

student.setqualifyingExamMarks(80);

student.setresidentialStatus('H');

student.setyearOfEngg(3);

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

System.out.println("Student Id : " +student.getStudentId());

System.out.println("Qualifying marks : " +student.getqualifyingExamMarks());

System.out.println("Year Of Engineering: " +student.getyearOfEngg());

String resstatus;

if(student.getresidentialStatus() == 'h' ||

student.getresidentialStatus() == 'H')

resstatus = "Hostellers";

else

resstatus = "Days Scholar";

System.out.println("Residential Status : " + resstatus);

}

}

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

/\*Program to understand the Scope Of Variables\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Local Variable - declared inside a method().

- can be used only inside that method().

- does not have a default value.

static variable- declared inside the class with static keyword.

- can be accessed anywhere within the class and outside the class with the class name.

- eg.Account.balance.

- Account is the class name

- balance is the static variable inside the Account class.

Class/

Instance variable - can be directly used inside the same class.

- object reference can be used to access it outside the class

- ex: accnt.balance.

- accnt is the object

- balance is the class variable.

public class ScopeOfVariables{

int i = 10; // instance variable

static int z; // static or class variable

static{

z=20;

System.out.println("Inside Static Block:" + z);

}

void test() {

int k = 200; // local variable

System.out.println("Local Variable:" +(k+i));

}

public static void main(String[] args){

ScopeOfVariables obj = new ScopeOfVariables();

obj.test();

System.out.println(obj.i);

}

}

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

/\* Develop a class "Rectangle". The class should have two instance variables with private access modifier.

length: int (length is the variable and int is the data type)

breadth: int

Use setter methods to initialize the instance variables.\*/

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

/\*Use the same "Rectangle" class created in the previous exercise of the same module and add a method "calculatePerimeter ()" which should calculate the perimeter based on the formula "2(length + breadth)" and prints on the console. Develop another class called PerimeterCalculator and add the main method which invokes the "calculatePerimeter ()" method on the Rectangle object.\*/

class Rectangle{

private int length;

private int breadth;

public void setLength(int length){

this.length = length;

System.out.println("The Length of the rectangle is :" +length);

}

public void setBreadth(int breadth){

this.breadth = breadth;

System.out.println("The Breadth of the rectangle is :" +breadth);

}

void calculatePerimeter(){

int perimeter = 2\*(length+breadth);

System.out.println("The Perimeter of the rectangle is:"+perimeter);

}

}

public class PerimeterCalculator{

public static void main(String[] args){

Rectangle obj = new Rectangle();

obj.setLength(60);

obj.setBreadth(40);

obj.calculatePerimeter();

}

}

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

/\*Typecast\*/

~~~~~~~~~~~~~

Converting a variable of larger datatype to the smaller datatype, with some data loss is known as narrowing conversion.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Narrowing conversion requires explicit type casting:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

public class ExplicitTypeCast{

public static void main(String[] args){

double d = 500;

long l = (long)d; // Explicit type casting

int i = (int)l; // Explicit type casting

System.out.println("double value: " +d);

System.out.println("long value: " +l);

System.out.println("int value: " +i);

}

}

Converting a variable of smaller datatype to the larger datatype, without data loss is known as widening conversion.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Widening conversion has implicit type casting:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

public class ImplicitTypeCast{

public static void main(String[] args){

int i = 300;

long l = i; //no explicit type casting

float f = l; //no explicit type casting

System.out.println("int value :"+i);

System.out.println("long value :"+l);

System.out.println("float value :"+f);

}

}

/\*A retail store management wants to provide a discount on the bill amount for its customers. Print the bill id, customer id, and the discounted bill amount for the .\*/

discountedBillAmount = billAmount - billAmount \* (discount/100)

public class RetailStore{

public static void main(String[] args){

int billId = 1001;

int customerId = 101;

int discount = 4;

double billAmount = 210.5;

double discountedBillAmount = billAmount -

(billAmount \* ((double)discount/100));

System.out.println("BillId: "+billId);

System.out.println("CustomerId: "+customerId);

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

}

}

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

/\*Create a class ReverseCalculator and implement the logic to reverse a number i.e., if the input is provided as 27, then the output should be displayed as 72, using appropriate control structures\*/

public class ReverseCalculator {

public static void main(String[] args)

{

int number = 27;

int orgnum = number;

int reverse = 0;

while(number>0)

{

int r = number%10;

reverse = (reverse\*10)+r;

number = number/10;

}

System.out.println("The reverse of the number "+orgnum+" using while is: " +reverse);

}

}

public class ReverseCalculator {

public static void main(String[] args)

{

int number = 27;

int orgnum = number;

int reverse = 0;

for(;number!=0;)

{

int r = number%10;

reverse = (reverse\*10)+r;

number = number/10;

}

System.out.println("The reverse of the number "+orgnum+" using for is: " +reverse);

}

}

public class ReverseCalculator {

public static void main(String[] args)

{

int number = 27;

int orgnum = number;

int reverse = 0;

do

{

int r = number%10;

reverse = (reverse\*10)+r;

number = number/10;

}

while(number!=0);

System.out.println("The reverse of the number "+orgnum+" using do while is: " +reverse);

}

}

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

/\*Create a class FunCalculator which calculates an Armstrong number and a lucky number. An Armstrong number is one in which the sum of the cube of the digits of a number results in the number itself. A lucky number is a number where the sum of squares of every even positioned digit (starting from the second position) is a multiple of 9.\*/

For e.g.: 153 = 13+53+33 = 153 is an Armstrong number.

1623 = 62+32 = 45 is a multiple of 9 and hence is a Lucky number.

import java.util.\*;

public class FunCalculator {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

int number = sc.nextInt();

System.out.println("Enter a Choice to check: \n1.Armstrong number\n"+"2.Lucky number");

int choice = sc.nextInt();

int orgnum = number;

int cubeSum = 0;

int sqsum = 0;

switch(choice) {

case 1:

while(number!=0) {

int r = number%10;

cubeSum = (cubeSum)+(r\*r\*r);

number = number/10;

}

System.out.println("your choice is " +choice);

if(cubeSum == orgnum) {

System.out.println("The number "+orgnum+" is an Armstrong Number!");

} else

System.out.println("The number "+orgnum+" is not an Armstrong Number!");

break;

case 2:

String s = Integer.toString(number);

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

int n = st.length;

for(int i=1; i<n; i+=2) {

int val = Integer.parseInt(st[i]);

sqsum = sqsum+(val \* val);

//number = number/10;

}

System.out.println("your choice is " +choice);

if(sqsum%9 == 0)

System.out.println("Entered number is a lucky Number!");

else

System.out.println("Entered number is not a lucky Number!");

break;

default:

System.out.println("Invalid Choice!");

}

}

}

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

/\*Program to break from a loop\*

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

public class WithLabelledLoop {

public static void main(String args[]) {

int i,j;

loop1: for(i=1; i<=10; i++) {

System.out.println();

loop2: for(j=1; j<=10; j++) {

System.out.print(j + " ");

if(j==1)

break loop1; //Statement 1

}

}

}

}

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

/\*Enhanced for Loop\*/

~~~~~~~~~~~~~~~~~~~~~

public class Bank {

public static void main(String[] args) {

double balance = 6000;

double rateOfInterest = 0.10;

double interest = 0;

double withdrawal = 500;

double deposit = 600;

int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

for(int i: arr) {

balance += deposit;

balance -= withdrawal;

interest = balance \* rateOfInterest;

balance += interest;

System.out.println("The interest for the month " + i + " is " + interest);

}

}

}

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

/\*Arrays\*/

* An Array can contain no value or n number in it.
* To find array length = arrayname.length;
* In arrays length is a property.
* But in Strings length is a method(). Eg.str.length();
* An array of a class can be created.
* toString is a static method in Arrays class.
* Eg. (Arrays.toString(marks))
* Arrays.fill(arrayname,value) will fill the entire array with the value.
* Eg.Arrays.fill(marks,100) – will fill the marks array with the value 100.
* Arrays.equals(arr1,arr2) checks two arrays and returns a Boolean value.
* Arrays.equals returns true only when both the arrays has equal length and same values.
* Arrays.sort(arr) will sort the elements of the array.

