Web Full Stack

Softwares required

1. Java 8 - JDK & JRE
2. Eclipse IDE for Enterprise Developer - (2020-06) 0R (2020-09)

Once you install java software you need to set the environment variable in your personal machine. So that javac will be recognized.

Java

Java is a platform independent and Object oriented programming language.

Platform Independence: It can run on any platforms without altering.

Object Oriented: An application will have realworld entities which are called as object, which will be created to make application work, one object calls another object behaviour to complete their task.

ex: Customer object can have deposit(), that can call Account object credit() functionality.

Two main building blocks of Java

1. Classes - Blue print of an object (template for an object)
2. Objects - Instances of a class

A class can be created inside a Java source file with an extension .java, a class can have variables, methods & constructors, all these should follow some standard naming conventions.

Filenames, classnames & constructors: Should begin with Capital letters & Follow camel case

variable names & method names: Should begin with Small letters & Follow camel case.

Commands we use to compile & run java program

1. javac: to compile
   1. ex: javac Filename.java
2. java: to run
   1. ex: java ClassnameHavingMainMethods

Note: In java main method signature is ‘public static void main(String[] args)’

public, static & void are keywords the java can recognize

main is a method name

String is predefined class

HelloWorld.java



Output:



Using Eclipse IDE

Eclipse is a tool to develop applications quickly, it gives lot of suggestions to the programmers so that they can quickly write programs.

Eclipse will have all the projects in the workspace, you have different perspectives like

* Java (Java applications)
* JavaEE - Default (servlets & JSP)

Packages: They are used to categorize your classes, they are like folders

Their names would be like a domain name ex: com.hsbc, com.google, com.oracle

HelloWorld.java



Java Fundamentals

1. Operators
2. Keywords
3. Datatypes
4. Variables
5. Conditional Statements
6. Loops
7. Arrays
8. Methods
9. Classes & Objects

Operators:

+, -, \*, /, =, ==, <=, >=, !=, &&, ||, ++, --, +=, -=, /=, \*=

Keywords: These are reserved words in java

public, package, static, void, int, long, float, double, char, byte, short, if, for, while, do, switch, case, default, break, return, continue, final, interface, class, enum, try, catch, finally, extends, implements, super, this, throw, throws, synchronized, abstract, volatile, native,

Datatypes

1. Primitive types: keywords in java and fixed in size
   1. byte, short, int, long - Integer types: whole numbers
   2. float & double - Floating types: real numbers
   3. char - Character types: single char
   4. boolean - Boolean true or false
2. Non-Primitive types: classes, interfaces, arrays, enum - not fixed in size
   1. String, Employee, Customer, Student, StringBuilder
   2. int[], float[], double[]

Variables: These can store data of specific type

Syntax: type variable\_name = value;

ex: int employeeId = 100;

ex: double salary = 25000;

TestFundamentals1.java



Output:



Passing data to a method having parameter



Conditional Statements

1. if
2. if-else
3. if else-if else-if….
4. if else-if else-if…. else
5. switch

Taking input from keyboard



Output:



How to create objects

Customer.java



TestCustomer.java



Customer.java



TestCustomer.java



Constructors take care of initializing the objects property.

Some important points on constructor

1. Compiler creates constructor if there’s not constructor inside the class
2. If User has created the constructor then compiler retains that constructor and doesn’t create any constructor, user may create default or parameterized constructor

OOPS features

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Encapsulation: Hiding the data by making variables private and the only way you can access them is through public methods like setters or getters

Student.java



TestStudent.java



Fundamentals of Java

* Operators
* Datatypes
* Variables
* Methods
* Classes & Object

Day 2:

Conditional Statements

Loops

Arrays

Object Arrays

Constructors

String & String methods

Inheritance

Gmail -> User(username, phone, password, gender,….)

Gmail -> username -> getUsername(), setPhone(long), getPhone(), setPassword(String), getPassword(), getGender(), setGender(String)

Employee -> id, name, age, salary, desig, setAge(int) { age >= 18}

Conditional Statements

1. if
2. if - else
3. if - else if else if….
4. if - else if else if….else
5. swtich

TestConditions



TestConditional2



switch:

switch also works like if else if .. else, but it only uses integer, string, character & enum



Loops:

1. for : when you know how many times you want to iterate
2. while: when you want to iterate until the condition is true
3. do-while: same like while but it is an exit control loop, at least once the statement inside the do-while will be executed

TestLoops.java



Arrays:

It is a container which can store more than one elements of same type, you can create simple arrays to complex type of arrays

1. Simple arrays:
   1. int[] items = {3, 1, 2, 6, 5};
   2. String[] values = {“Alex”, “Raj”, “Vijay”};
   3. int[] items = new int[5];
      1. items[0] = 1, items[1] = 5, items[2] = 10
2. Complex arrays: Array of complex types i.e., complex objects
   1. Customer[] customers = new Customer[5]; creates memory to store 5 customer objects
      1. customers[0] = new Customer(1, “Alex”, 50);
      2. customer[1] = new Customer(2, “Bruce”, 45);
      3. customer[4] = new Customer(5, “David”, 22);

You can create different dimensional arrays like one dimensional array & two dimensional array.

1. One dimensional array: int[] arr = new int[5]; // memory is allocated for 5 int values
2. Two dimensional array: int[][] arr = new int[5][4]; // memory is allocated for 20 int value
   1. int[0][0] = 15; // 0th row & 0th column
   2. int[1][1] = 10; // 1st row & 1st column

TestOneDArray.java



Creating complex type array

Customer[] customers = new Customer[5];

Note: each index will have null by default

customers is an array of size 5, it means in each index you can store customer object and maximum you can store 5 customer objects

Customer.java



TestCustomerArray.java



Output:



Taking value of customers dynamically

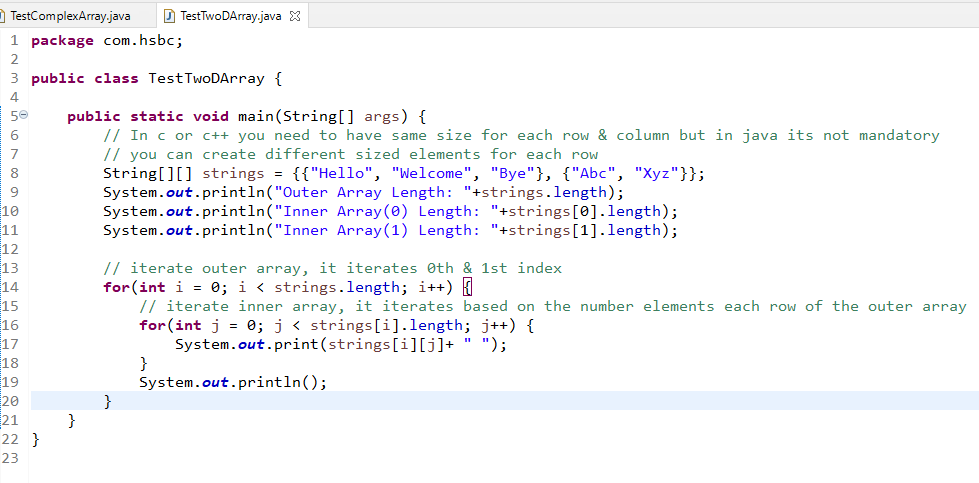


Two-dimensional array

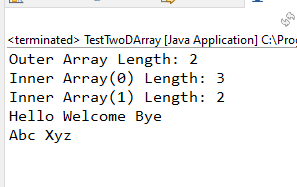
1. type[][] variable\_name = new type[row][col];
2. type[][] variable\_name = {{v1, v2, v3, v4}, {v5, v6, v7},…};

