Modules:

* Java 8
* Hibernate
* Spring Core
* RESTful Webservices

Java: It is platform Independent & Object Oriented language

What is platform Independent?

* It can be run on any platforms (OS) without altering.
* It is write once and run anywhere.

What is object oriented programming language?

* It represents real world entities in the application, these real world entities are also called as Objects.
* An object will have properties and behaviours
* Properties: What object has.
* Behaviours: What object does.

Example of Objects: In a banking application - Customer, Account, Employee are all the objects it can have.

Customer: Properties are customerId, name, age, gender

Customer: Behaviours are withdraw(), deposit(), getDetails()

Employee: Properties are employeeId, name, salary

Employee: Behaviours are createCustomer(), deleteCustomer() and so on.

and etc.

Two basic building blocks of OOPs.

1. Classes: It is a blue print of an object or template of an object.
2. Objects: It is created from the class which is a real world entity.

How to create class?

Syntax:

class className {

object’s properties

object’s behaviours

}

Example:

class Employee {

int employeeId;

String employeeName;

.....

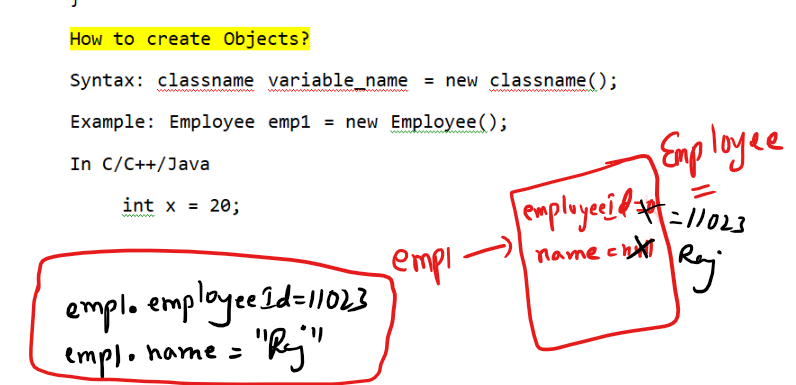
void takeComplaints() {

......

}

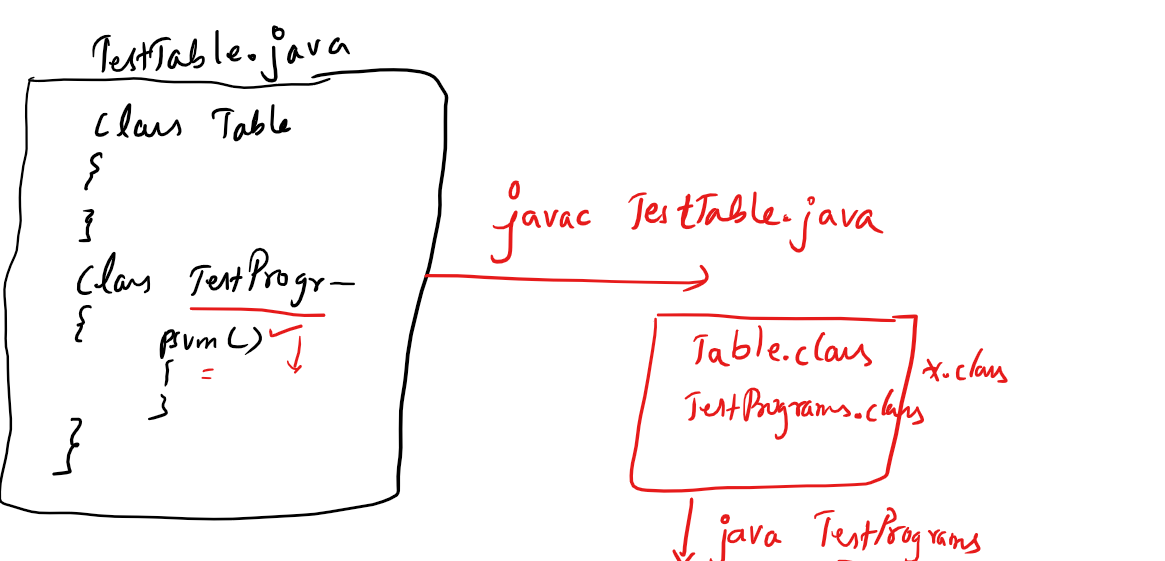
....

}



Setting Path:





Constructors: It is a kind of method whose name is same as class name but it will not have return type.

Note: if a class doesn’t have any constructor then compiler adds a default constructor.

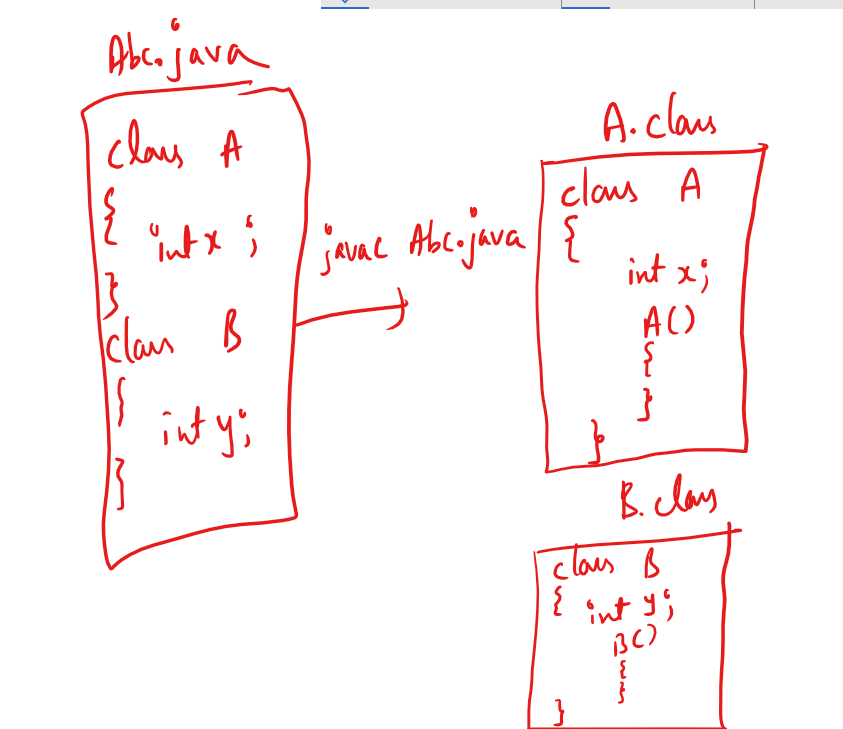
Note: If you add a constructor of your own then compiler doesn’t add a default constructor.

How the default constructor looks:

It looks same as class name followed by () i.e.,

Inside a class Table you will get a default constructor as Table(){

}



Note: A class can have any number of constructors i.e, you can overload the constructor.

Overloading: Same name but different set/type of parameters

i.e.,

Table(): Default constructor

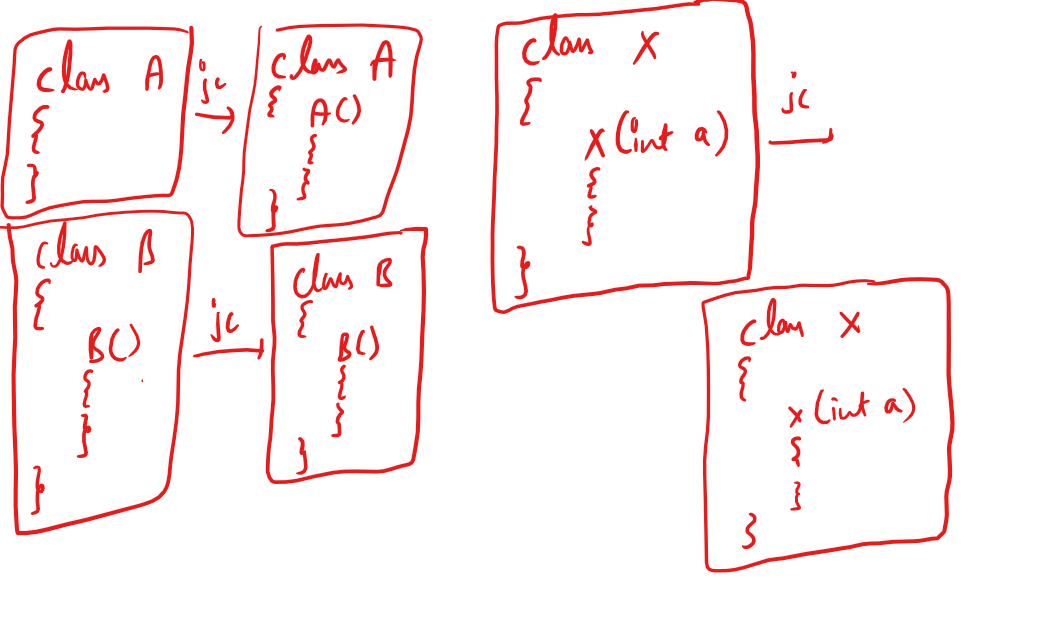
Table(int x, int y): 2 parameterized constructor

Table(double x, double y): 2 parameterized constructor

Table(int a, double b): 2 parameterized constructor

Table(double a, int b): 2 parameterized constructor

Table(int x): 1 parameterized constructor.



Constructor is called when you create object, so you can use it to initialize the variables while you create object.