Class Person{

}

Person[] persons = new Person[5];

Creating an array of a class will create an array of **null** values by default.

Advantages of Arrays:

~~~~~~~~~~~~~~~~~~~~~

1.Type checking is done at Compile time

2.Ability to hold objects as well as primitive type data.

import java.util.\*;

public class Bank {

public static void main(String[] args) {

Customer[] customer = new Customer[2];

Customer customer1 = new Customer("A","Accno1234");

Customer customer2 = new Customer("B","Accno5678");

customer[0] = customer1;

customer[1] = customer2;

int count = 1;

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

Customer custobj = customer[i];

String custName = custobj.getCustName();

System.out.println("Customer " +count+": \n");

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

String custId = custobj.getCustId();

System.out.println("Customer Id is : " +custId+"\n");

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

count++;

}

}

}

class Customer {

String custName;

String custId;

Customer(String custName, String custId) {

this.custName = custName;

this.custId = custId;

}

public String getCustName() {

return custName;

}

public String getCustId() {

return custId;

}

}

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

/\*A project unit in a company wants to keep track of five employees in the project and their salaries, and also find out the average of their salaries. They also want to find the number of employees who get a salary greater than the average salary and those who get lesser.

public class EmployeeRecord {

public static void main(String[] args) {

double totalSal=0;

double average =0;

int greatcount = 0;

int lesscount = 0;

double salary[] = {23500.0, 25080.0, 28760.0, 22340.0, 19890.0};

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

totalSal = totalSal + salary[i];

}

average= totalSal/salary.length ;

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

if(salary[i]<average) {

lesscount++;

} else

greatcount++;

}

System.out.println("Employee Record"+"\n");

System.out.println("---------------"+"\n");

System.out.println("The Average Salary of the Employee is : " +average+"\n");

System.out.println("The No.Of Employees having salary greater than average is : "+greatcount+"\n");

System.out.println("The No.Of Employees having salary lesser than average is : "+lesscount);

}

}

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

/\*Create a Java class and initialize an integer array of size 5. Create two methods, one method will store the marks of 5 subjects in the array and another method will display the data from the array. Traverse the array using enhanced for loop.\*/

public class StudentMarks {

public static void main(String[] args) {

int[] marks = {95,90,98,85,100};

int count = 1;

for(int i : marks) {

Marks marobj = new Marks();

marobj.setMarks(i);

System.out.println("Subject " + count+":" +marobj.displayMarks());

count++;

}

}

}

class Marks {

private int marks;

public void setMarks(int marks) {

this.marks = marks;

}

public int displayMarks() {

return marks;

}

}

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

Object Oriented Concepts in Java:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Abstraction: - identify essentials

~~~~~~~~~~~~~

Abstract class should start with the abstract keyword.

Abstract method should start with the abstract keyword and end with a semi-colon(;). But no implementation should be given inside it.

An Abstract class can have a non- abstract method as well.

Unlike interfaces ,an abstract class have private methods as well.

Unlike Interfaces, an abstract class can have all kinds of variables.

Eg:

Abstract class Animal{

Abstract public void bark();

}

**The Sub Class will have the implementation of that abstract method.**

**The Sub class extending the abstract super class should have an implementation of the method in super class. Or else it will throw an error.**

**An Abstract class can extend another abstract class. It need not have an implementation.**

Class Dog extends Animal{

Public void bark(){

System.out.println(“Im a Dog”);

}

}

Abstraction is the process of identifying the essential details to be known and ignoring the non-essential details from the perspective of the user of the system.

Abstraction is a generic concept, where we try to hide all the complexities.

Eg:

Banking Application - Branch Manager, Accountant, Cashier. Each user needs to know some details and need not know other details of the bank.

Encapsulation: - Wrapping data collectively

~~~~~~~~~~~~~

In order to hide the internal state from the outside world, we wrap data and code collectively within a single unit, this process is known as Encapsulation. It mandates that all interactions must be performed through an object's methods.

Encapsulation is related to hiding data belonging to a specific object. Only operations can be performed , access to data is not allowed.

class Customer{

double accountBalance;

double getBalance(){

}

}

It means that the variable accountBalance of the Customer class should not be accessed directly by other class entities. It can be accessed only with getBalance() method. Thus we can say that we are hiding the state of Customer class outside this class.

This is possible through access modifiers in Java.

Inheritance: - reusability

~~~~~~~~~~~~

In a Bank, we have different types of loans such as HomeLoan, CarLoan, GoldLoan. All these Loans share some common state and behavior.

Inheritance helps us to improve the reusability of code in such cases.

The attributes and behavior common to all types of loans are modeled as Loan class.

class Loan{

double tenure;

double principal;

double interestrate;

void calculateEMI(){

}

}

Now all the Loan classes can use this data. This reusability of state and behavior is achieved by Inheritance.

VehicleLoan,CarLoan,GoldLoan can inherit from Loan Class.

Polymorphism:

~~~~~~~~~~~~~

Poly - many: Morph - forms

A task performed in many ways is Polymorphism.

Constructor:

~~~~~~~~~~~~

Constructor is a special method without return type having name same as class name. Once the object gets created it is being initialized by the constructor.

Constructor is used to set a initial state to an object by setting a value.

Can assign initial values to the member variable using a constructor.

1.Constructor has the same name as the class name.

2.It doesn’t have a return type.

3.It may or may not have parameters.

4.Whenever you create an object of the sub class which extends a super class, by default the super class constructor is implicitly called, then the sub class constructor is called.