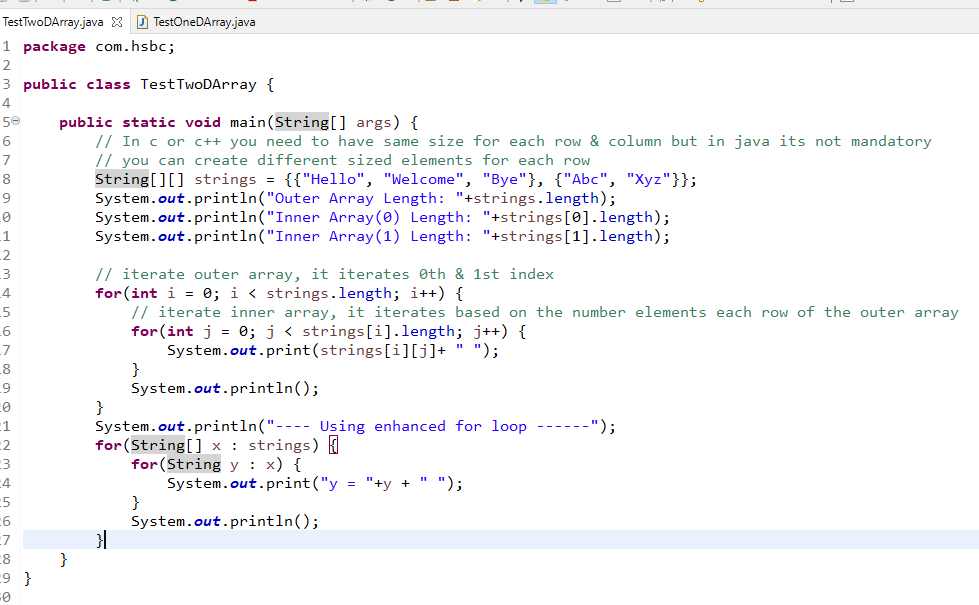
To iterate two-dimensional array you need to two loops i.e., one nested within another

You can use both traditional for loop if in case you want to do any changes if in case you are only retrieving you can use for-each

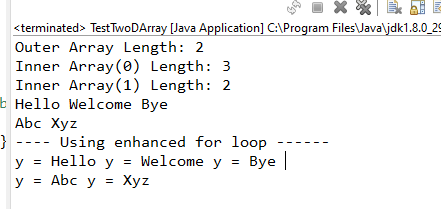


Output:





Output:



We have lot of predefined classes in java, we need instance of those classes to access the methods, some methods you can directly if they are static methods.

class Abc {   
 void test1() { …. }

static void test2() { …. }   
}

// test2() can be accessed using Abc.test2();

// test1() can be accessed by using reference of instance

Abc a = new Abc();

a.test1();

String class in an inbuilt class which has methods that are very useful, some of the methods are:

equals()

length()

equalsIgnoreCase()

toUpperCase()

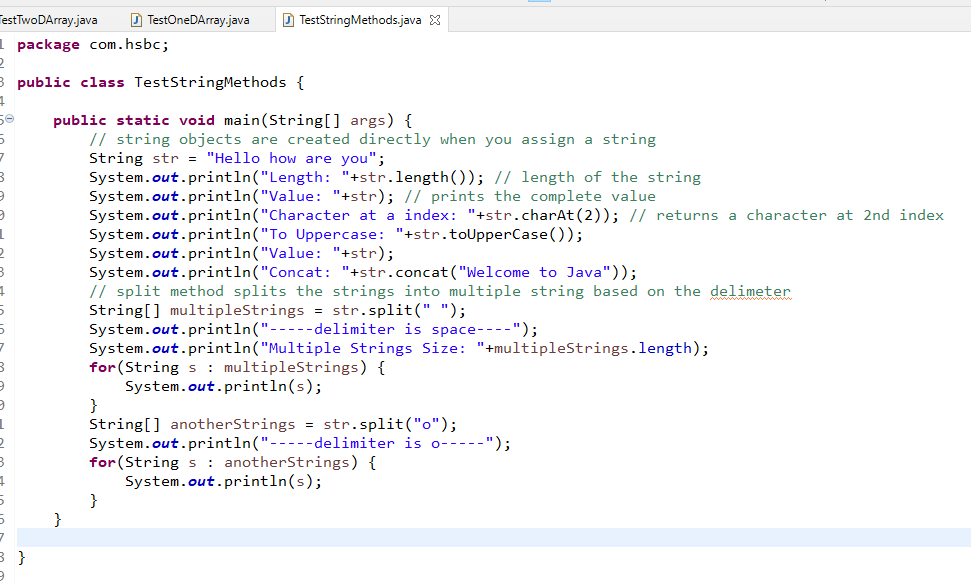
toLowerCase()

charAt()

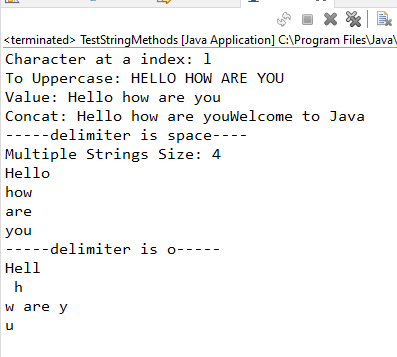
indexOf()

lastIndexOf()

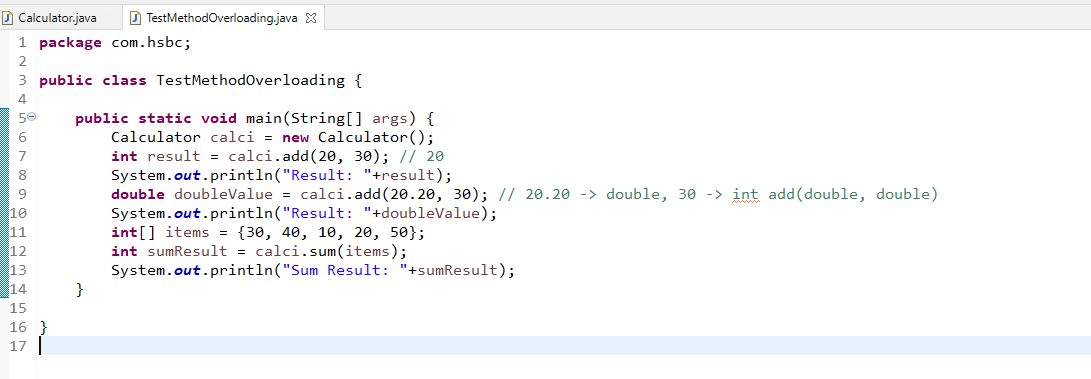
split()