You can find the informations present inside the class using javap command, stands for javaParse, which is used to disassemble the class,

Usage of javap command is same like java command i.e.,

* javap Table
* javap A
* javap TestPrograms

TestTable.java

// notepad TestTable.java

class Table {

int rows;

int cols;

Table() {

System.out.println("Table() created");

rows = 1;

cols = 1;

}

Table(int x, int y) {

System.out.println("Table(int, int) created");

rows = x;

cols = y;

}

void calculateCells() {

int r = rows \* cols;

System.out.println("Result = " + r);

}

}

class TestPrograms {

public static void main(String args[]) {

Table t1 = new Table(30, 40); // Table(int, int){}

Table t2 = new Table(10, 40);

Table t3 = new Table();

t3.rows = 100;

t3.cols = 200;

t1.calculateCells();

t2.calculateCells();

t3.calculateCells();

}

}

Output:



TestEmployee.java

class Employee {

int empId;

String empName;

double salary;

String gender;

Employee(int id, String name, double sal, String g) {

empId = id;

empName = name;

salary = sal;

gender = g;

}

void raiseSalary() {

salary = salary + (salary \* 0.05);

}

void updateSalary(double sal) {

salary = sal;

}

void display() {

System.out.println(empId+" "+empName+ " "+salary+ " "+gender);

}

}

class TestEmployee {

public static void main(String[] args)

{

Employee e1 = new Employee(101,"Rahul",20000,"Male");

Employee e2 = new Employee(102,"Jennifer",30000,"Female");

Employee e3 = new Employee(103,"Alex",10000,"Male");

e1.display(); e2.display(); e3.display();

System.out.println("------Raise Salary Request--------------");

e1.raiseSalary(); e2.raiseSalary(); e3.raiseSalary();

System.out.println("------Display After Raise--------------");

e1.display(); e2.display(); e3.display();

System.out.println("--------Update Salary Request------------");

e1.updateSalary(50000); e2.updateSalary(50000);

e3.updateSalary(50000);

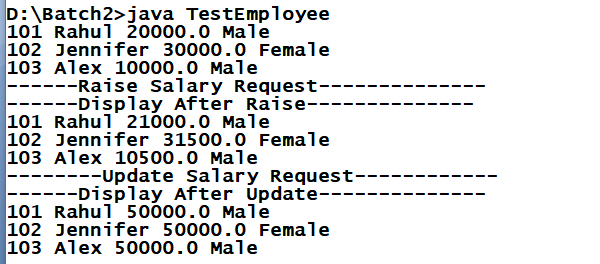
System.out.println("------Display After Update--------------");

e1.display(); e2.display(); e3.display();

}

}

Output:



The keyword this: This is a current object reference, it is used to avoid conflicts between local variable names and global variable names when their name’s are same

this keyword can be used inside Constructor and Non-Static methods.

Note: Using this keyword inside a static method is a compilation error, because static members are accessed without creating objects.

Like this keyword one more keyword called super also not allowed inside static methods.

TestEmployee.java

class Employee {

int empId;

String empName;

double salary;

String gender;

Employee(int empId, String empName, double salary, String gender) {

this.empId = empId;

this.empName = empName;

this.salary = salary;

this.gender = gender;

}

void raiseSalary() {

salary = salary + (salary \* 0.05);

}

void updateSalary(double salary) {

this.salary = salary;

}

void display() {

System.out.println(empId+" "+empName+ " "+salary+ " "+gender);

System.out.println(this.empId+" "+this.empName+ " "

+this.salary+ " "+this.gender);

}

}

class TestEmployee {

public static void main(String[] args)

{

Employee e1 = new Employee(101,"Rahul",20000,"Male");

Employee e2 = new Employee(102,"Jennifer",30000,"Female");

Employee e3 = new Employee(103,"Alex",10000,"Male");

e1.display(); e2.display(); e3.display();

System.out.println("------Raise Salary Request--------------");

e1.raiseSalary(); e2.raiseSalary(); e3.raiseSalary();

System.out.println("------Display After Raise--------------");

e1.display(); e2.display(); e3.display();

System.out.println("--------Update Salary Request------------");

e1.updateSalary(50000); e2.updateSalary(50000);

e3.updateSalary(50000);

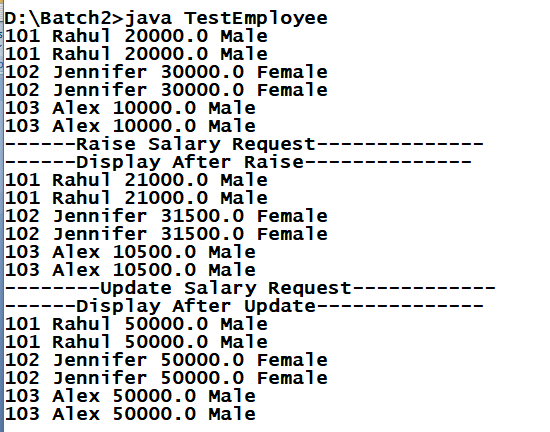
System.out.println("------Display After Update--------------");

e1.display(); e2.display(); e3.display();

}

}

Output:



Static keyword: static is used on variables & methods to access without creating objects.

When you want to access some members inside the class without instantiating it then you can have static variables & methods.

You can access static members using class names.

example:

Calculator.java

class Calculator {

static int add(int x, int y)

{

return (x + y);

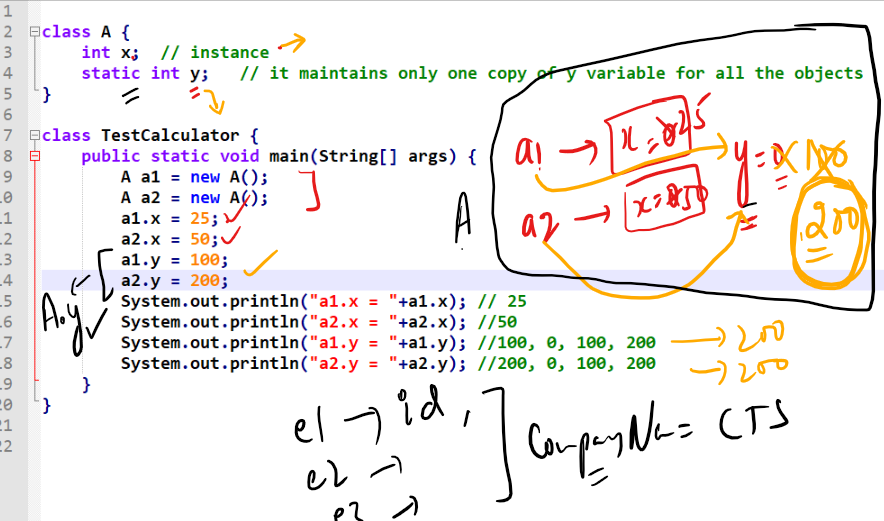
}

}

// to call add method you don’t need to create object of Calculator.

int result = Calculator.add(30, 40);

result value will be 70.

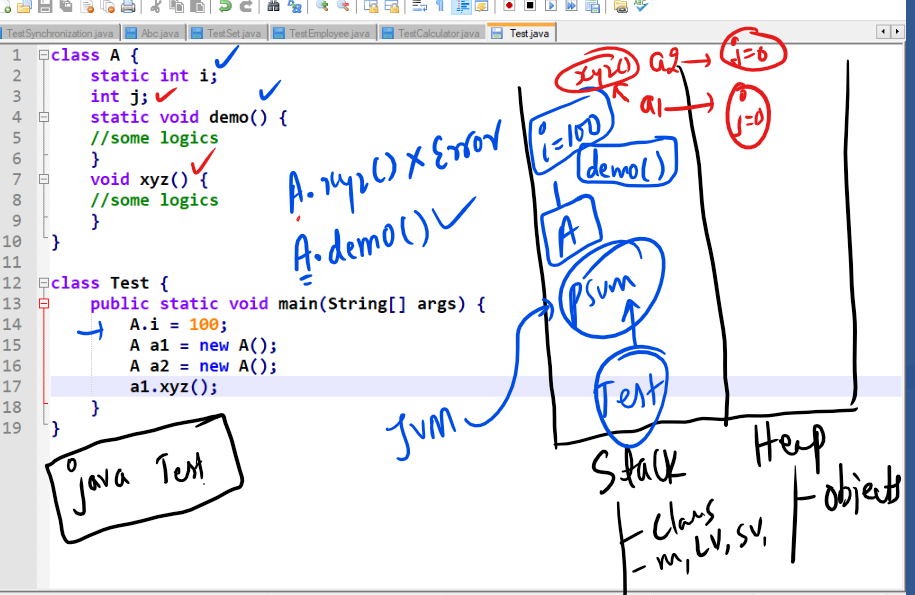


static members can be accessed either through class names or reference variables.

i.e.,

A.y or a1.y

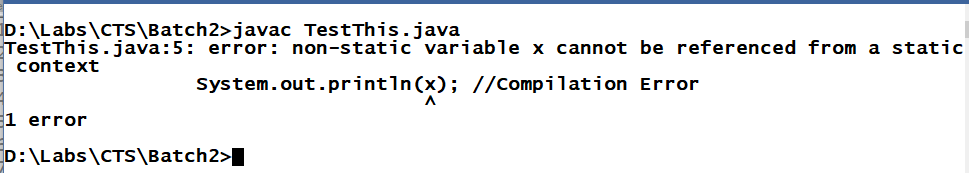
Memory management





Note: static methods must not access non-static variables directly, but you can use object reference and access the non-static variable

Output:



Note: Static methods cannot access non-static members directly, whereas non-static methods can access static or non-static members directly.

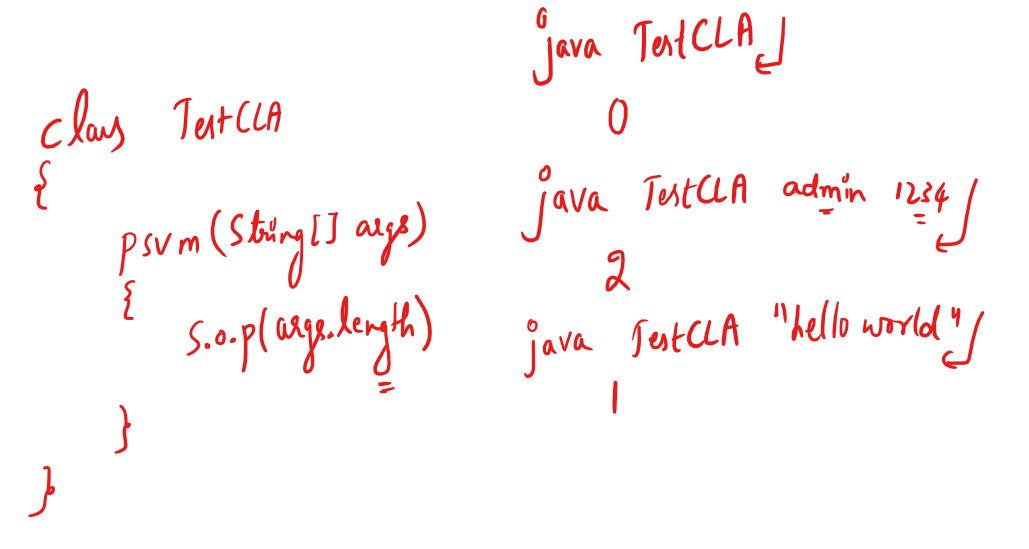
Example:

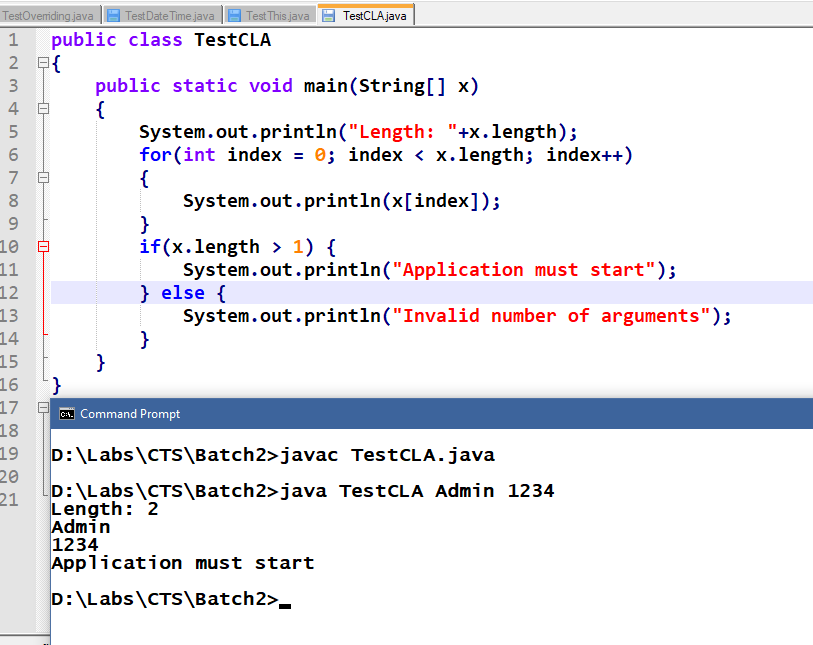


Command Line Argument:

It is an argument you can pass to the main method while running your code, String[] args in your main method is a command line argument.

public static void main(String[] args) {....}

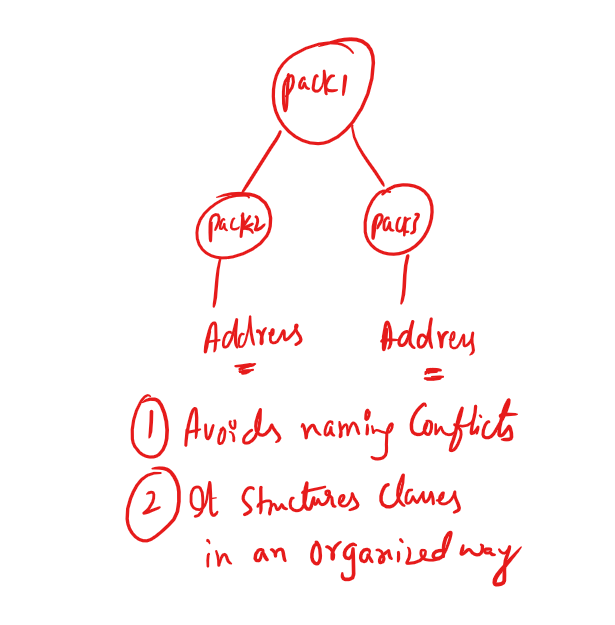
Note: [] bracket you can have after the data type or after variable name in an array.  




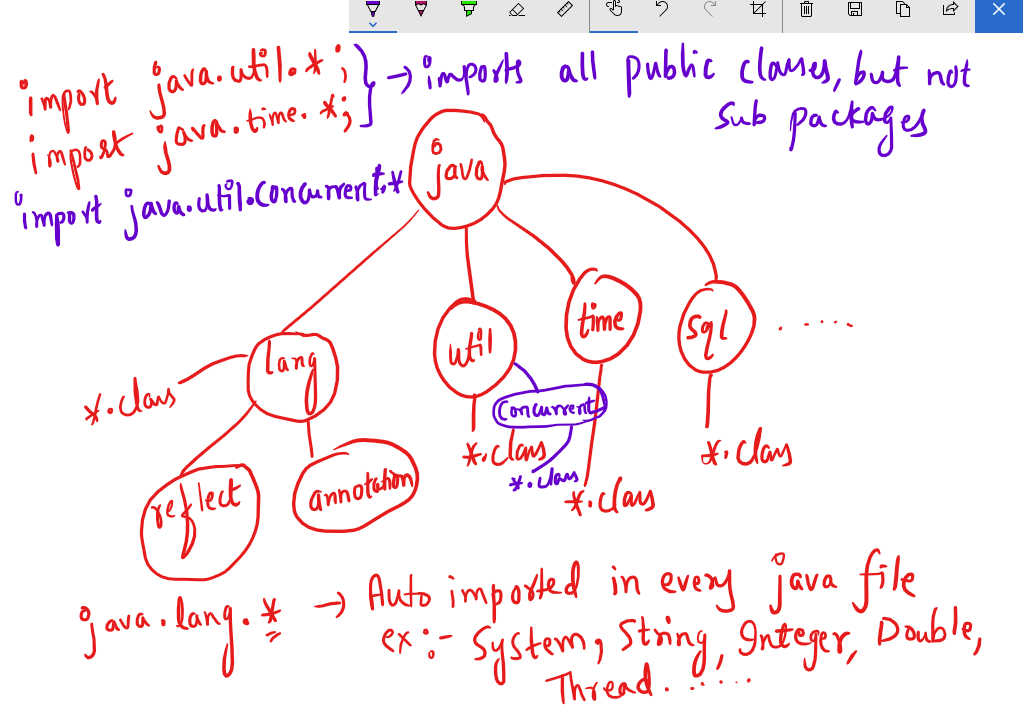
Packages in Java

What are packages?

They are like folders in Java, which will have collection of classes and sub packages.



In Java there are lot of predefined packages like



Note: import java.util.Scanner; will import only Scanner class whereas import java.util.\*; will import all the public classes.

Some of the predefined classes in java.lang are

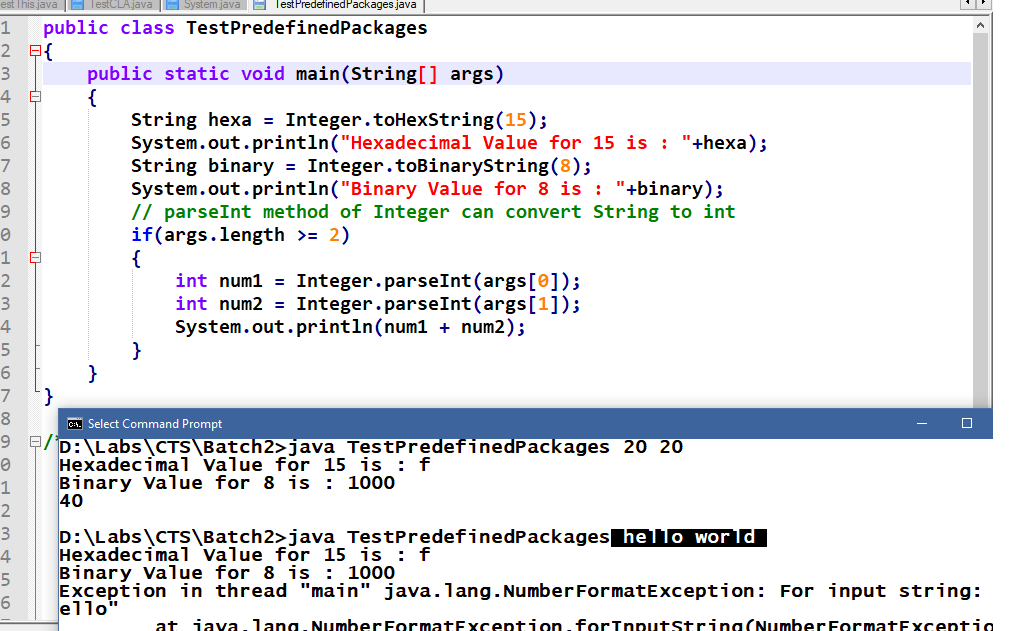
Integer, String, Float, Double, Exception and many more

javap java.lang.Integer will list

public static String toHexString(int)

public static String toBinaryString(int)

.....



Enhanced for loop or for each loop

It is same like traditional for loop but it simplifies the iteration, it can be used only if you are iterating in forward direction and all the elements.

Syntax:

for (type var : collections) {...}

Here collections are variables that holds array or collection of elements, type var must match to the collection type

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

for(int temp : elements) {

// now temp picks each item in the array till it iterates over all the elements.  
}

String[] args, a command line argument can also be iterated using for each loop i.e.,

for(String temp : args) {/\* some code \*/}

You can also declare an array using new operator as below:

int[] elements = new int[5];

Now elements will occupy 5 memory to store 5 integer values, by default each index elements will be having 0 (zero) you must initialize each index as below:

elements[0] = 20; elements[1] = 40 and so on.

You can also declare an array of Complex data types as below:

Employee[] employees = new Employee[4];

here you can have 4 employee objects in employees array, but initially all the index will have null value.

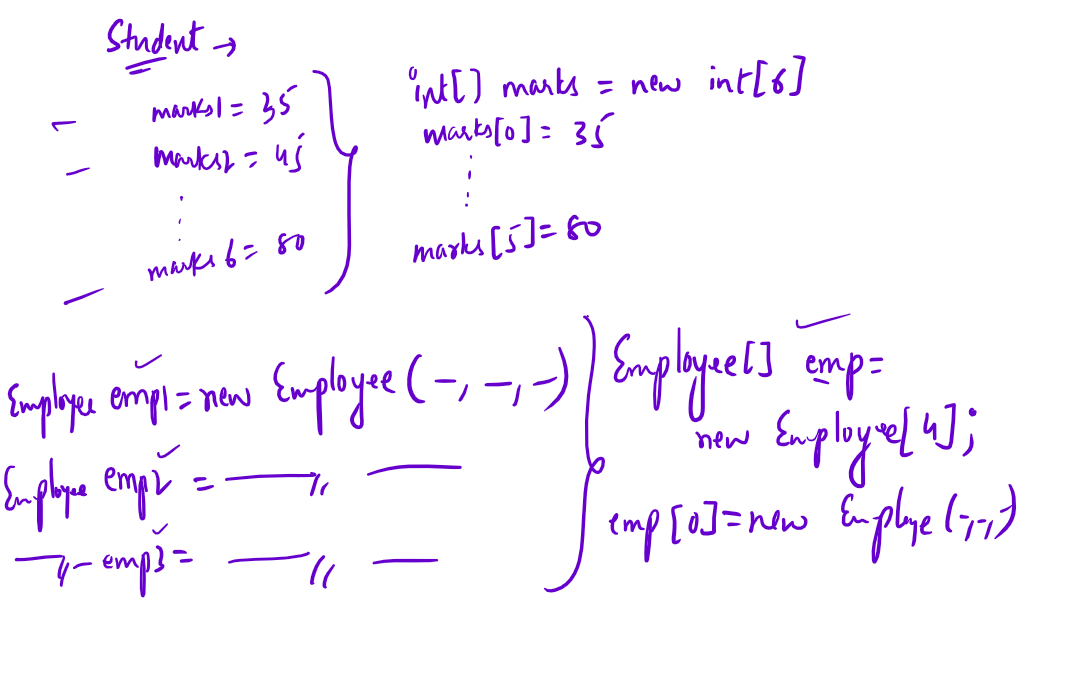
you can initialize each index by creating object.