this keyword:

~~~~~~~~~~~~

1.this keyword is used to refer to the current instance of an object.

2.It can be used to access instance variables if both local variables and instance variables have the same name.

/\*ABC Confectionary is a chocolate manufacturer. Every chocolate which is manufactured will be with a default weight and cost. The cost and weight might be modified later based on business needs.\*/

Using default constructor, parameterized constructor,this keyword:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

class Chocolate{

int barcode;

String name;

double weight;

double cost;

Chocolate()

{

barcode = 101;

name = "Cadbury";

weight = 12;

cost = 10;

}

Chocolate(int barcode,String name,double weight,double cost)

{

this.barcode = barcode;

this.name = name;

this.weight = weight;

this.cost = cost;

}

int getBarCode(){

return barcode;

}

void setBarCode(int barcode){

this.barcode = barcode;

}

String getName(){

return name;

}

void setName(String name){

this.name = name;

}

double getWeight(){

return weight;

}

void setWeight(double weight){

this.weight = weight;

}

double getCost(){

return cost;

}

void setCost(double cost){

this.cost = cost;

}

}

public class Chocolate1{

public static void main(String[] args){

Chocolate chocolate = new Chocolate();

System.out.println("Calling default Constructor\n");

System.out.println("BarCode:" +chocolate.getBarCode());

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

System.out.println("Weight:" +chocolate.getWeight());

System.out.println("Cost:" +chocolate.getCost());

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

System.out.println("Calling parameterized Constructor\n");

Chocolate chocolate1 = new Chocolate(101,"Cadbury",12,10);

System.out.println("BarCode:" +chocolate1.getBarCode());

System.out.println("Name:" +chocolate1.getName());

System.out.println("Weight:" +chocolate1.getWeight());

System.out.println("Cost:" +chocolate1.getCost());

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

System.out.println("Modifying values using setter\n");

chocolate.setBarCode(102);

chocolate.setName("Hershey's");

chocolate.setWeight(24);

chocolate.setCost(50);

System.out.println("BarCode:" +chocolate.getBarCode());

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

System.out.println("Weight:" +chocolate.getWeight());

System.out.println("Cost:" +chocolate.getCost());

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

}

}

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

/\*Program using parameterized constructor\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

public class BankApplication{

public static void main(String[] args){

Bank bank = new Bank("IBank","Chennai","979279");

bank.displayBankDetails();

}

}

class Bank{

String bankName;

String area;

String phNo;

Bank(String bName,String area,String phNo)

{

this.bankName = bName;

this.area = area;

this.phNo = phNo;

}

void displayBankDetails()

{

System.out.println("Bank Name:" + bankName);

System.out.println("Bank Area:" + area);

System.out.println("Bank PhNo:" + bankName);

}

}

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

Introduction to inheritance:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

By Default, the super class/parent class extends the Object Class.

The Object class has some predefined methods like toString(),hachCode() etc.

1.Inheritance makes the subclass inherit all the characteristics of the superclass.

2.The parent class is termed as the superclass and the child class is the subclass.

3.The most important advantage of inheritance is reusability.

4.A subclass inherits the member-variables and the member methods of the superclass.

5.Inheritance can be either Single-Level or Multi-Level Inheritance

6.Java does not support **Multiple** Inheritance in the form of classes.

One class cannot extend from two classes.

7.In Java, the keyword extends is used by the child class to inherit the behavior of the parent class.

8.When a subclass instance is created, the order of constructor execution is from the root class constructor in the hierarchy to the subclass constructor.

9. A reference variable of a sub class can be held by a super class, but not vice versa.

Eg:

Class Animal{}

Class Pet extends Animal{}

Class Dog extends Pet{}

Dog dog = new Dog();

1.Dog constructor is called. Then Pet Constructor, Animal constructor ,and finally the Object Constructor.

Pet pet = new Dog(); is allowed

Dog dog = new Pet() ; is not allowed

class Loan{

String accountNo;

double principal;

int tenure;

float interest;

public double calculateEMI()

{

double SI = (principal\*tenure\*interest)/100;

double EMI = (SI+principal)/tenure;

return EMI;

}

}

class HomeLoan extends Loan{

HomeLoan()

{

accountNo = "AccNo12345";

principal = 20000;

tenure = 5;

interest = 8.5f;

}

}

public class Loan1{

public static void main(String[] args){

HomeLoan hloan = new HomeLoan();

double amt = hloan.calculateEMI();

System.out.println("EMI per year is: " +amt);

}

}

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

A construction company wants to keep a record of the employees working in it. There are permanent employees as well as contract employees. Contract employees work on an hourly basis whereas permanent employees are paid monthly salary.

class Employee {

int empId;

String name;

double salary;

double getSalary() {

return salary;

}

void setSalary(double salary) {

this.salary = salary;

}

int getEmpId() {

return empId;

}

void setEmpId(int empId) {

this.empId = empId;

}

String getName() {

return name;

}

void setName(String name) {

this.name = name;

}

}

class PermanentEmployee extends Employee {

double basicPay;

double hra;

int experience;

//Constructor

/\*PermanentEmployee(String name,int empId,double basicPay,double hra,int experience)

{

this.name = name;

this.empId = empId;

this.basicPay = basicPay;

this.hra = hra;

this.experience = experience;

}\*/

double getBasicPay() {

return basicPay;

}

void setBasicPay(double basicPay) {

this.basicPay = basicPay;

}

double getHra() {

return hra;

}

void setHra(double hra) {

this.hra = hra;

}

int getExperience() {

return experience;

}

void setExperience(int experience) {

this.experience = experience;

}

void calculateSalary() {

double component;

if(experience<3)

{

component = 0;

}

else if(experience>=3 && experience<5)

{

component = (5\*basicPay)/100;

}

else if(experience>=5 && experience<10)

{

component = (7\*basicPay)/100;

}

else

{

component = (12\*basicPay)/100;

}

salary = component + basicPay + hra;

}

}

class ContractEmployee extends Employee {

double wages;

int hours;

double getWages(){

return wages;

}

void setWages(double wages){

this.wages = wages;

}

int getHours(){

return hours;

}

void setHours(int hours){

this.hours = hours;

}

void calculateSalary(){

salary = hours \* wages;

}

}

public class EmployeeRecords{

public static void main(String[] args){

//calling parameterised constructor

//PermanentEmployee pempl = new PermanentEmployee("Anil",101,10000,1500,3);

PermanentEmployee pempl = new PermanentEmployee();

ContractEmployee cempl = new ContractEmployee();

pempl.setName("Anil");

pempl.setEmpId(101);

pempl.setBasicPay(10000);

pempl.setHra(1500);

pempl.setExperience(3);

cempl.setName("Ankit");

cempl.setEmpId(102);

cempl.setWages(500);

cempl.setHours(10);

pempl.calculateSalary();

System.out.println("Perm:Your Salary is:"+pempl.getSalary());

cempl.calculateSalary();

System.out.println("Cont:Your Salary is:"+cempl.getSalary());

}

}

Interfaces:

An interface can have common actions between classes.

It can hold the common actions that can be performed.

So with which we can use multiple implementations inside the classes that implements it.

Interface provides us a way to communicate b/w two different projects.

Interfaces establishes a **contract** between two classes which are talking to each other.

An interface can extend another interface.

If an interface extends another interface, and if a class wants to implement one of those interfaces, then it should implement methods from both the interfaces.

But if you want to implement only one of the methods in the interfaces, then we can use an abstract class.

An abstract class can implement part of the methods defined in the interfaces.

A class can implement multiple interfaces, but cannot extend multiple abstract classes.

Interfaces can also have default methods using the keyword **default** in the method.(only after Java 8)

The default method in an interface can be overridden in a class which implements it.

Interfaces can have only public methods.

* Cannot have variables with constant values which cannot change.

Polymorphism

~~~~~~~~~~~~~

The ability of an object to behave differently for the same method call is called polymorphism.

Types of Polymorphism:

~~~~~~~~~~~~~~~~~~~~~~

Static Polymorphism:

~~~~~~~~~~~~~~~~~~~~

1.Method overloading

2.It uses compile-time binding

3.Method invoked is based on the reference type.

4.Constructors can be overloaded.