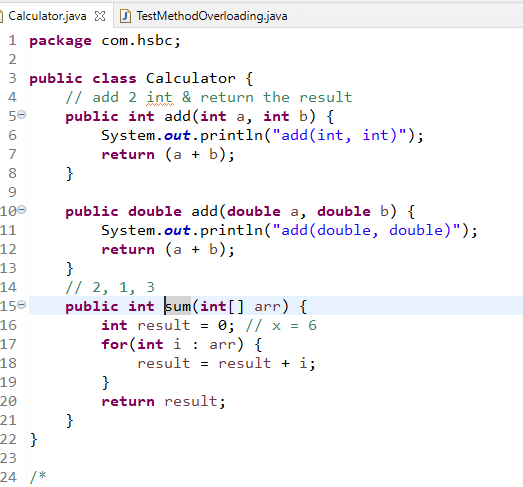


Output:

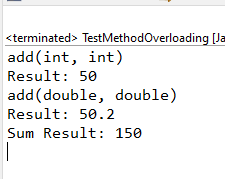


TestMethodOverloading





Output:



this keyword

It refers to current object, it differentiates between the instance variable & parameter variable when the variable is sharing same name.

Another use of this is it is used call current class constructor, to reuse the initialization statement.

Employee.java

**package** com.hsbc;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** String gender;

**private** **double** salary;

**private** String email;

**private** **long** phone;

// creating the constructor when id & name is mandatory

**public** Employee(**int** id, String name) {

**this**.id = id;

**this**.name = name;

System.***out***.println("Employee(id, name)");

}

// creating the constructor when certain fields are mandatory

/\*

\* this() -> Employee()

\* this(int, String) -> Employee(int, String)

\* this(int, String, double) -> Employee(int, String, double)

\* this(int, String, String, double, String, long) -> Employee(int, String, String, double, String, long)

\*

\*/

**public** Employee(**int** id, String name, **double** salary) {

**this**(id, name); // Employee(int, String)

**this**.salary = salary;

System.***out***.println("Employee(id, name, salary)");

}

// you can create constructor to initialize any number of properties

**public** Employee(**int** id, String name, String gender, **double** salary, String email, **long** phone) {

**this**(id, name, salary); // Employee(int, String, double);

**this**.gender = gender;

**this**.email = email;

**this**.phone = phone;

System.***out***.println("Employee(id, name, gender, salary, email, phone)");

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** **long** getPhone() {

**return** phone;

}

**public** **void** setPhone(**long** phone) {

**this**.phone = phone;

}

**public** **double** getSalary() {

**return** salary;

}

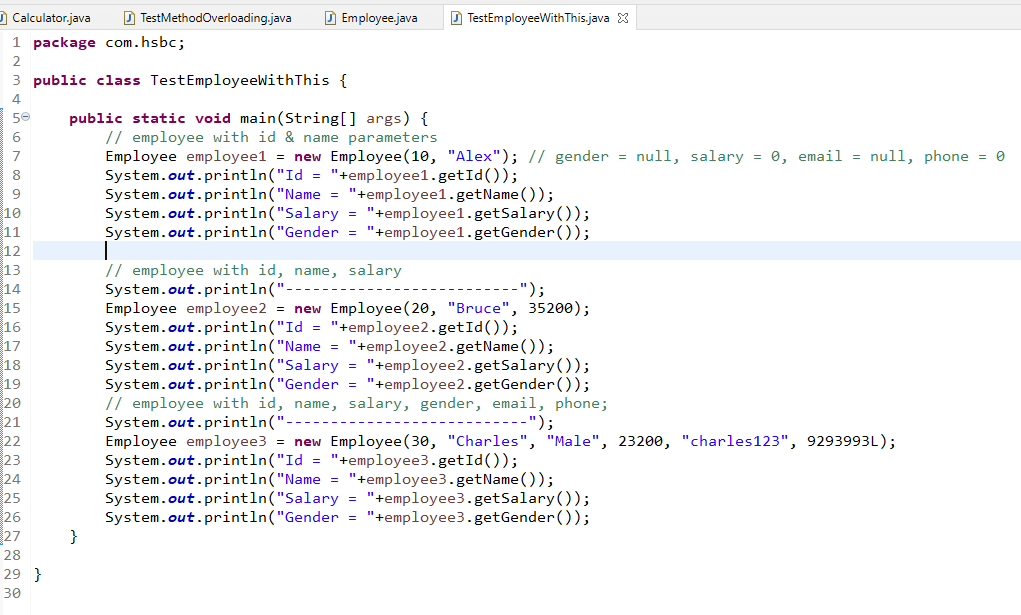
**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

}

TestEmployeeWithThis.java



Output:

Employee(id, name)

Id = 10

Name = Alex

Salary = 0.0

Gender = null

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

Employee(id, name)

Employee(id, name, salary)

Id = 20

Name = Bruce

Salary = 35200.0

Gender = null

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

Employee(id, name)

Employee(id, name, salary)

Employee(id, name, gender, salary, email, phone)

Id = 30

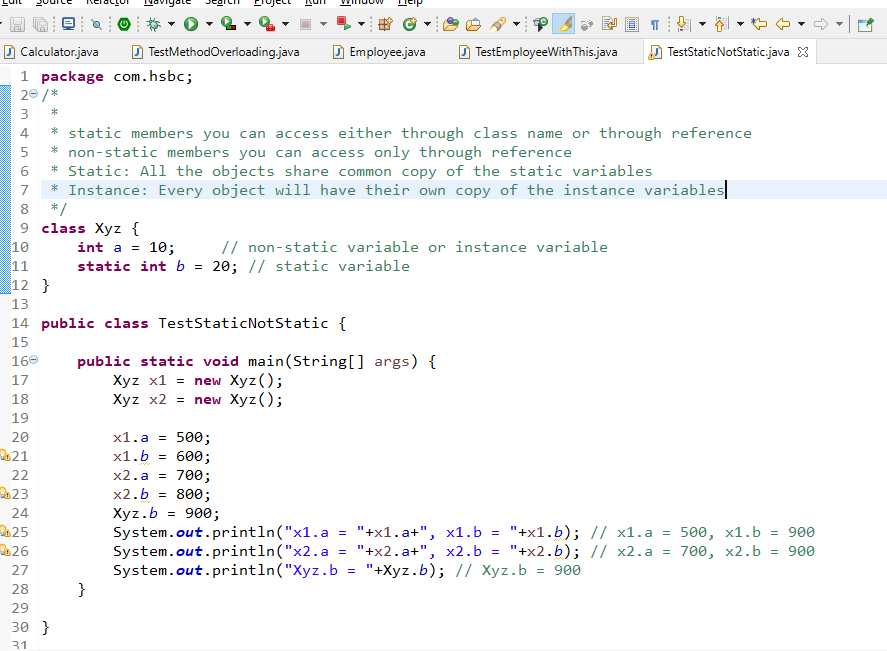
Name = Charles

Salary = 23200.0

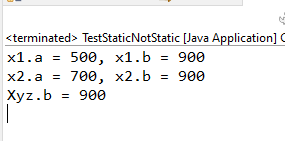
Gender = Male

Note: call to the constructor using this() must always be in the very first line of the constructor

Static & Non-Static (Instance)



Output:



Inheritance:

A process of acquiring common properties & behaviours of an object from another object, you can create common properties & behaviours in the super class & inherit to the subclass.