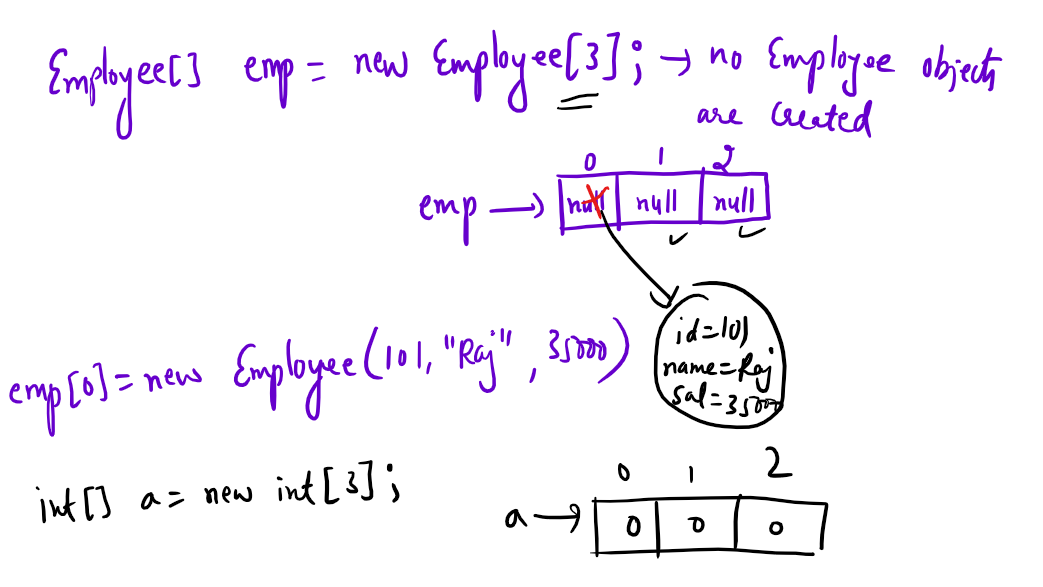
employees[0] = new Employee(101, “Rahul”, 35000);

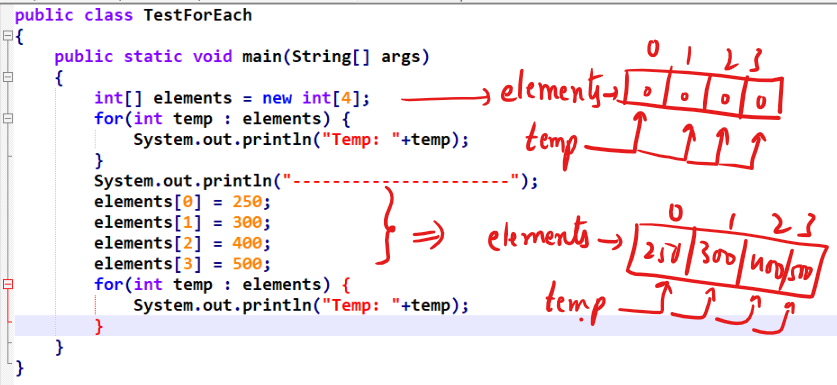
employees[1] = new Employees(102, “Vikram”, 40000);

and so on.

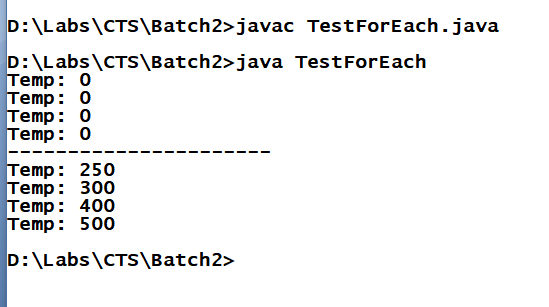
It is required if you want to maintain multiple Employee objects. Because creating 4 reference variables of same type is of no use instead you can have an array of a particular type and mention the size.







Output:



You can apply for-each loop on command line arguments as well as shown below:



OOPs concepts:

Encapsulation

Inheritance

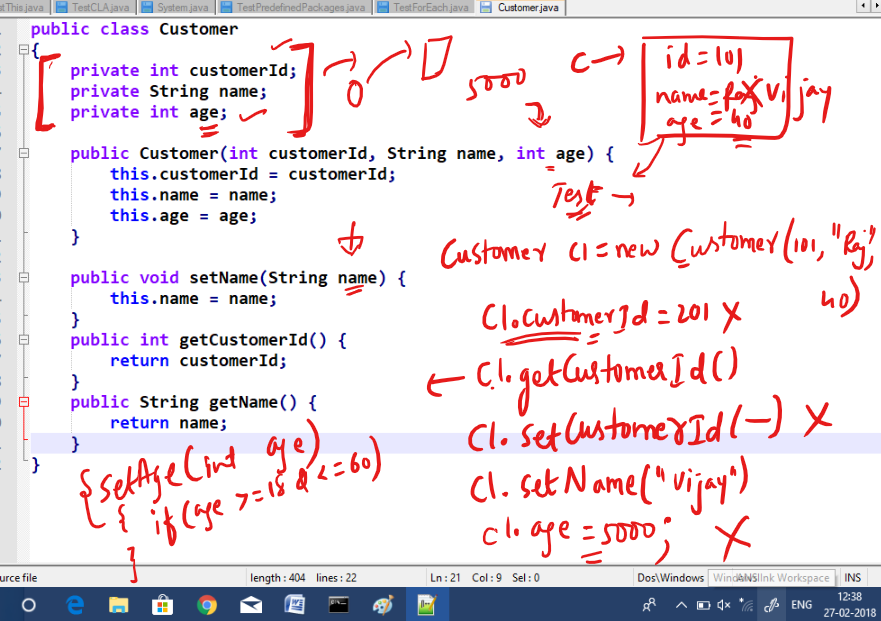
Polymorphism

Abstraction

What is Encapsulation?

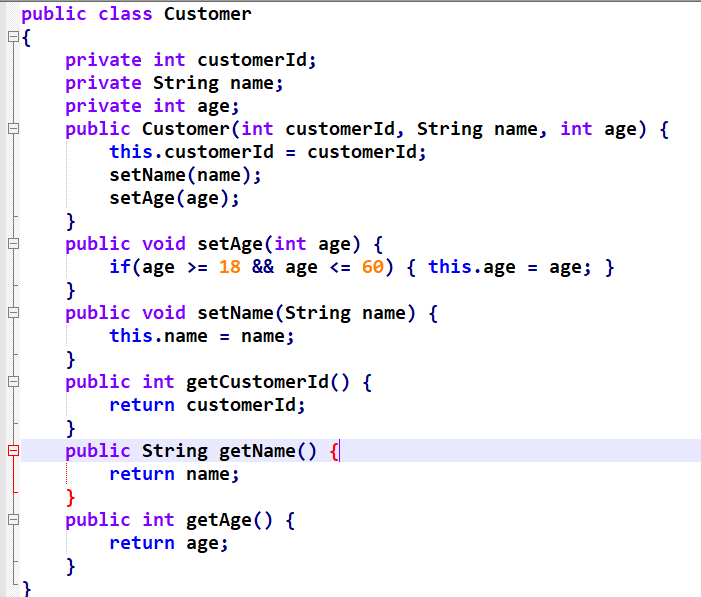
It is used to hide the variables using private so that you can access only through methods which are public.

An encapsulation will allow you to bind variables and methods in a single unit called class and a fully encapsulated class will have private variables & public methods which are also called as Java Beans.

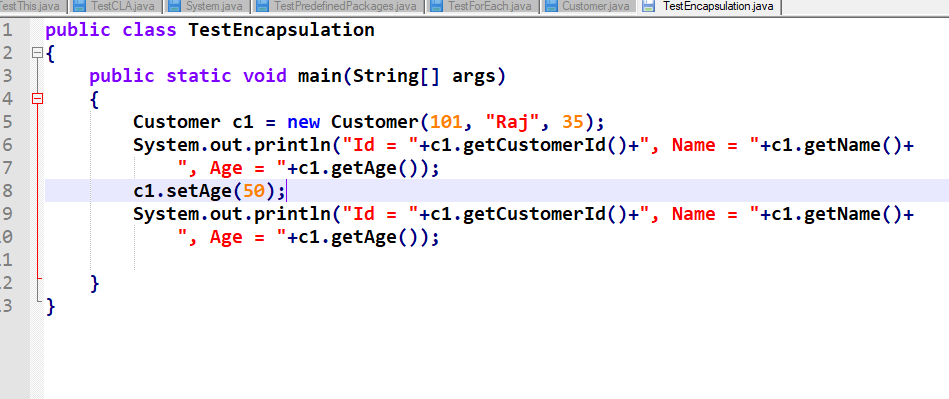


By making private you can have more control over the variables value.