Dynamic Polymorphism:

~~~~~~~~~~~~~~~~~~~~~

1.Method overriding.

2.It uses late binding or runtime binding

3.Method invoked is based on the object type.

4.It lets to redefine a method in a child class which is already defined in a parent class.

5.Private methods cannot be overridden.

6.The overridden method cannot have a weaker access privilege in the child class.

/\*OverLoading constructor\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.\*;

class Registration {

private String customerName;

private String panCardNo;

private int voterId;

private String passportNo;

private int licenseNo;

private long[] telephoneNo;

Registration(String customerName,String passportNo,long[] telephoneNo ) {

this.customerName = customerName;

this.passportNo = passportNo;

this.telephoneNo = telephoneNo;

}

Registration(String customerName,int licenseNo,String panCardNo,long[] telephoneNo) {

this.customerName = customerName;

this.licenseNo = licenseNo;

this.panCardNo = panCardNo;

this.telephoneNo = telephoneNo;

}

Registration(String customerName,int voterId,int licenseNo,long[] telephoneNo) {

this.customerName = customerName;

this.voterId = voterId;

this.licenseNo = licenseNo;

this.telephoneNo = telephoneNo;

}

Registration(String customerName,String panCardNo,int voterId,long[] telephoneNo) {

this.customerName = customerName;

this.panCardNo = panCardNo;

this.voterId = voterId;

this.telephoneNo = telephoneNo;

}

public String getCustomerName() {

return customerName;

}

public String getPanCardNo() {

return panCardNo;

}

public int getVoterId() {

return voterId;

}

public String getPassportNo() {

return passportNo;

}

public int getLicenseNo() {

return licenseNo;

}

public long[] getTelephoneNo() {

return telephoneNo;

}

}

public class EnigmaInternet {

public static void main(String[] args) {

long[] telephoneNo = {9452425421l,7676765252l};

Registration register = new Registration("Kevin","MN9891N",telephoneNo);

System.out.println("Congratulations "+register.getCustomerName()+"!!! you have been successfully registered for our services with the following details:\n");

System.out.println("Passport Number:" +register.getPassportNo());

System.out.println("Phone Numbers:" +Arrays.toString(register.getTelephoneNo()));

long[] telephoneNo1 = {2345615451l,6763562562l};

Registration register1 = new Registration("Julias",123,"PN7878",telephoneNo1);

System.out.println("Congratulations "+register1.getCustomerName()+"!!! you have been successfully registered for our services with the following details:\n");

System.out.println("License Number:" +register1.getLicenseNo());

System.out.println("Pan card Number:" +register1.getPanCardNo());

System.out.println("Phone Numbers:" +Arrays.toString(register1.getTelephoneNo()));

long[] telephoneNo2 = {9634524353l,9887373737l};

Registration register2 = new Registration("Jammy",45453,765,telephoneNo2);

System.out.println("Congratulations "+register2.getCustomerName()+"!!! you have been successfully registered for our services with the following details:\n");

System.out.println("Voter Id:" +register2.getVoterId());

System.out.println("License Number:" +register2.getLicenseNo());

System.out.println("Phone Numbers:" +Arrays.toString(register2.getTelephoneNo()));

long[] telephoneNo3 = {9867456367l,7645367356l};

Registration register3 = new Registration("Rose","PN8934",34356,telephoneNo3);

System.out.println("Congratulations "+register3.getCustomerName()+"!!! you have been successfully registered for our services with the following details:\n");

System.out.println("Pan card Number:" +register3.getPanCardNo());

System.out.println("Voter Id:" +register3.getVoterId());

System.out.println("Phone Numbers:" +Arrays.toString(register3.getTelephoneNo()));

}

}

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

/\*Method overloading\*/

~~~~~~~~~~~~~~~~~~~~~~

class PlayerRating{

private int playerPosition;

private String playerName;

private float criticOneRating;

private float criticTwoRating;

private float criticThreeRating;

private float averageRating;

private char category;

PlayerRating(int playerPosition,String playerName)

{

this.playerPosition = playerPosition;

this.playerName = playerName;

}

public void calculateAverageRating(float criticOneRating,float criticTwoRating)

{

averageRating = (criticOneRating + criticTwoRating)/2;

}

public void calculateAverageRating(float criticOneRating,float criticTwoRating,float criticThreeRating)

{

averageRating = (criticOneRating + criticTwoRating + criticThreeRating)/3;

}

public void calculateCategory()

{

if (averageRating>8)

category = 'A';

else if ((averageRating>5) && (averageRating<=8))

category = 'B';

else

category = 'C';

}

public void display()

{

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

System.out.println("The Player Position is:" +playerPosition);

System.out.println("The Average Rating is:" +averageRating);

System.out.println("The Category is:" +category);

}

}

public class PlayerClub{

public static void main(String[] args){

PlayerRating obj = new PlayerRating(1,"Beckham");

obj.calculateAverageRating(9f,9.9f);

obj.calculateCategory();

obj.display();

PlayerRating obj1 = new PlayerRating(1,"Oscar");

obj1.calculateAverageRating(1f,1f,1f);

obj1.calculateCategory();

obj1.display();

}

}

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

Super Keyword:

~~~~~~~~~~~~~~

if a child class method requires all the code in the parent class to be executed and along with it,

the super keyword can be used to achieve this.

Super can be used to

\*To invoke a parent class constructor,the super statement must be the first statement in the child class constructor.

To use a super class default constructor from subclass,use super().

To use a super class parameterized constructor taking an int value, use super(int).

\*To access instance variables of parent class(other than private variables) from the child class, super can be used when both the parent

and child class have variables with same name.

\*To access methods of parent class(other than private) from the child class, super can be used when both the parent

and child class have methods with same name.

for eg:

~~~~~~~

class Loan {

double calculateEMI(double principal) {

double simpleInterest = (principal\*8.5\*5) / 100;

double emi = (simpleInterest+principal)/5;

return emi;

}

}

class HomeLoan extends Loan{

double calculateEMI(double principal){

double emi = super.calculateEMI(principal); //To invoke parent class method super keyword is used

int additionaltax = 300;

emi= emi + additionaltax;

return emi;

}

}

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

/\*A college cultural event "Show Your talent" is being conducted and the organizing committee has decided to come up with online registration for the same. Initially, the registration fee for single and team events was the same, but later the organizing committee decided to calculate the registration fee based on the number of participants.\*/

Method Overriding and Super:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

class EventRegistration {

String name;

String nameOfEvent;

double registrationFee;

EventRegistration(String name,String nameOfEvent) {

this.name = name;

this.nameOfEvent = nameOfEvent;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getNameOfEvent() {

return nameOfEvent;

}

public void setNameOfEvent(String nameOfEvent) {

this.nameOfEvent = nameOfEvent;

}

public double getRegistrationFee() {

return registrationFee;

}

public void setRegistrationFee(double registrationFee) {

this.registrationFee = registrationFee;

}

public void registerEvent() {

System.out.println("Thank you " +name+" for your participation.");

if(nameOfEvent.equalsIgnoreCase("ShakeALeg")) {

registrationFee = 100;

} else if (nameOfEvent.equalsIgnoreCase("Sing&Win")) {

registrationFee = 150;

} else if (nameOfEvent.equalsIgnoreCase("Actathon")) {

registrationFee = 70;

} else if (nameOfEvent.equalsIgnoreCase("PlayAway")) {

registrationFee = 130;

} else {

System.out.println("Please choose a valid event");

}

System.out.println("Your registration fee is:" +registrationFee);

}

}

class SingleEventRegistration extends EventRegistration {

int participantNo;

SingleEventRegistration(String name,String nameOfEvent,int participantNo) {

super(name,nameOfEvent);

this.name = name;

this.nameOfEvent = nameOfEvent;

this.participantNo = participantNo;

}

public int getparticipantNo() {

return participantNo;

}

public void setparticipantNo(int participantNo) {

this.participantNo = participantNo;

}

public void registerEvent() {

super.registerEvent();

System.out.println("You are participant No:" +participantNo+"\n");

}

}

class TeamEventRegistration extends EventRegistration {

int noOfParticipants;

int teamNo;

TeamEventRegistration(String name,String nameOfEvent,int noOfParticipants,int teamNo) {

super(name,nameOfEvent);

this.name = name;

this.nameOfEvent = nameOfEvent;

this.noOfParticipants = noOfParticipants;

this.teamNo = teamNo;

}

public int getnoOfParticipants() {

return noOfParticipants;

}

public void setnoOfParticipants(int noOfParticipants) {

this.noOfParticipants = noOfParticipants;

}

public int getteamNo() {

return teamNo;

}

public void setteamNo(int teamNo) {

this.teamNo = teamNo;

}

public void registerEvent() {

//super.registerEvent();

int baseFee = 0;

if(nameOfEvent.equalsIgnoreCase("ShakeALeg")) {

baseFee = 50;

} else if (nameOfEvent.equalsIgnoreCase("Sing&Win")) {

baseFee = 60;

} else if (nameOfEvent.equalsIgnoreCase("Actathon")) {

baseFee = 80;

} else if (nameOfEvent.equalsIgnoreCase("PlayAway")) {

baseFee = 100;

} else {

System.out.println("Please choose a valid event");

}

registrationFee = baseFee \* noOfParticipants;

System.out.println("Thank you " +name+" for your participation.");

System.out.println("Your registration fee is:" +registrationFee);

System.out.println("The team No. is" +teamNo+"\n");

}

}

public class ShowYourTalentRegistration {

public static void main(String[] args) {

EventRegistration participant1 = null;

EventRegistration participant2 = null;

EventRegistration team1 = null;

participant1 = new SingleEventRegistration("Jenny","Sing&Win",1);

participant1.registerEvent();

team1 = new TeamEventRegistration("Aura","ShakeALeg",5,1);

team1.registerEvent();

participant2 = new SingleEventRegistration("Hudson","PlayAway",2);

participant2.registerEvent();

}

}

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

Final Keyword:

~~~~~~~~~~~~~~

Final can be used in 3 scenarios:

1.Before a variable:

~~~~~~~~~~~~~~~~~~~~

A final variable's value once initialized can't be changed, i.e. it is a constant private final int Tenure = 20;

If a variable is marked as final,it can be initialised only once either in the constructor or while declaring.