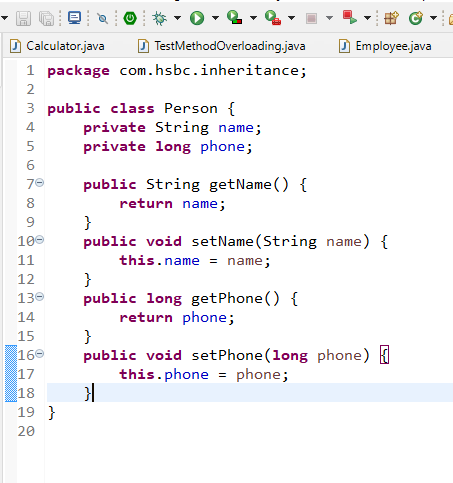
Java allows you to achieve inheritance using extends keyword.

class A { }

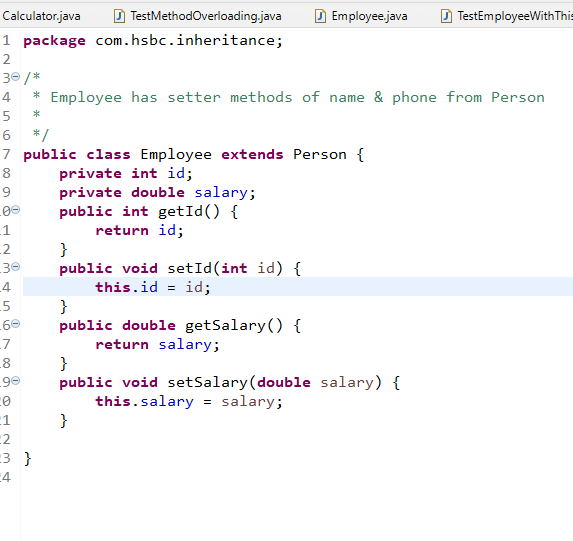
class B extends A { }

Note: private members & constructors wouldn’t be inherited

Person.java



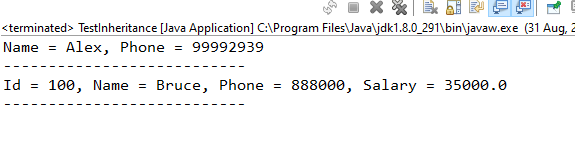
Employee.java



TestInheritance.java

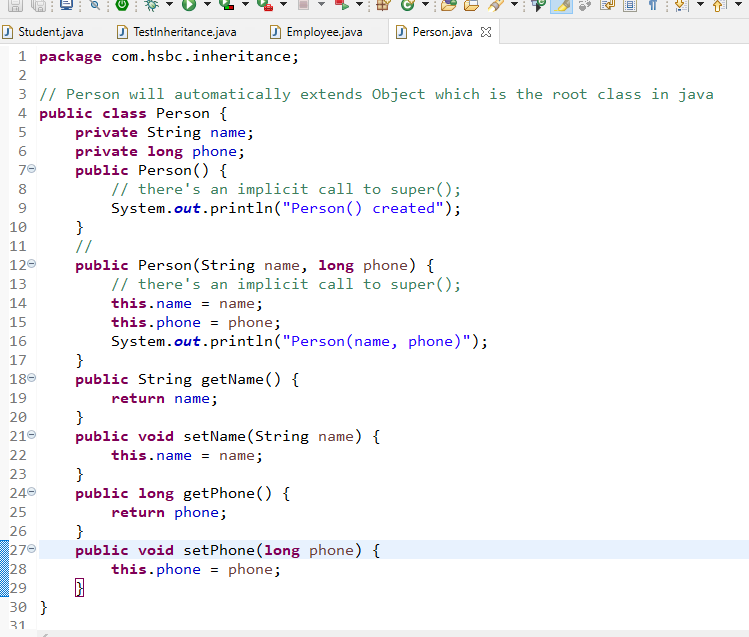


Output:

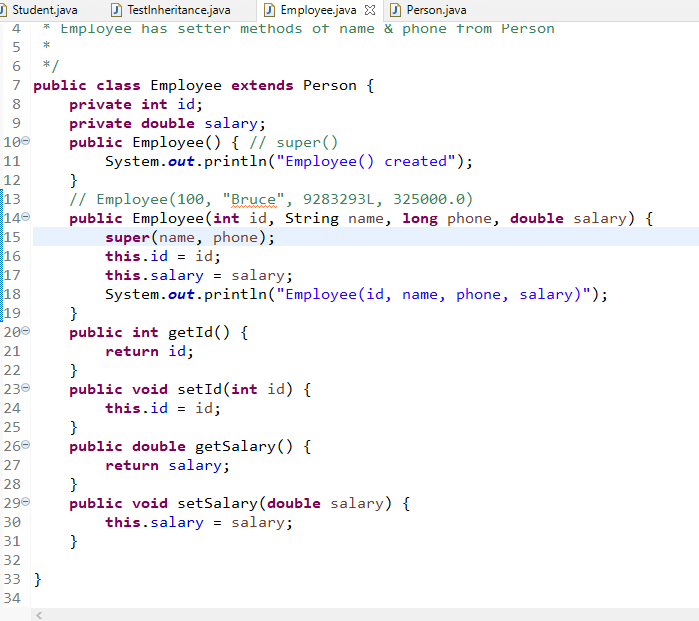


Day 3 - 01-09-2021

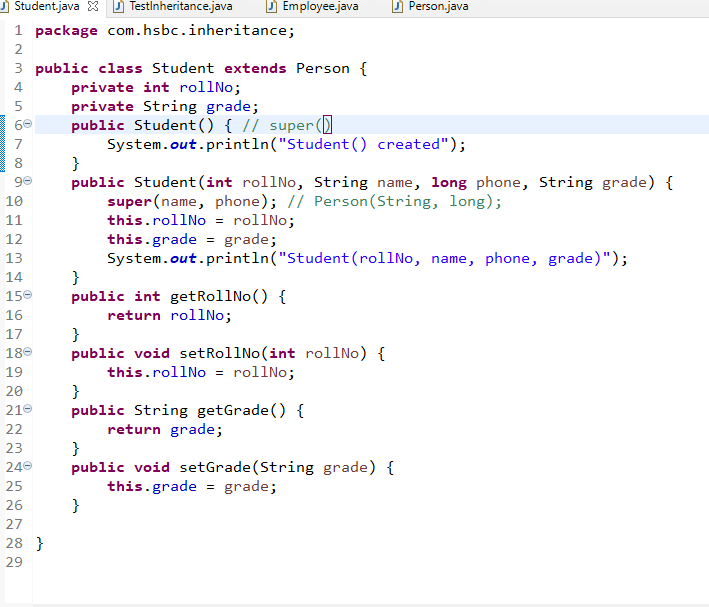
Person.java



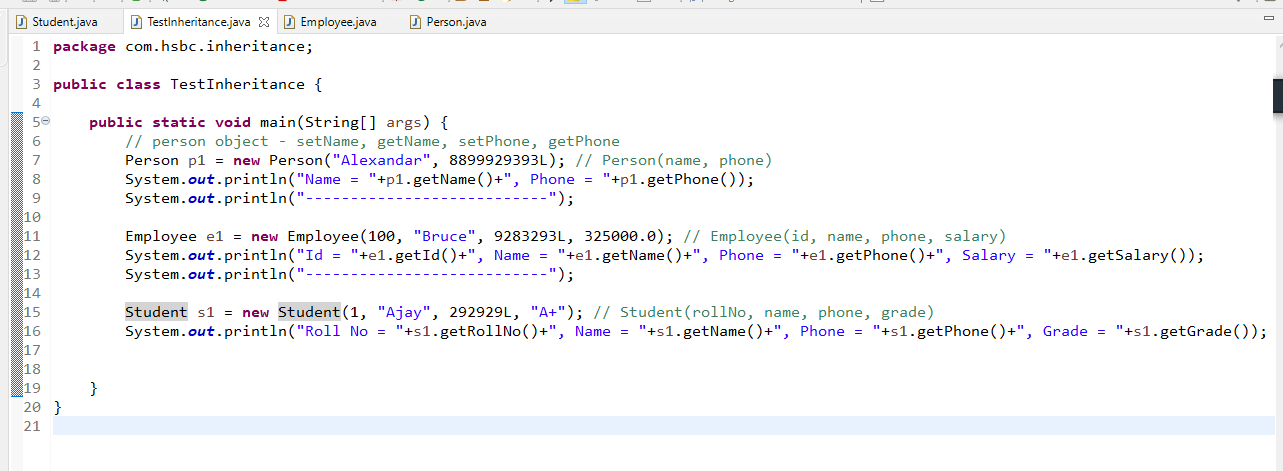
Employee.java



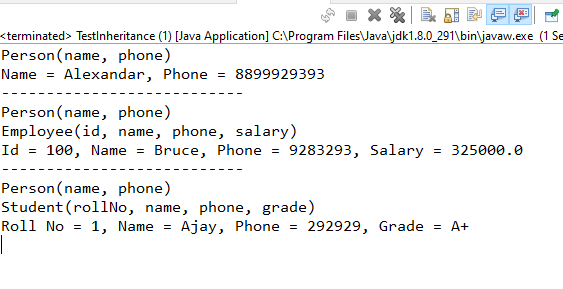
Student.java



TestInheritance.java



Output:



Note: If a class doesn’t have any super class then it automatically extends Object class which is the root class in java

Some of the important methods of Object class are:

1. toString(): called when you print the object, it returns memory address by default in hexadecimal format
2. equals(): used to compare two objects
3. hashCode(): returns a unique id of the object which is called as hashCode of the object.

Polymorphism

An object with many forms i.e., multiple methods of the same name but performs different action, (or) A method which performs different action based on the object

ex: power button has different behaviours like on/off both

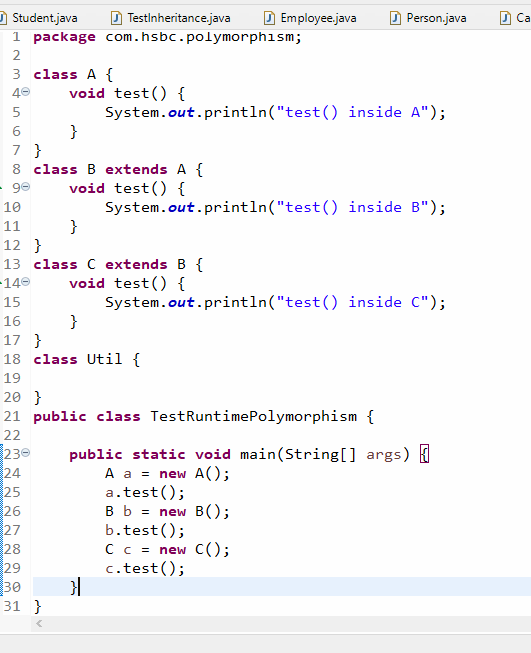
There are two types of polymorphism

1. Compile time polymorphism: method overloading: You can predict which method gets called at compilation
2. Runtime polymorphism: method overriding: You can’t predict which method gets called at compilation time, because the method execution depends on the object that is calling

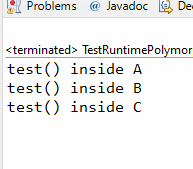
Method Overloading: Same method name but different signature(type of parameters, number of parameters) in the same class

Method Overriding: Same method name & the signature written in subclass with different logics particular to the subclass.

Simple polymorphism program



Output:



Note: You can assign multiple instances to the same reference, however the reference type should be superclass type

A a = new A(); B b = new B(); C c = new C();

A obj;

obj = a; // obj is of A type and a is of A type

obj = b; // obj is of A type and b is of B type

obj = c; // obj is of A type and c is of C type

This is possible because higher types can always take lower types

ex:

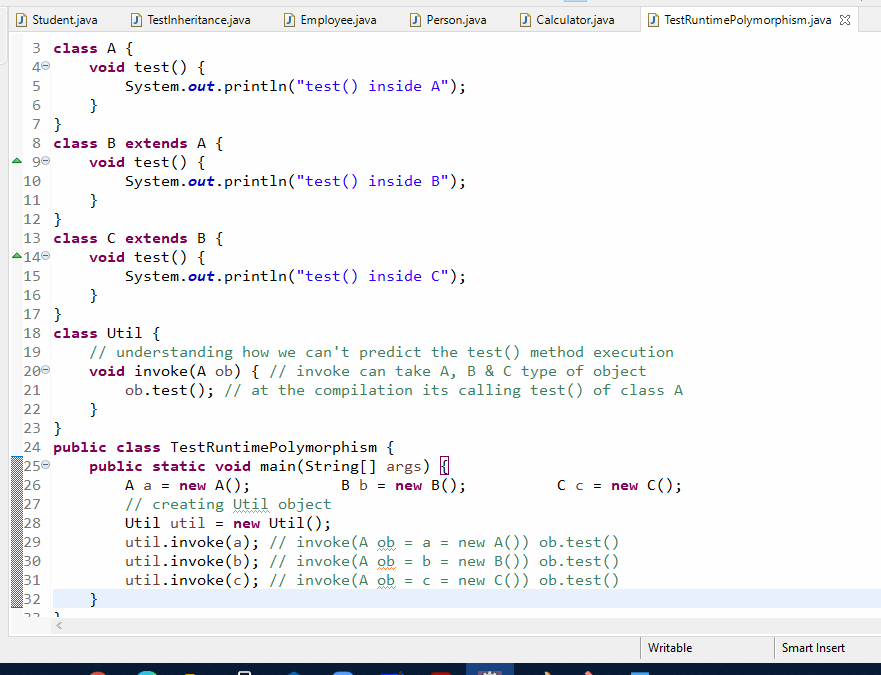
long x ;

int y = 30; short z = 50; byte t = 60;