Customer.java



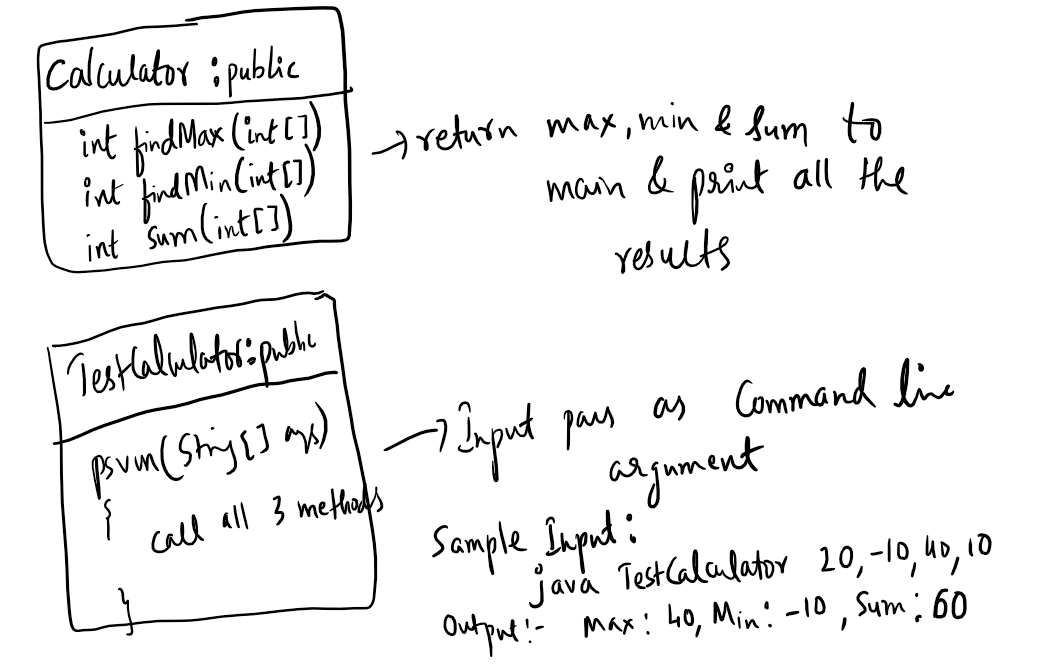
TestEncapsulation.java



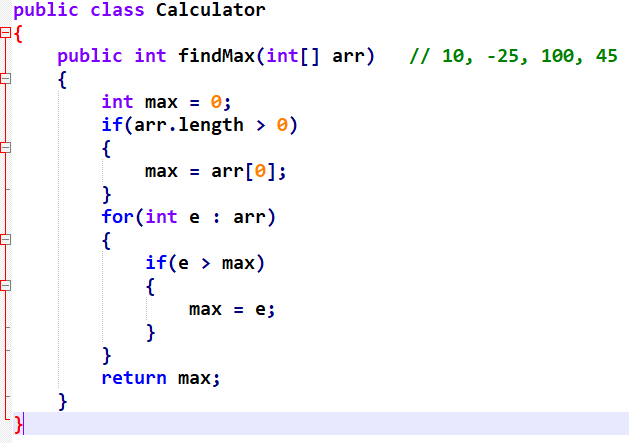
Output:



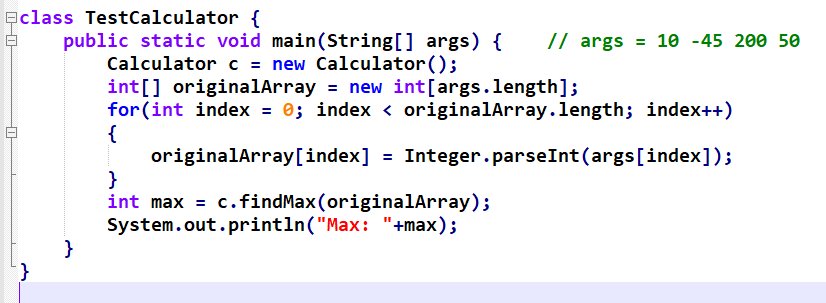
Assignment:



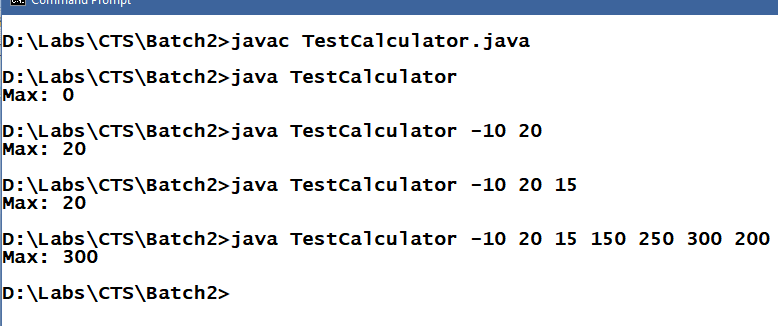
Solution for findMax(int[])



Calling findMax

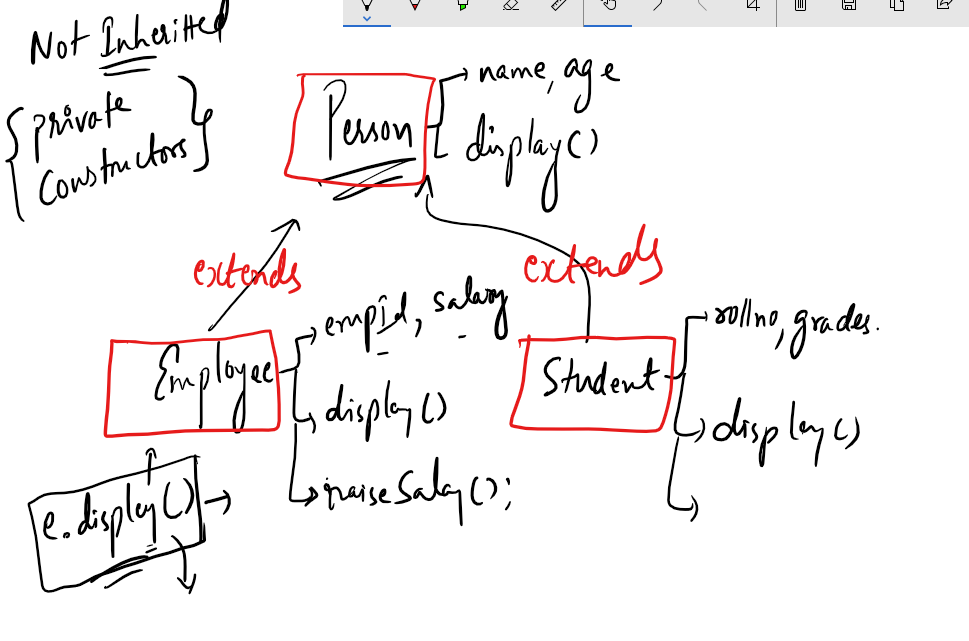


Output:



Inheritance

Process acquiring the properties and behaviours of an object from another object.



In subclass display() is overridden so that when you create an object of Employee or Student it calls overridden method, if you don’t override then it calls inherited method.

TestInheritance.java

class Person {

String name = "Rahul";

public void setName(String name) {

this.name = name;

}

public void display() {

System.out.println("Name = "+name);

}

}

class Employee extends Person {

int empId = 102;

public void display() {

System.out.println("EmpId = "+empId+", Name = "+name);

}

}

class Student extends Person {

String grades = "A";

public void display() {

System.out.println("Name = "+name+", Grade = "+grades);

}

}

class TestInheritance {

public static void main(String[] args) {

Person p1 = new Person(); p1.display();

Employee e1 = new Employee();

e1.display(); e1.setName("Vijay"); e1.display();

p1.display();

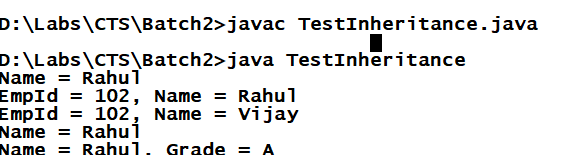
Student s1 = new Student();

s1.display();

}

}

Output:

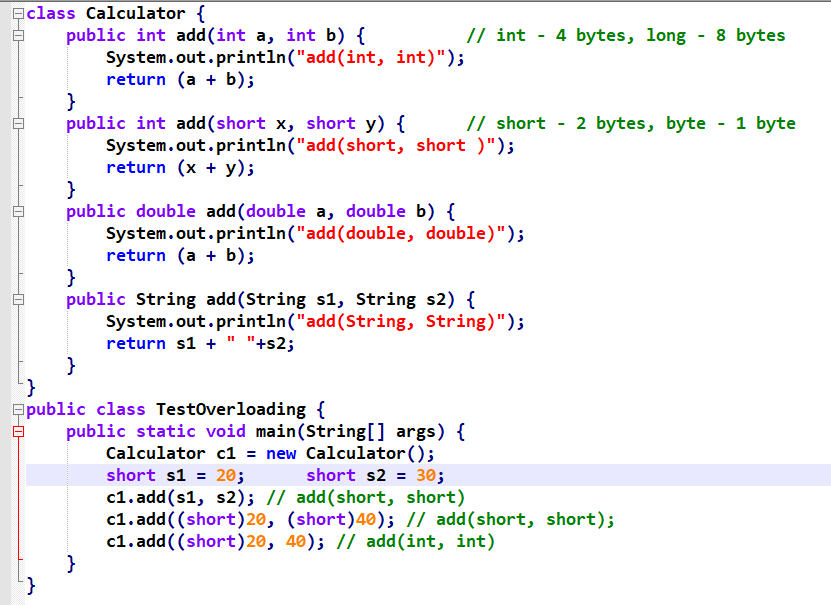


Polymorphism: Same method name with many forms, i.e., a method that behaves differently upon different situation.