2.Before a method:

~~~~~~~~~~~~~~~~~~

A final method cannot be overridden in a subclass

public final void calculateEMI(){...}

3.Before a class:

~~~~~~~~~~~~~~~~~

A final class cannot be subclassed. (i.e. you cannot extend the class)

public final class Loan{...}

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

Static Modifier:

~~~~~~~~~~~~~~~~

1.Static variables are global variables that all the objects of a particular class share.

2.The static keyword is used before

i) Variables:

`````````

static variables are variables prefixed with the static keyword.

static variables are global variables, which are shared by all instances of the class.

static variables are accessed using the class name, it can also be accessed using objects.

ii) Methods:

````````

static methods are methods prefixed with the static keyword.

static methods access static members of the class.

static methods are accessed through class names.

iii) Block of code:

`````````````

static blocks are blocks prefixed with the static keyword.

static blocks execute when the class is loaded.

static block executes only once irrespective of how many instances created for that class.

static block executes before the main method executes.

iv) import statement (will be explained in later sections)

3.A method or a variable which is static can be called directly by using its class name rather than creating an object reference of that class.

4.A static block gets executed only once, irrespective of how many instances created for that class

Note: The main method in JVM does not have an instance to call. So it calls by the class name.

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

/\*Program to understand how to create variables, blocks, and methods using static and how to instantiate objects outside the class for the class's private constructors.\*/

class Loan {

int loanNo;

int accountNo;

int customerNo;

float loanAmount;

int loanDuration;

float interest;

private static int loanCounter;

static{

loanCounter = 101;

}

Loan() {

loanCounter++;

}

static int getLoanCounter() {

return loanCounter;

}

static void setLoanCounter(int loanCounter) {

loanCounter = loanCounter;

}

public int getLoanNo() {

return loanNo;

}

public void setLoanNo(int loanNo) {

this.loanNo = loanNo;

}

public int getAccountNo() {

return accountNo;

}

public void setAccountNoNo(int accountNo) {

this.accountNo = accountNo;

}

public int getCustomerNo() {

return customerNo;

}

public void setCustomerNo(int customerNo) {

this.customerNo = customerNo;

}

public float getLoanAmount() {

return loanAmount;

}

public void setLoanAmount(float loanAmount) {

this.loanAmount = loanAmount;

}

public int getLoanDuration() {

return loanDuration;

}

public void setLoanDuration(int loanDuration) {

this.loanDuration = loanDuration;

}

public float getInterest() {

return interest;

}

public void setInterest(float interest) {

this.interest = interest;

}

Loan(int accountNo,int customerNo,float loanAmount,int loanDuration,float interest) {

loanCounter++;

this.accountNo = accountNo;

this.customerNo = customerNo;

this.loanAmount = loanAmount;

this.loanDuration = loanDuration;

this.interest = interest;

}

}

public class LoanTester {

public static void main(String[] args) {

Loan obj1 = new Loan();

Loan obj2 = new Loan();

Loan obj3 = new Loan(1102,101,20000f,1,2f);

Loan obj4 = new Loan(1102,101,20000f,1,2f);

int counter = obj1.getLoanCounter();

System.out.println("LoanCounter:" +counter);

counter = obj2.getLoanCounter();

System.out.println("LoanCounter:"+counter);

counter = obj3.getLoanCounter();

System.out.println("LoanCounter:"+counter);

counter = obj4.getLoanCounter();

System.out.println("LoanCounter:"+counter);

}

}

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

/\*Program using static and final keywords\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

class Employee {

private int point;

public int getPoint() {

return point;

}

public void setPoint(int point) {

this.point = point;

}

}

class PerformanceRating {

private static final int Outstanding = 5;

private static final int Good = 4;

private static final int Average = 3;

private static final int Poor = 2;

static int calculatePerformance(Employee employee) {

if (employee.getPoint()>=80 && employee.getPoint()<=100)

return Outstanding;

else if(employee.getPoint()<=79 && employee.getPoint()>=60)

return Good;

else if(employee.getPoint()<=59 && employee.getPoint()>=50)

return Average;

else

return Poor;

}

}

public class PerformanceCalculator{

public static void main(String[] args){

int rating;

Employee obj1 = new Employee();

obj1.setPoint(98);

rating = PerformanceRating.calculatePerformance(obj1);

System.out.println("obj1 "+rating);

Employee obj2 = new Employee();

obj2.setPoint(60);

rating = PerformanceRating.calculatePerformance(obj2);

System.out.println("obj2 "+rating);

Employee obj3 = new Employee();

obj3.setPoint(24);

rating = PerformanceRating.calculatePerformance(obj3);

System.out.println("obj3 "+rating);

}

}

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

Exceptions:

~~~~~~~~~~

Exceptions are caused by events that deviate from their normal expected flow.

The runtime environment (JRE) produces and throws the exception object whenever an exception occurs. The instant after an exception object is thrown, the program stops the execution. If the exception is not taken care of, then it is propagated to the calling environment. It can be either a runtime system or a calling method.

Because when JVM is unable to allocate memory that is an IO Error that can’t be handled. So the program execution stops there.

The intValue() method, when invoked on a null value throws NullPointerException during runtime.

The following methods from Throwable class are used to get exception details:

1.String getMessage()

2.void printStackTrace()

3.String toString()

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

public class ExceptionDemo {

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

return a/b;

}

public static void main(String[] args) {

try {

divide(9,0);

} catch (ArithmeticException exception) {

System.out.println(exception);

//exception.printStackTrace();

//System.out.println(exception.getMessage());

//System.out.println(exception.toString());

}

finally {

System.out.println("Inside finally");

}

}

}

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

Collections:

~~~~~~~~~~~~

Using Java Collections,various operations can be performed on the data like searching,sorting,insertion,manipulation,deletion etc.

We know arrays have advantages like

1.Type checking is done at Compile time

2.Ability to hold objects as well as primitive type data

However, they can not grow and shrink dynamically. In addition, they do not have an integrated algorithm for sorting or searching.

To overcome these disadvantages, Java gives us the Collections Frameworks.

Collections FrameWork:

~~~~~~~~~~~~~~~~~~~~~~~

\*The collections framework is dynamic in nature.

\*We have contains() method to search and sort() method to sort.

\*We have add() and remove() methods that are inbuilt to add elements and delete elements respectively.

\*Collections are reference types so they will be allocated on different memory locations.

\*The collections framework offers many interfaces and classes for manipulating and representing collections. Introduced in J2SE 1.2, it regulates the way we access data and store using collections. All these are in java.util package.

The framework has the following advantages:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

1.Ready to use classes and algorithms.

2.Better program speed and quality.

3.Reduced programming effort.

List -- Interface

* List Interface cares about which position each object is in.
* Like arrays, it allows index-based access to its elements
* Elements can be added in by specifying position-where it should be added in
* If element is added without specifying the position – it is added at the end.
* Lists are ordered collections which allow duplicates.

class **ArrayList** implements List<E>

* Array implementation of the List Interface.
* Allows random access and hence fast.
* Addition and removal of elements is time consuming compared to LinkedList.

class **LinkedList** implements List<E>

* Doubly Linked implementation of the List Interface.
* Allows sequential access from front and back, hence slower than ArrayList.
* Addition and removal of elements is faster and easier.

class **Vector** implements List<E>

* Thread Safe implementation of a List.
* Implements Thread safety by using Synchronising methods.
* There will be a performance impact when we use Vector in a multithreaded scenario, because all the methods in Vector are Synchronised.

Iteration:

~~~~~~~~~

Accessing items in a collection is a very frequent and common operation. Let us take a look at various ways of traversing the elements in a collection.