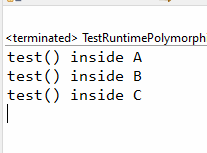
x = y; // x is long type and y is int type

x = z; // x is long type and z is short type

x = t; // x is long type and t is byte type



Output:



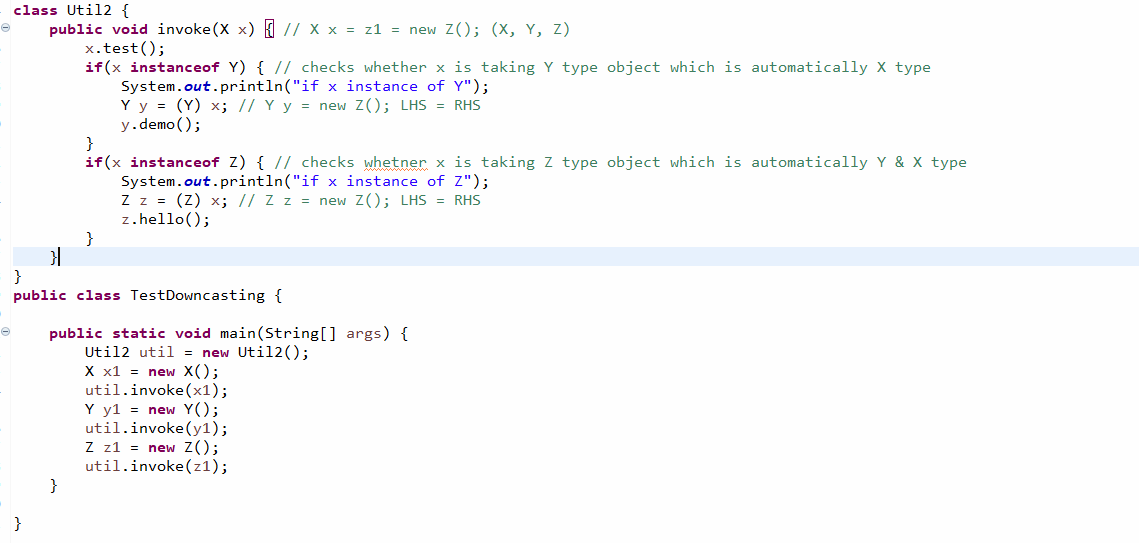
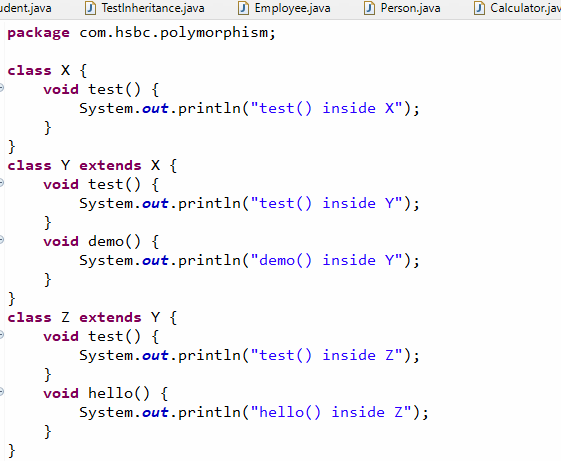
When you have a super class reference only overridden methods can be called as they are defined from super class, but you can’t access subclass members with super class reference, inorder to access subclass members you need to do a typecasting i.e., explicit downcasting.

A a = new B();

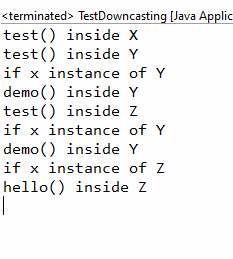
‘a’ can access only members of class A, if a wants to access members of class B you need to explicit downcast

B b = (B) a; // but you need to be cautious, because it may generate class cast exception, to avoid that you can use instanceof keyword

TestDowncasting.java



Output:



Abstraction: It hides the complexity from the user & shows only the necessary details so that user can work on the application or object easily.

Here in OOPS the complexities are implementations of the methods and showing only the necessary details means showing only the methods signatures.

It makes developer to think about what the method does instead of how the method does

Ex:

Developer1: store(User user) { …. connecting to DB … } 1 or -1

Developer2: register() -> UI -> reads user data can calls store(user); 1 or -1

How to achieve abstraction

It can be achieved in two ways

1. interface: complete abstract: all the methods are abstract
2. abstract class: partial abstract: some methods can be abstract & some are not

abstract means incomplete methods i.e., methods without body i.e., method declarations.

Interface:

It is like a class but will have only abstract methods & constants, it will not have constructors or complete methods

How to create interface

public interface interface\_name {   
 return\_type method\_name(arguments);  
 return\_type methods\_name(arguments);  
}

Ex:

interface Vehicle {  
 void mileage();  
 void wheels();  
}

abstract class Car implements Vehicle { } // implements wheels()

class Innova extends Car { } // implements milage()

class Swift extends Car { } // implements mileage()

abstract class Bike implements Vehicle {} // wheels()

class KTM extends Bike { } // implements mileage()

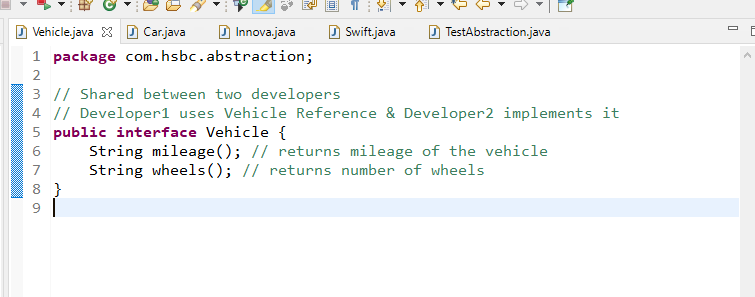
class Pulsar extends Bike { } // implements mileage()

Now all the above classes needs to mandatorily implement mileage & wheels methods.