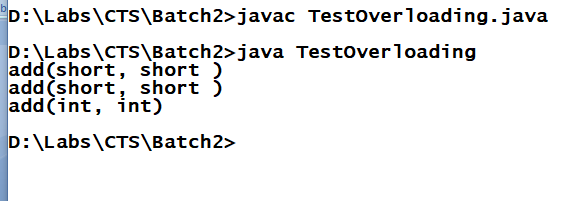
There are 2 types of Polymorphism

* Compile time polymorphism >> Overloading
* Runtime polymorphism >> Overridding

Overloading Example

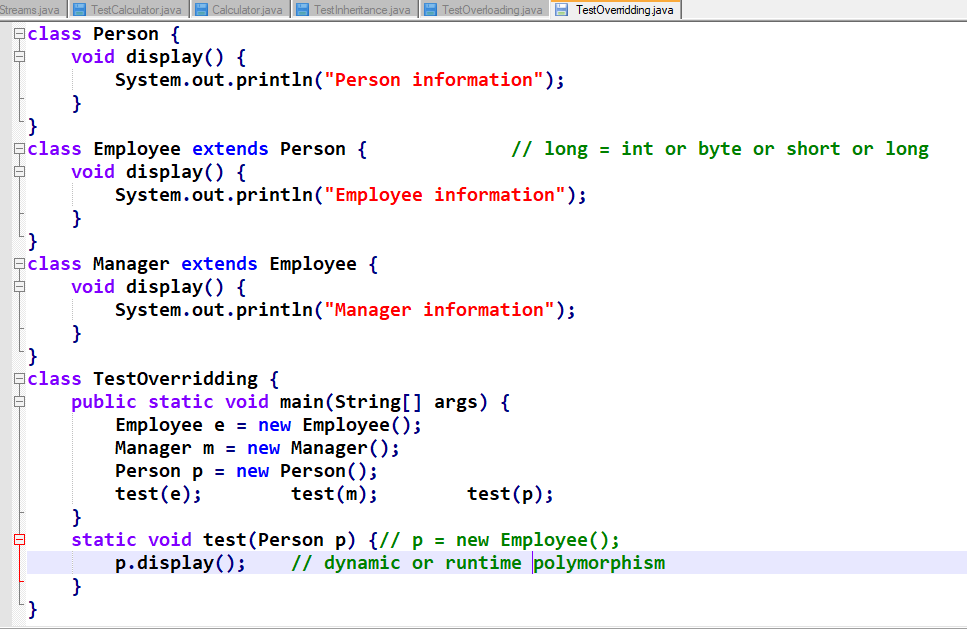


Output:

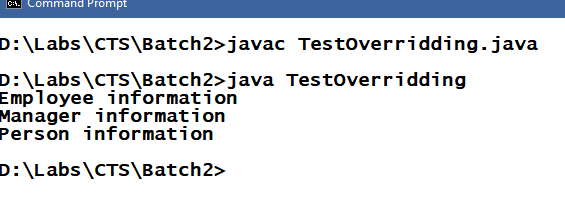


Runtime Polymorphism:

Method invocation is decided at runtime; here overridden methods are decided based on the type of object you are using to invoke the overridden method.

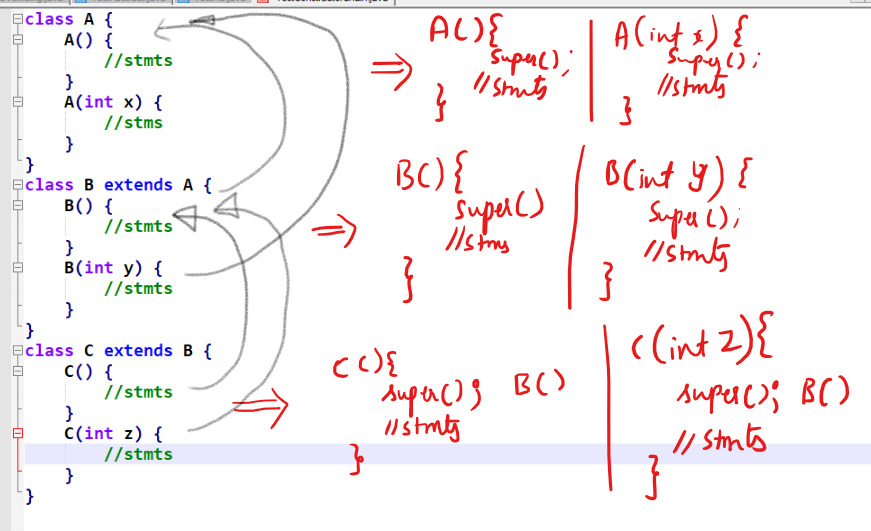


Output:

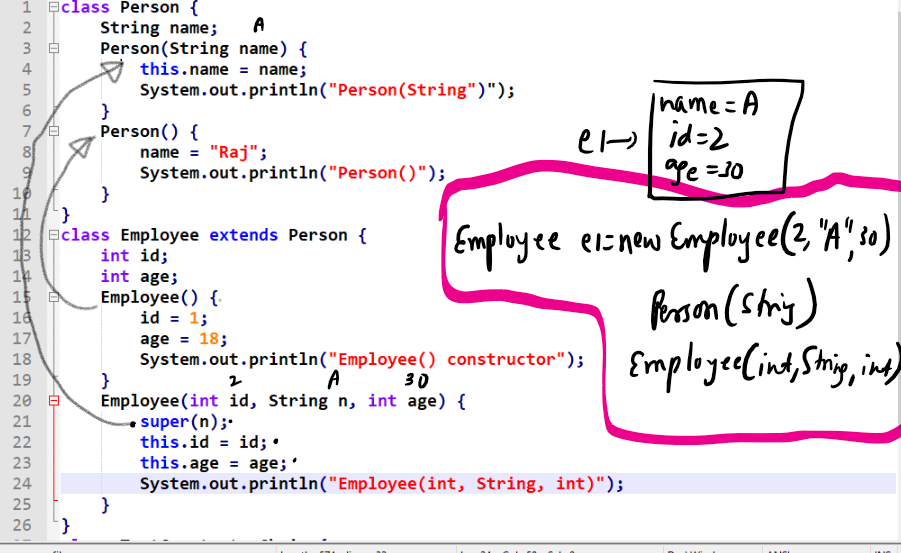


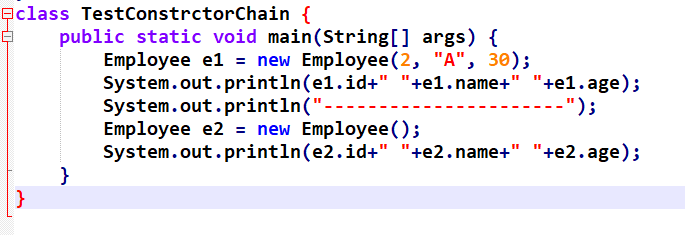
Assignment:

* Understanding about Abstraction - Abstract classes & Interfaces.
* super keyword in the constructor chaining or constructor invocation.

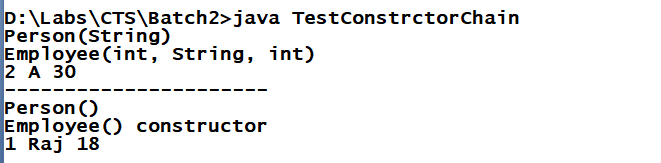


Every subclass constrctors implicitly calls super class default constructor from its first line, you can also explicitly call super class constructors using super() call, but you do that when you want call parameterized constructor of your parent class.





Output:



final keyword:

It is used on variables, methods and classes to restrict modifications.

final variables are constants and can’t be modified

final methods can’t be overridden.

final classes can’t be inherited.

Usage:

final double PI = 3.14; //every object will have their own

copy of PI

final static double PI = 3.14;//all the object will have

//single copy of PI

final void test1() {...}

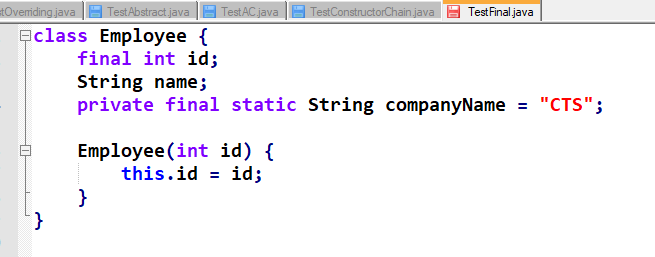
final class Manager extends Employee {....}

Note: final variables must be initialized; the initialized value itself is final because it can’t be modified.

Note: final non-static variables can be initialized inside constructor or at the time declaration.

Note: final static variables must be initialized at the time of declaration itself.

Note: Not initializing final variables will lead to compilation error.



Abstraction:

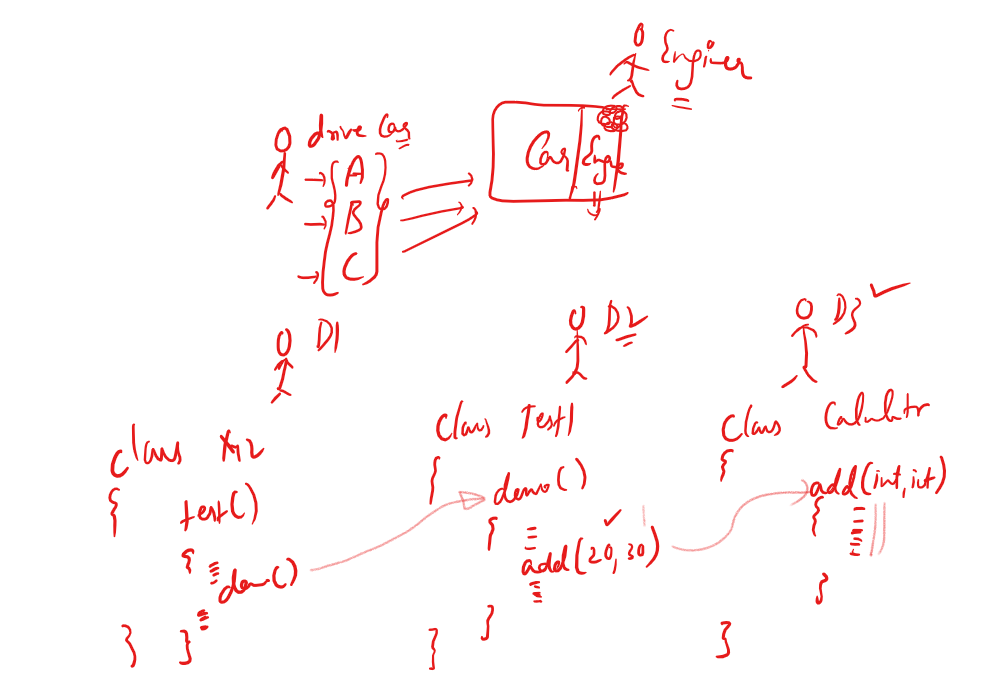
It is used to hide the complexity of an object and expose only the necessary interfaces.

It is mainly for developers who must know what a method does instead of know how it does (how it is implemented).

Example: System.out.println(); // now println() does printing but we know what it does but don’t know how it is implemented.