1.Iterator interface is used for the subclasses of Collection interface

* .hasnext(),
* .next()

2.ListIterator interface :

* .hasnext(),
* .next(),
* .hasPrevious(),
* .previous()

2.Enhanced for loop / for-each is used for unordered/ordered collections

3.for loop is used for ordered collections ex: Lists

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

import java.util.ArrayList;

public class List{

public static void main(String[] args){

ArrayList<Integer> num = new ArrayList<Integer>();

num.add(10);

num.add(20);

num.add(30);

num.add(40);

num.add(50);

System.out.println(num+"\n");

System.out.println(num.get(0)+"\n");

System.out.println(num.size());

num.clear();

System.out.println(num+"\n");

System.out.println(num.isEmpty());

}

}

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

/\*ArrayLisyt Program\*/

~~~~~~~~~~~~~~~~~~~~~~

import java.util.ArrayList;

class Course {

String courseName;

Course(String courseName) {

this.courseName = courseName;

}

//override

public String toString() {

return courseName ;

}

}

public class List {

public static void main(String[] args) {

ArrayList<Course> courseList = new ArrayList<>();

courseList.add(new Course("Java"));

courseList.add(new Course("Hibernate"));

courseList.add(new Course("AngularJS"));

System.out.println(courseList);

//for loop

System.out.println("Using For Loop to list the Courses");

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

for(int index=0; index < courseList.size(); index++) {

System.out.println(courseList.get(index)+"\n");

}

//Enhanced Loop

// Can be read as: for each Course c in courseList

System.out.println("Using Enhanced Loop to list the Courses");

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

for(Course c : courseList) {

System.out.println(c+"\n");

}

}

}

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

/\*LinkedList Program with Iterator\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.List;

import java.util.Iterator;

import java.util.LinkedList;

class Course {

String courseName;

public Course(String courseName) {

this.courseName = courseName;

}

@Override

public String toString() {

return courseName;

}

}

public class ListInterface {

public static void main(String[] args) {

List<Course> courseList = new LinkedList<>();

courseList.add(new Course("Java"));

courseList.add(new Course("Hibernate"));

courseList.add(new Course("AngularJS"));

// Accessing the list of courses Using Iterator

Iterator<Course> courseIterator = courseList.iterator();

System.out.println("Using Iterator to access the list of courses");

while (courseIterator.hasNext()) {

Course c = courseIterator.next();

System.out.println(c); // toString() method has been overridden in the Course class

}

// Accessing the list of courses Using for loop

System.out.println("Using for loop to access the list of courses");

for (int index = 0; index < courseList.size(); index++) {

System.out.println(courseList.get(index));

}

// Accessing the list of courses Using enhanced for loop (for-each)

System.out.println("Using enhanced for loop to access the list of courses");

for (Course c : courseList) { // Can be read as: for each Course c in courseList

System.out.println(c);

}

}

}

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

/\*To understand ArrayList and ListIterator in Collection.

/\*Define an ArrayList of String and add any four names to it. Using ListIterator, traverse through the ArrayList first in forward and then in the reverse direction, and print the names to the console.\*/

import java.util.\*;

public class array{

public static void main(String[] args){

ArrayList<String> name = new ArrayList<String>();

name.add("Jenny");

name.add("Aura");

name.add("Josh");

name.add("kevin");

System.out.println("ArrayList" +name+"\n");

System.out.println("ArrayList in forward direction");

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

ListIterator<String> it = name.listIterator();

//Iterator<String> it = name.iterator();

//previous() cannot be used with Iterator interface.

//So using ListIterator interface.

while (it.hasNext()) {

System.out.println(it.next());

}

System.out.println("ArrayList in reverse direction");

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

while (it.hasPrevious()) {

System.out.println(it.previous());

}

}

}

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

/\*To understand LinkedList and Iterator in Collection.

/\*Define a LinkedList of String and add any four names to it.

Using 'for loop', traverse through the array and print the names to the console.Remove the first and the last element and print the list using 'advanced for loop'.Add new names at the first and last position and print the list using an iterator.\*/

import java.util.\*;

public class Linklist{

public static void main(String[] args){

LinkedList<String> li = new LinkedList<String>();

li.add("Jenny");

li.add("Jose");

li.add("Aura");

li.add("Kevin");

//Print using for loop

System.out.println("Printing the List using For Loop");

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

for(int i = 0;i<li.size();i++)

{

System.out.println(li.get(i));

}

//Remove first and Last element

li.removeFirst();

li.removeLast();

//Print them using enhanced for loop

System.out.println("Printing the List using Enhanced Loop");

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

for(String s : li)

{

System.out.println(s);

}

//Add two names in the first and last position

System.out.println("Printing the List using an Iterator");

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

li.addFirst("Jenny");

li.addLast("Kevin");

Iterator<String> it = li.iterator();

while(it.hasNext())

{

System.out.println(it.next());

}

}

}

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

Sets(Interface):

~~~~~~~~~~~~~~~

Interface Set<E> extends Collection<E>

* Sets are unordered collections which do not allow duplicates.
* HashSets are instances of HashSet, Set, Collection and Iterable interfaces.
* LinkedHashSets are instances of LinkedHashSet, HashSet, Set, Collection and Iterable interfaces.
* TreeSets are instances of TreeSet,SortedSet,Set,Collection and Iterable interfaces.

/\*List allows duplicate values.

Sometimes duplicate values is not desired.

so we go in for Sets\*/

The java.util.Set<E> represents a collection of unique item,i.e. duplicate data is not allowed.

\*Set has only the methods from the Collection interface.

\*Since sets are unordered, they can not be accessed using indexes.

\*The enhanced for loop and iterator helps to access the elements of a set.

\*Set eliminates duplicates.

\*The add() method will return false if our program attempts to add a duplicate element.

\*The equals () method returns true only when they have the same elements. Also, it works for all implementations of the Set interface.

1.HashSet - Uses Hash tables for Storing elements. No order of elements.

2.LinkedHashSet - Uses combination of Hash tables and Linked Lists for Storing elements.Elements are arranged in their order of insertion.

Also, LinkedHashSet maintains the order of insertion.

3.TreeSet - Uses Tree Based Structure for Storing elements. Elements are arranged in their natural ordering(Eg.Alphabetical for Strings,

Ascending for Numbers).

TreeSet does not accept null values because of its ordering.

Apart from preventing duplicates, if a set is required to have sorted elements, we can use a TreeSet.

\*Hash table is a data structure used to implement an associative array, a structure that can map keys to values.

\*A Tree is a non-linear data structure made up of nodes that form a hierarchy consisting of a root node and potentially many levels of additional nodes.

\* For sets to detect duplicates among user-defined objects, the equals() and hashCode() methods must be overridden.