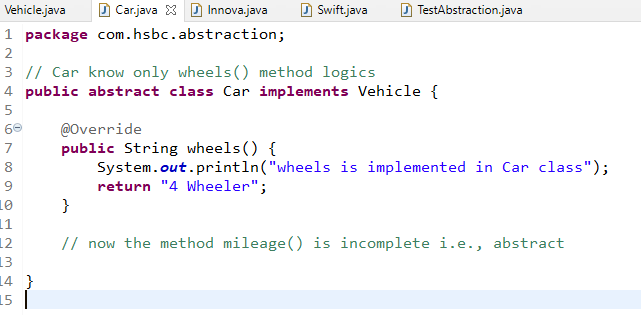
Rule:

If you have an interface you can only create abstract methods, a class needs to mandatorily override all the abstract methods else class should be made abstract.

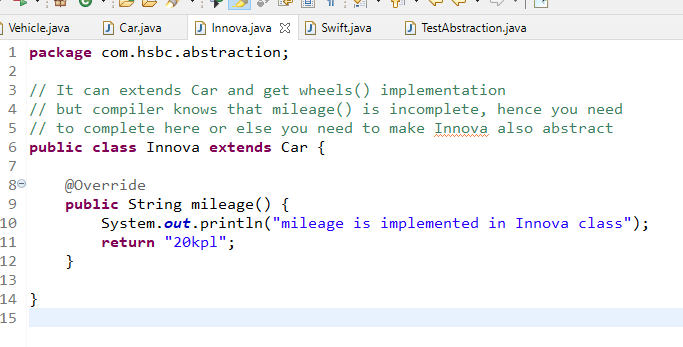
Vehicle.java



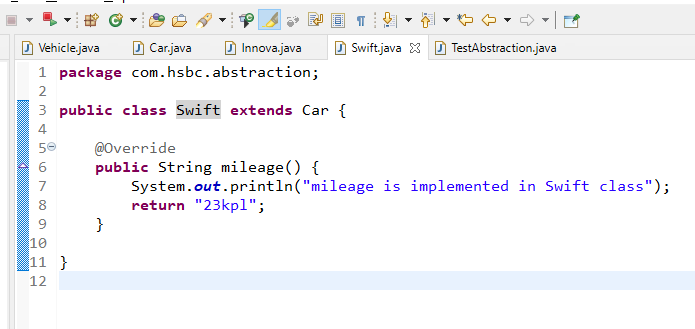
Car.java



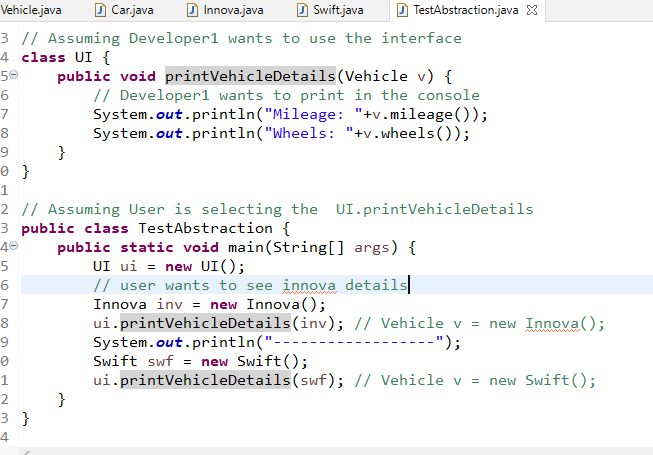
Innova.java



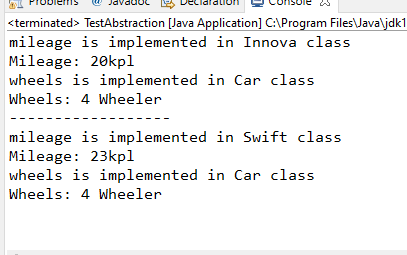
Swift.java



TestAbstraction.java



Output



Note: Inside interface all the members are by default public, even if you don’t use public, and also all the methods are abstract by default, all the variables are constants by default

Some of the important points on interface

1. The main purpose of using interface is to achieve the complete abstraction
2. interface can have only abstract methods
3. interface members are public by default
4. methods are public & abstract by default
5. variables are public, final & static
6. interface cannot have constructors
7. interface cannot be instantiated
8. class can implement more than one interface

if I1, I2, I3 are 3 interfaces then class C1 implements I1, I2, I3 {}

1. interface can extend more than one interface like multiple inheritance

If I1, I2, I3 are interfaces then interface I4 extends I1, I2, I3 { } is ok

Access Specifiers in Java

They are used to represent the visibility of classes, methods & variables, totally there are 4 access specifiers

1. private
2. no keyword for it (default access specifier)
3. protected
4. public

|  |  |  |  |
| --- | --- | --- | --- |
| Private | Default | Protected | Public |
| Visibility is only within the class | Visible only within the package and not visible outside the package | Visible within the package & outside the package only to the subclass | Visible to everyone |

package com;  
public class A { }   
class B { }

package org;  
import com.A; // valid  
import com.B; // compilation error

Passenger -> bookTickets();

cancelTickets();

Admin -> bookTickets();

cancelTickets();

modifyPassenger();

interface PassengerInterface { bookTickets(); cancelTickets() ; }

interface AdminInterface extends PassengerInterface { modifyPassenger(); }

class TickeService implements PassengerInterface, AdminInterface { } (OR)

class TickeService implements AdminInterface { }

abstract class Passenger { bookTickets(); cancelTickets(); }

abstract class Admin extends Passenger { modifyPassenger(); }

class TicketService extends Admin { 3 methods you will implement }

Developer -> UI -> Passenger p = new TicketService(); p.bookTickets(); p.cancelTickets();

Developer -> UI -> Admin a = new TicketService();

a.bookTickets();

a.cancelTickets();

a.modifyPassenger();

Developer -> UI -> Passenger -> PassengerInterface pi = new TicketService();

pi.bookTickets();

pi.cancelTickets();

pi.modifyPassenger(); // not possible at all

Developer -> UI -> Admin -> AdminInterface ai = new TicketService();

ai.bookTickets();

ai.cancelTickets();

ai.modifyPassenger(); //possible

When you develop any application you need to follow an architecture called as layered architecture i.e., MVC

Factory Pattern:

It takes care of creating the object & returning to the client code, by hiding which class is instantiated.

interface DAO {   
 void store();  
 void fetch();  
}

class DAOMySQL implements DAO { all methods must be implemented }

// without factory pattern Service Layer

1st approach: DAOMySQL dao = new DAOMySQL();

2nd approach: DAO dao = new DAOMySQL();

3rd appraoach: Using factory pattern

class DAOFactory {   
 public DAO getInstance() {   
 return new DAOMySQL();  
 }  
}

3rd approach:

DAOFactory factory = new DAOFactory();  
DAO dao = factory.getInstance(); // returns DAOMySQL

Service Layer can call methods of DAO interface like

dao.store();

dao.fetch();