Example: Integer.parseInt(..); // parseInt() does what we know, but we don’t know its implantation

So here these methods are made us available through abstraction so that we can access them without knowing how it is implemented.



Because of abstraction the code you modify at one place will not affect other codes because other codes must know what the method does instead of how it is implemented.

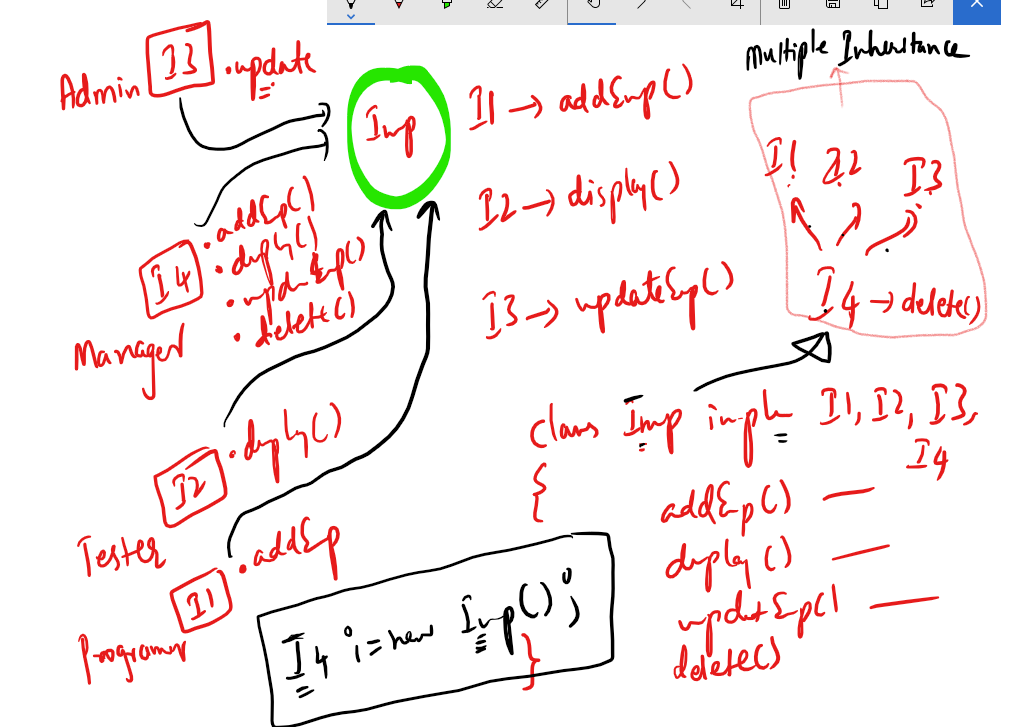
Abstraction can be achieved in 2 ways

1. interfaces
2. abstract classes

Interfaces: It is a kind of class which will have only abstraction methods and constants; everything inside interface is public by default.

Things you can achieve through interfaces are:

1. Complete abstraction, through this modification is going to be less.
2. Multiple inheritances.
3. It is secured because it doesn’t allow you to access the methods that are not present inside interface.



Some points in interface:

* Interface will have only abstract methods and constants.
* Interface will not have constructors.
* All the members of interfaces are public by default (you cannot change the visibility).
* Through interface you can achieve multiple inheritances.
* You cannot create object of interface but you can reference to the object that has implemented the interface.

TestInterfaces.java

interface X {

int add(int a, int b); // add is by default public and abstract

}

interface Y {

int sub(int a, int b); // sub is by default public and abstract

}

interface Z extends X, Y {

int mul(int a, int b); // mul is by default public and abstract

}

class Operations implements X, Y, Z {

public int add(int a, int b) {

System.out.println("add(int, int)");

return (a + b);

}

public int sub(int a, int b) {

System.out.println("sub(int, int)");

return (a - b);

}

public int mul(int a, int b) {

System.out.println("mul(int, int)");

return (a \* b);

}

}

class TestInterfaces {

public static void main(String[] args) {

Operations op = new Operations();

X x = op;

x.add(20, 30);

Z z = op;

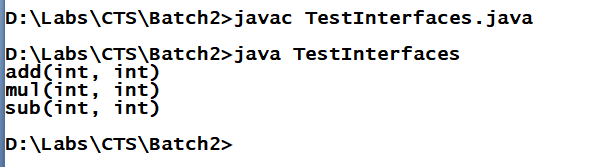
z.mul(2, 3);

z.sub(4, 3);

}

}

Output:



Abstract class: It is a class with abstract keyword, it allows you to have methods with body and without body i.e., method definitions and method declarations.

An abstract method must always be written either inside interfaces or abstract classes.

Usage:

abstract class Car {

abstract void showMileage();

void showWheels() {

// 4 wheels  
}

}

class Innova extends Car {

void showMileage() { 16kmpl }

}

class Swift extends Car {

void showMilage() { 20kmpl }

}

class TestAbstract {

public static void main(String[] args) {

Car c;

Innova i = new Innova();

Swift s = new Swift();

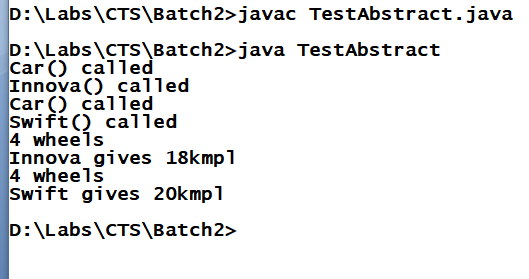
c = i; c.showWheels(); c.showMileage();

c = s; c.showWheels(); c.showMileage();

}

}

Output:





Similarity between interface and abstract class are: We can’t create object on both.

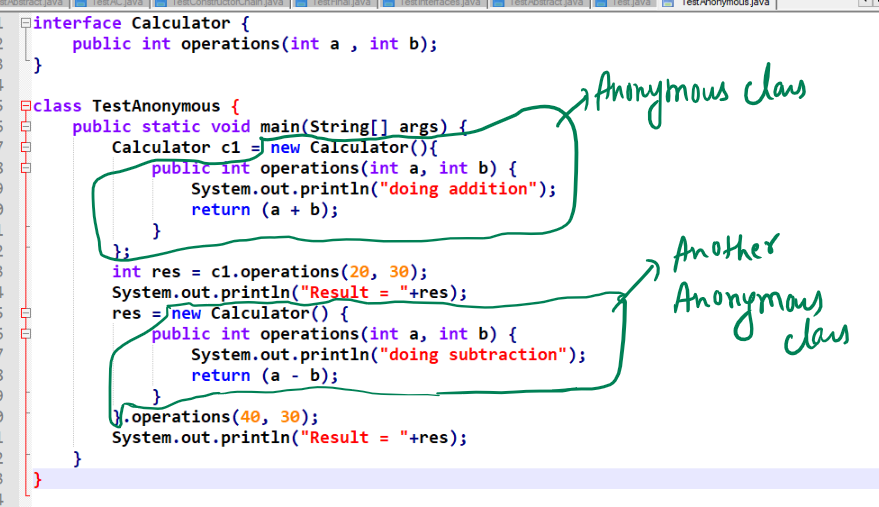
Note: An abstract class may or may not have abstract methods but abstract methods should be inside abstract class, abstract keyword does 2 tasks i.e., declaring method and restricting object creation.

Anonymous inner class

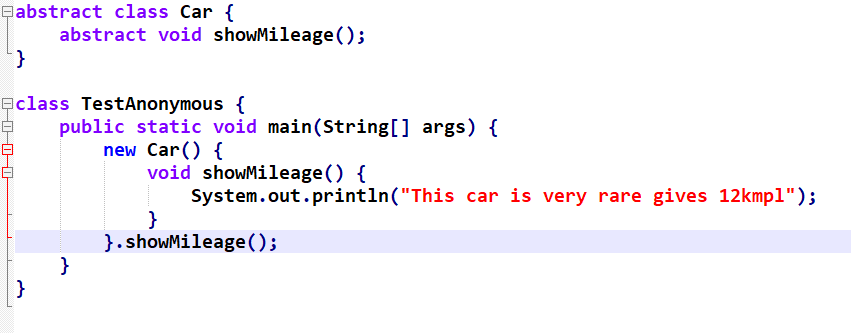
An inner class which doesn’t have any name, you can use anonymous classes when you don’t want to create multiple classes which are of no use outside the class.

An anonymous class is a sub class of a class or interface but it will not have any name.

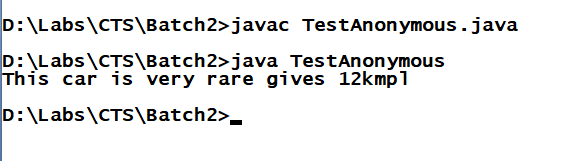
Anonymous class is a kind of class which is declared & initialized at the same time.



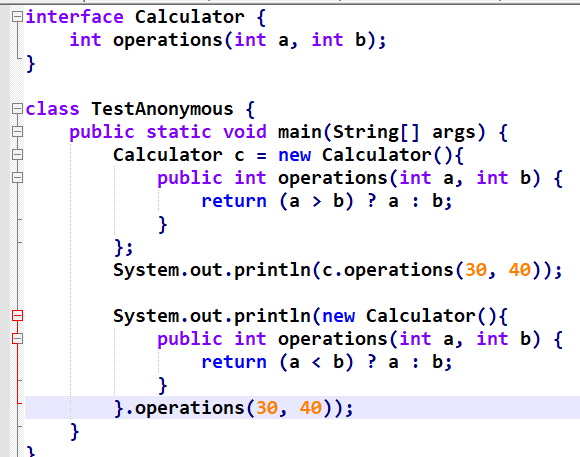
Use anonymous classes when you want the implementations to be used only in one method not reusable in other class methods.



Output:



You can also have anonymous class method invocation inside println() as below:



Lambda Expressions:

These are replacement for anonymous classes which simplifies the anonymous class implementation, but lambda expression must be used only if the interface has only one method.

Note: Just like you can pass anonymous class implementations as a method parameter you can pass Lambda expressions in the method parameter.