/\*Sets - Example\*/

~~~~~~~~~~~~~~~~~

import java.util.\*;

public class Sets {

public static void main(String[] args) {

String[] carsArray = new String[] {"Volvo","BMW","Audi","Honda"};

Set<String> hashSet = new HashSet<>(Arrays.asList(carsArray));

Set<String> linkedhashSet = new LinkedHashSet<>(Arrays.asList(carsArray));

Set<String> treeSet = new TreeSet<>(Arrays.asList(carsArray));

System.out.println("HashSet - Random Order");

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

for(String name:hashSet){

System.out.println(name);

}

System.out.println("LinkedHashSet - Insertion Order");

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

System.out.println();

for(String name:linkedhashSet){

System.out.println(name);

}

System.out.println("TreeSetString - Natural Order");

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

for(String name:treeSet){

System.out.println(name);

}

Integer[] numArray = new Integer[] {100,8000,300,700,50};

Set<Integer> treeSet1 = new TreeSet<>(Arrays.asList(numArray));

System.out.println("TreeSetNumbers - Natural Order");

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

for(int i:treeSet1){

System.out.println(i);

}

}

}

/\*HashSet

~~~~~~~~~

import java.util.\*;

public class SetsMaps{

public static void main(String[] args){

HashSet<String> cars = new HashSet<>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Toyota");

System.out.println("Cars");

System.out.println(cars);

HashSet<String> bikes = new HashSet<>();

bikes.add("Yamaha");

bikes.add("BMW");

bikes.add("Honda");

bikes.add("Suzuki");

System.out.println("Bikes");

System.out.println(bikes);

//Union Operation

cars.addAll(bikes);

System.out.println("Cars Union Bikes");

System.out.println(cars);

//Intersection Operation

cars.retainAll(bikes);

System.out.println("Cars Intersection Bikes");

System.out.println(cars);

//Difference Operation

cars.removeAll(bikes);

System.out.println("Cars Difference Bikes");

System.out.println(cars);

}

}

/\*Tree Set and LinkedHashSet

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.\*;

public class Set {

public static void main(String[] args) {

TreeSet<String> courseSet = new TreeSet<>();

TreeSet<Integer> ts = new TreeSet<>();

LinkedHashSet<Integer> numberSet = new LinkedHashSet<>();

// Adding elements to the set

numberSet.add(12);

numberSet.add(24);

numberSet.add(12);

// Displaying the Set

System.out.println(numberSet);

// Adding elements to the String Set

courseSet.add("Java");

courseSet.add("Hibernate");

courseSet.add("Angular JS");

// Adding elements to the Integer TreeSet

ts.add(20);

ts.add(21);

ts.add(22);

ts.add(20);

ts.add(23);

System.out.println(ts);

System.out.println(ts.ceiling(21));

System.out.println(ts.floor(22));

System.out.println(courseSet.first());

System.out.println(courseSet.last());

// Iterating over the Sting set using enhanced for loop

System.out.println("Printing using Enhanced loop");

for(String s: courseSet) {

System.out.println(s);

}

}

}

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

/\*Create a Java program that adds Student objects into a HashSet. The Student class has name and rollNumber as its attributes and a toString() method. If we add two Student objects having the same rollNumber to the HashSet, it should be considered as duplicate and should not get added.\*/

import java.util.\*;

class Student {

private String name;

private int rollNo;

Student(String name,int rollNo) {

this.name = name;

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

@Override

public String toString() {

return name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if ((obj == null) || (getClass() != obj.getClass())) {

return false;

}

/\*if (getClass() != obj.getClass()) {

return false;

}\*/

Student student = (Student) obj;

return (student.rollNo == this.rollNo);

}

@Override

public int hashCode() {

// return Objects.hash(rollNo);

return rollNo;

}

}

public class StudentDetails {

public static void main(String[] args) {

HashSet<Student> hs = new HashSet<>();

hs.add(new Student("Kevin",100));

hs.add(new Student("Jenny",101));

hs.add(new Student("Jenny",100));

System.out.println(hs);

}

}

/\*Program to remove duplicates\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.\*;

public class RemoveDuplicates{

public static void main(String[] args){

ArrayList<Integer> numbersList = new ArrayList<>(Arrays.asList(1, 1, 2, 3, 3, 3, 4, 5, 6, 6, 6, 7, 8));

System.out.println(numbersList);

LinkedHashSet<Integer> hashSet = new LinkedHashSet<>(numbersList);

ArrayList<Integer> listWithoutDuplicates = new ArrayList<>(hashSet);

System.out.println(listWithoutDuplicates);

}

}

Map:

~~~

\*Map is a container that contains mappings from keys to Values.

\*HashMap is an instance of HashMap and Map.

\*LinkedHashMap is an instance of LinkedHashMap,HashMap and Map.

\*TreeMap is an instance of TreeMap,SortedMpap,Map.

\*HashMap is not an instance of Collection and Iterable interfaces.

\*There are times when we need to maintain data corresponding to some other data like associating words and their meanings in a dictionary, employees with their employee numbers, and so on.

\*To store such data, which needs to exist in pairs, Java provides us with Maps.

\* In Map, if we try to insert multiple values in the same key, the previous value is replaced.

\*java.util.Map<K,V> represents a collection that allows the mapping of keys and values to form key-value pairs.

\*Values in a map can be duplicates, but keys have to be unique.

Having unique keys allow easier and faster access to the values.

\*Keys are displayed in an order in a TreeMap, SO it does not accept a null value in a key.

\*But in HashMap and LinkedHashMap, since they dont display the elements in any particular order,null is accepted in a key.

\*Map interface does not extend the Collection interface. Hence, there is no iterator for maps.

\*Also, map values cannot be accessed without keys. So a map cannot be traversed directly.

\*To overcome this, Map provides us with methods to retrieve a Collection that we can traverse.

Set setOfKeys = map.keySet();

Collection valueCollection = map.values();

Set<Entry> setOfEntries = map.entrySet();

The java.util.Map.Entry interface provides two useful methods:

1.getKey()

2.getValue()

/\*Program to learn about Maps\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.\*;

public class LearningMaps {

public static void main(String[] args) {

Map<Integer,String> studMarks = new HashMap<Integer,String>();

System.out.println(studMarks.isEmpty());

System.out.println(studMarks.size());

//Adding Elements

studMarks.put(100,"Jenny");

studMarks.put(101,"Jose");

studMarks.put(102,"Kevin");

studMarks.put(103,"Aura");

System.out.println(studMarks.isEmpty());

System.out.println(studMarks.size());

System.out.println(studMarks+"\n");

//Retrieving elements usinf format statement

System.out.format("Key: %s,Value : %s\n",100,studMarks.get(100));

//Trying to retrieve element which is not in there

System.out.format("Key:%s, Value: %s\n",108,studMarks.get(108));

//Deleting elements

studMarks.remove(100);

System.out.println(studMarks+"\n");

//Updating elements

studMarks.put(100,"Jose");

studMarks.put(101,"Jenni");

System.out.println(studMarks+"\n");

}

}

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

Map(Interface):

~~~~~~~~~~~~~~

1.HashMap - don't guarantee the order of entries. Displays in a random order.

2.TreeMap- Entries are arranged according to the natural ordering of the keys.

3.LinkedHashMap - Maintains the entries in the order of insertion.

4.ConcurrentHashMap - concurrent update and retrieval of data.