Creating the application using layered architecture

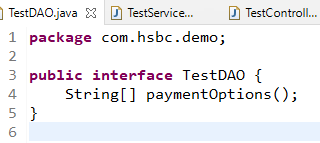
1. DAO layer: It will have methods that interact with the database, it should have methods as per the interface rules
2. Service layer: It will have methods that calls DAO layer methods, it should also have methods a per the interface rules
3. Controller layer: it will have methods that calls Service layer methods
4. View layer: it will have the User Interface
5. Factory Pattern: Takes care of creating the instances of DAO & Service and returning to the Service & Controller respectively. i.e., DAO instance returned to the Service & Service instance is returned to the Controller

Note: Certain features are missing here

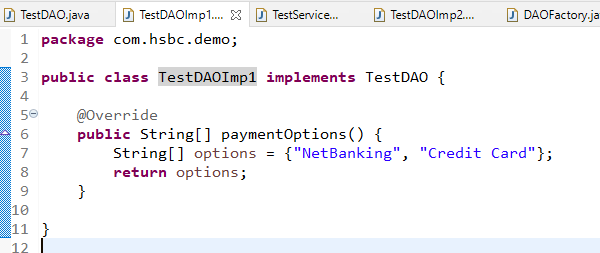
1. We are missing database here hence we use arrays as the temporary
2. We are also missing the view layer like Web, Mobile, Desktop and also controller will decide the view so we can’t have controller now, hence we use console itself as a view layer & controller layer

View layer role: Take input from the user & show the output to the user, now in console you can use Scanner to take input & System.out.println() for output.

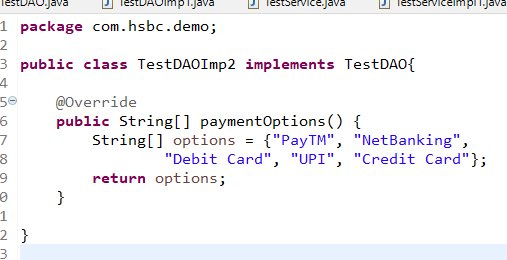
TestDAO.java



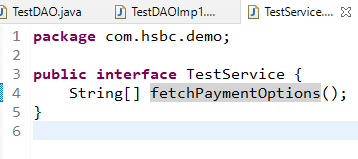
TestDAOImp1.java



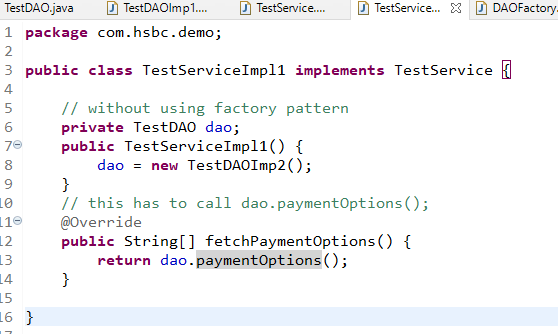
TestDAOImp2.java



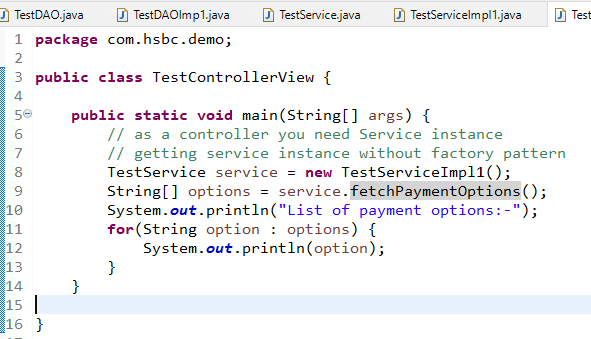
TestService.java



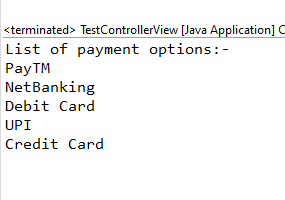
TestServiceImpl1.java



TestViewController.java

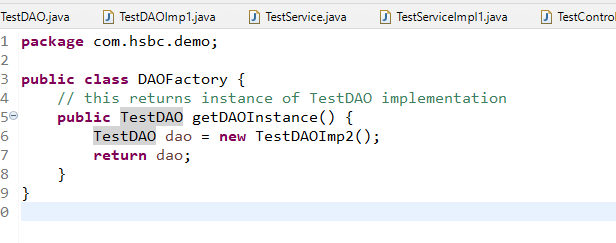


Output:



Here we are getting the output from the TestDAOImp2, but if we change the DAO we are changing Service as well, to remove this dependency we go with factory pattern which takes care creating object & returning the object

DAOFactory.java



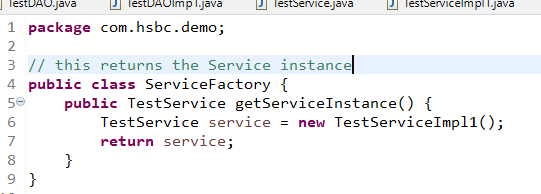
Here the factory is creating TestDAOImp1 instance and client which is Service can use this factory and get the DAO instance that way you are completely abstracting the object creation at the client side.

TestServiceImp1.java

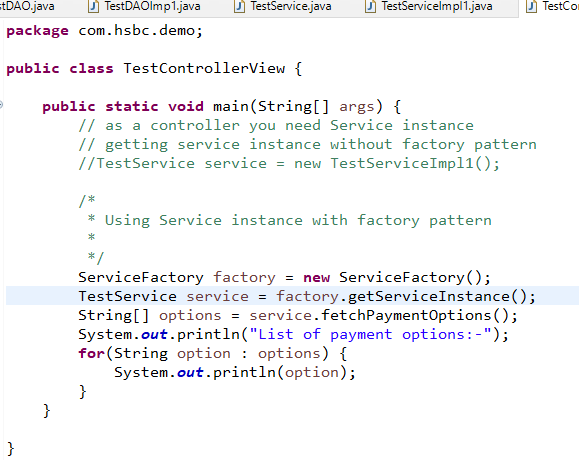


But you need to change in the Controller also as controller is creating instance using new operator

ServiceFactory.java



TestControllerView.java



The above one has 2 factory classes, but what you can do is you can create one ObjectFactory class that returns different types of instances like Service, DAO and also these factory methods can have some parameter and return the object of different implementation by checking the condition using the parameter

ex:

public TestDAO getDAOInstance() { … } // existing one  
public TestService getServiceInstance() {..} // existing one

// you need to implement

public TestDAO getDAOInstance(int option) {   
 // if option is 1 return TestDAOImp1  
 // if option is 2 return TestDAOImp2  
}

public TestService getServiceInstance(TestDAO dao) {  
 // return new TestServiceImp1(dao)  
}

// client code shouldn’t be modified, without modifying service & dao it should work on TestDAO1 & TestDAO2 by passing option from the main method

i.e.,

if you pass 1, then it gives TestDAOImp1

TestDAO dao = factory.getDAOInstance(“1”);

TestService service = factory.getServiceInstance(dao);

if you pass 2, then it gives TestDAOImp2

TestDAO dao = factory.getDAOInstance(“2”);

TestService service = factory.getServiceInstance(dao);