TreeMap & ConcurrentHashMap cannot contain null keys.

import java.util.\*;

class Course {

private String courseName;

Course(String courseName) {

this.courseName = courseName;

}

@Override

public String toString() {

return courseName;

}

}

public class MapInterface {

public static void main(String[] args) {

HashSet<Course> courseSet1 = new HashSet<>();

courseSet1.add(new Course("Java"));

courseSet1.add(new Course("DBMS"));

courseSet1.add(new Course("Python"));

HashSet<Course> courseSet2 = new HashSet<>();

courseSet2.add(new Course("HTML"));

courseSet2.add(new Course("CSS"));

courseSet2.add(new Course("JavaScript"));

HashMap<Integer,Set<Course>> student = new HashMap<>();

student.put(1001,courseSet1);

student.put(1002,courseSet2);

// Retrieving the set of Courses by studentID using get() method

Set<Course> courseSet3 = student.get(1001);

System.out.println(courseSet3);

Set<Course> courseSet4 = student.get(1002);

System.out.println(courseSet4);

// Iterating over the set of keys using for-each loop

Set<Integer> setOfKeys = student.keySet();

for(Integer i : setOfKeys) {

System.out.println(student.get(i));

}

// Iterating over the collection using values() method

for(Set<Course> courses : student.values()) {

System.out.println(courses);

}

}

}

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

/\*Program to understand EntrySet, Keys,Values\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.\*;

public class MapCollection {

public static void main(String[] args) {

Map<Integer,String> courseMap = new HashMap<Integer,String>();

courseMap.put(101,"Java");

courseMap.put(106,"Java");

courseMap.put(103,"Python");

courseMap.put(102,"HTML");

courseMap.put(105,"JavaScript");

courseMap.put(104,"CSS");

//Any number of nulls are allowed in values

courseMap.put(107,null);

//Only one key can be null as keys are unique

courseMap.put(null,"DBMS");

//Set Representation of Key Value pairs in a Map

Set<Map.Entry<Integer,String>> entrySet = courseMap.entrySet();

//Iterating over the Entry Set

for(Map.Entry<Integer,String> entry : entrySet) {

System.out.format("Key : %s,Value: %s\n",entry.getKey(), entry.getValue());

}

//Updating the value of a key using for loop

for(Map.Entry<Integer,String> entry : entrySet){

if (entry.getKey() == 103){

entry.setValue("C++");

}

}

System.out.println(entrySet+"\n");

Set<Integer> keySet = courseMap.keySet();

System.out.println("Set view of Keys:(No Duplicates)" +keySet+"\n");

for(Integer key : keySet){

System.out.println(key + "-" +courseMap.get(key));

}

//Collection view of values

Collection<String> values = courseMap.values();

System.out.println("Collection view of Values:(May contain duplicates)" +values+"\n");

}

}

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

Usually Tree Map displays the result in ascending Order. To change the order we use comparator method:

/\*Number Comparator operator in TreeMap\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.\*;

public class comparatorTreeMap{

public static void main(String[] args){

Comparator<Integer> descendComparator = new Comparator<Integer>(){

@Override

public int compare(Integer i1, Integer i2){

return i1.compareTo(i2)\* -1;

}

};

SortedMap<Integer,String> treeMap = new TreeMap<>(descendComparator);

treeMap.put(1067,"Jenny");

treeMap.put(676,"Kate");

treeMap.put(998,"Kevin");

treeMap.put(1020,"Josh");

treeMap.put(999,"Aura");

System.out.println("Entries in the Reverse Order:");

for(Map.Entry<Integer,String> entry : treeMap.entrySet()){

System.out.println(entry);

}

}

}

/\*String comparator operator in TreeMap\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.\*;

public class comparatorTreeMap{

public static void main(String[] args){

Comparator<String> descendStringComparator = new Comparator<String>(){

@Override

public int compare(String s1, String s2){

return s1.compareTo(s2)\* -1;

}

};

SortedMap<String,String> treeMap = new TreeMap<String,String>(descendStringComparator);

treeMap.put("Java","A");

treeMap.put("DBMS","Z");

treeMap.put("Python","C");

treeMap.put("C Sharp","D");

treeMap.put("SQL","B");

System.out.println("Entries in the Reverse Order:");

for(Map.Entry<String,String> entry : treeMap.entrySet()){

System.out.println(entry);

}

}

}

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

/\*Program to compare length of the string and display it in order\*/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import java.util.\*;

public class comparatorTreeMap{

public static void main(String[] args){

Comparator<String> stringLengthComparator = new Comparator<String>(){

@Override

public int compare(String s1, String s2){

return s1.length()-s2.length();

}

};

SortedMap<String,String> treeMap = new TreeMap<String,String>(stringLengthComparator);

treeMap.put("Java","ABC");

treeMap.put("DBMSystems","XZ");

treeMap.put("Python","CDE");

treeMap.put("C Sharp","DEF");

treeMap.put("SQL","BDEF");

System.out.println("Entries in the Order of their length:");

for(Map.Entry<String,String> entry : treeMap.entrySet()){

System.out.println(entry);

}

}

}

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

Choosing the right Collection:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Choose the right collection based on the requirement to improve performance.

Some of the points to be considered are:

1.If duplicate elements are not allowed to choose Set otherwise choose List.

2.ArrayList is quicker than LinkedList to randomly access elements.

3.For quick removal and addition, LinkedList is better than ArrayList.

4.Use collections such as TreeMap, TreeSet when elements in the collection are required to be sorted and ordered.

5.Use concurrent collections to support concurrent access.

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

import java.util.\*;

public class StudentMain{

public static void main(String[] args){

Map<Integer,Student> studentsMap = new HashMap<>();

System.out.println("Values can be Custom Objects\n");

studentsMap.put(1, new Student("Jenny",100,98));

studentsMap.put(2, new Student("Kevin",101,89));

studentsMap.put(3, new Student("Aura",102,67));

studentsMap.put(4, new Student("Josh",103,86));

studentsMap.put(5, new Student("John",104,75));

studentsMap.put(6, new Student("John",104,75));

for(Map.Entry<Integer, Student> entry : studentsMap.entrySet()){

System.out.format("Key: %s, Value: %s\n",entry.getKey(),entry.getValue());

}

}

}

class Student {

private String name;

private int rollNo;

private int totalMarks;

Student(String name,int rollNo,int totalMarks) {

this.name = name;

this.rollNo = rollNo;

this.totalMarks = totalMarks;

}

public String getName() {

return name;

}

public int getRollNo() {

return rollNo;

}

public int getTotalMarks() {

return totalMarks;

}

@Override

public String toString(){

return name;

}

@Override

public int hashCode(){

return Objects.hash(name);

}

@Override

public boolean equals(Object obj) {

if ((obj == null) || (getClass() != obj.getClass())) {

return false;

}

/\*if (getClass() != obj.getClass()) {

return false;

}\*/

Student student = (Student) obj;

if (student.name.equals(this.name)){

return true;

}

//return (student.name == this.name);

return false;

}

}

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

import java.util.\*;

public class StudentMain{

public static void main(String[] args){

Map<Integer,Student> studentsMap = new HashMap<>();

System.out.println("Values can be Custom Objects\n");

studentsMap.put(1, new Student("Jenny",100,98));

studentsMap.put(2, new Student("Kevin",101,89));

studentsMap.put(3, new Student("Aura",102,67));

studentsMap.put(4, new Student("Josh",103,86));

studentsMap.put(5, new Student("John",104,75));

studentsMap.put(6, new Student("John",104,75));

for(Map.Entry<Integer, Student> entry : studentsMap.entrySet()){

System.out.format("Key: %s, Value: %s\n",entry.getKey(),entry.getValue());

}

}

}

class Student {

private String name;

private int rollNo;

private int totalMarks;

private char grade;

Student(String name,int rollNo,int totalMarks) {

this.name = name;

this.rollNo = rollNo;

this.totalMarks = totalMarks;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

public int getTotalMarks() {

return totalMarks;

}

public void setTotalMarks(int totalMarks) {

this.totalMarks = totalMarks;

}

public char calculateGrade(){

if (totalMarks>=60)

return 'A';

else if(totalMarks<60 && totalMarks>40)

return 'B';

else if(totalMarks<=40)

return 'C';

else

return 0;

}

@Override

public String toString() {

return name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if ((obj == null) || (getClass() != obj.getClass())) {

return false;

}

Student student = (Student) obj;

return (student.rollNo == this.rollNo);

}

@Override

public int hashCode() {

// return Objects.hash(rollNo);

return rollNo;

}

}

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