Example on passing anonymous class as a parameter:

interface Calculator { int operations(int a, int b); }

Assume there is a method that takes Calculator as the parameter

void test(Calculator c)

{ c.operations(20, 30); }

Now the Anonymous class implementation can be done as below:

1st way:

Calculator c = new Calculator(){

public int operations(int x, int y) {

return (x + y); }

};

test(c);

2nd way:

test(new Calculator(){

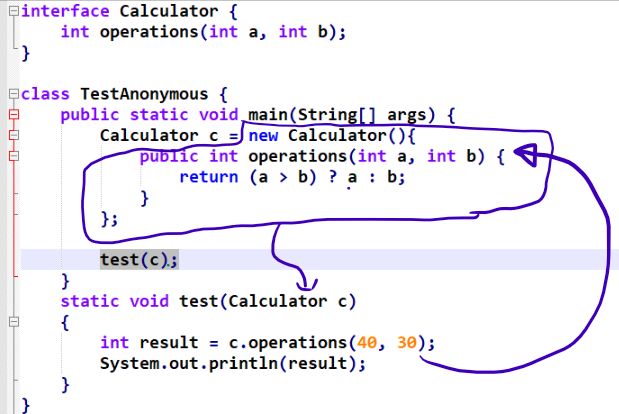
public int operations(int x, int y) {

return (x - y);

}

});

Example on anonymous class without lambda expression looks as below:



Output:

40

How to write lambda expressions:

When you write lambda expressions you will not use method names instead you use only these () braces.

i.e., instead of writing as below

public int operations(int x, int y) { return (x + y); }

You will write as below

(x, y) -> (x + y);

Since lambda is only for those interfaces having only one method the expression would be an implementation to that single method, hence you don’t need method name.

If you want more than statement to be written you can use {} brace as below:

(x, y) -> { /\* statements \*/ return (x + y); }

Complete Usage:

Without lambda expression:

Calculator c = new Calculator() {

public int operations(int x, int y) {

/\* some statements \*/

return (x + y);

}

}

c.operations(20, 30) >> returns 50

With lambda expression:

Calculator c = (x, y) -> {

/\* some statements \*/

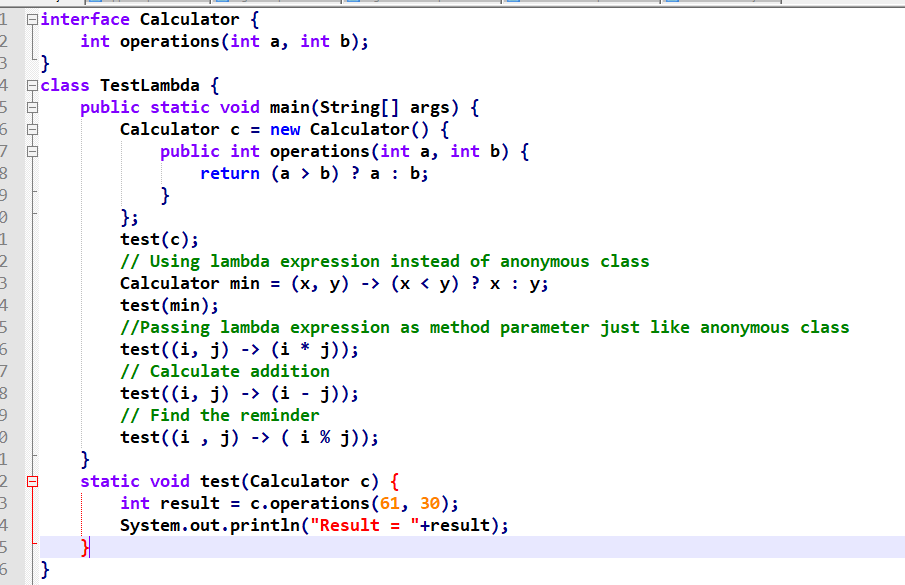
return (x + y);

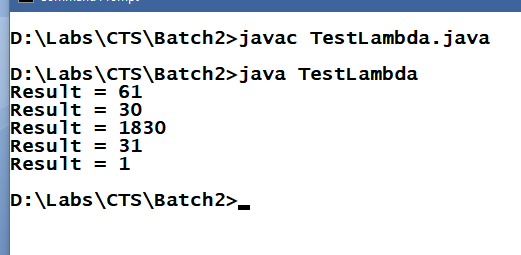
}

Above expression is for set of statements, if you have only one statement then you can write expression as below:

Calculator c = (x, y) -> (x + y);

c.operations(20, 30) >> returns 50

Example  
  
Output:



Some points on Lambda Expressions

* It must be used only when you have only one method in an interface, an interface which has only one method is called as functional interface
* Lambda expression allows you to pass methods as a parameter which is why lambda expression are called as functional parameter which means a parameter having a function
* You cannot use lambda expression on classes though you have only one method because every class is subclass of Object which itself provides many methods to all the classes.
* Some of the predefined interfaces having only one method provided by Java are:
  + Runnable -> public void run();
  + Comparator -> public int compare(Object, Object);
  + Comparable -> public int compareTo(Object)
  + ActionListener -> public void actionPerformed(ActionEvent)

...

Predefined classes in Java

All these classes are present in java.lang.\* package

Object >> Top most class for all the classes

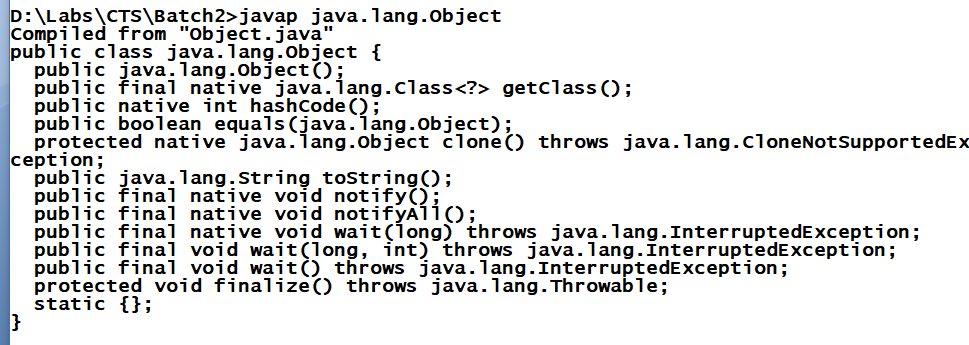
String >> you can use to work Strings like concatenation, getting a character at specific index, changing everything to uppercase or lower case and etc.

Integer >> It is used to work on int type value

Double >> It is used to work on double type value

And so on.

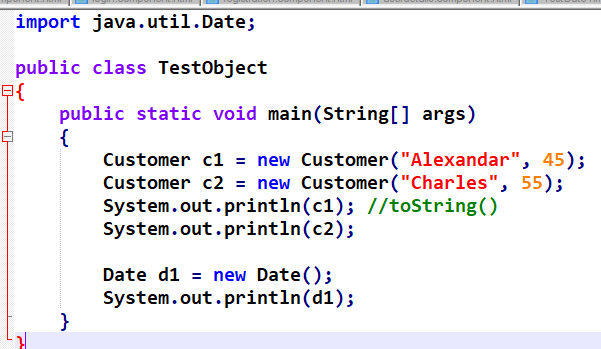
Object: It is a top most class in java, whose methods are available for all the classes.



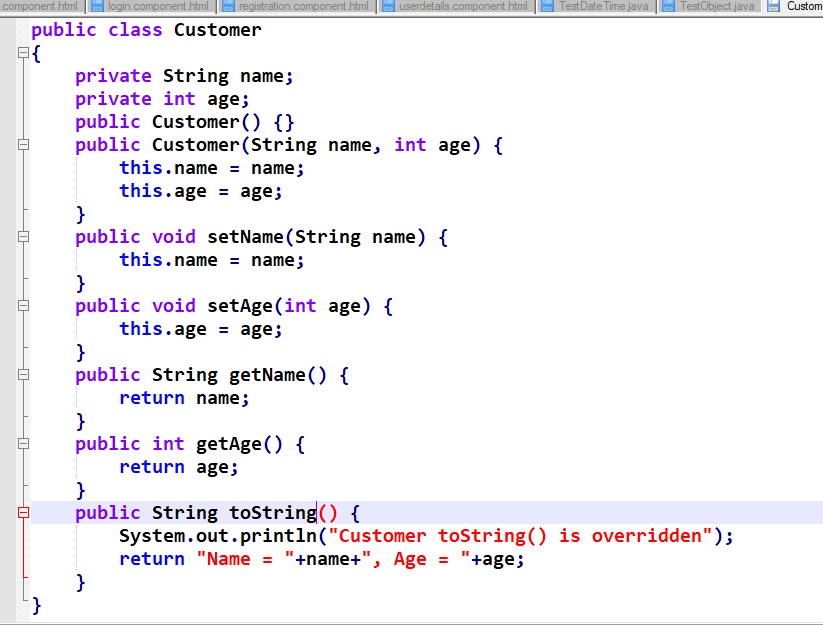
Some methods are:

toString()>> it is automatically called when you print object by default it prints memory address i.e., toString() method has an implementation inside object class to return memory address, but you can override to print your objects information’s.

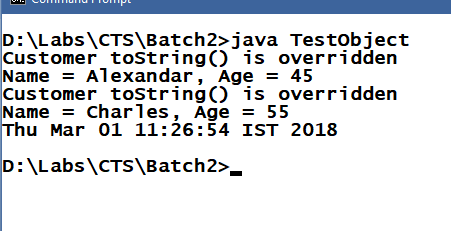
TestObjects.java



Customer.java



Output:



equals(Object): It is a method present in Object class used to compare an object with another object, the default implementation of equals() is to compare memory address of two objects, but you can override to compare object’s property in your class.

Signature: public boolean equals(Object)

Customer.java

public class Customer

{

private String name;

private int age;

public Customer() {}

public Customer(String name, int age) {

this.name = name;

this.age = age;

}

public void setName(String name) {

this.name = name;

}

public void setAge(int age) {

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public String toString() {

System.out.println("Customer toString() is overridden");

return "Name = "+name+", Age = "+age;

}

public boolean equals(Object obj) {

Customer c = (Customer)obj;

if(c.getAge() == this.getAge()) {

return true;

} else {

return false;

}

}

}

TestObject.java

import java.util.Date;

public class TestObject

{

public static void main(String[] args)

{

Customer c1 = new Customer("Alexandar", 45);

Customer c2 = new Customer("Charles", 55);

Customer c3 = new Customer("Charles", 55);

System.out.println(c1); //toString()

System.out.println(c2);

System.out.println(c3);

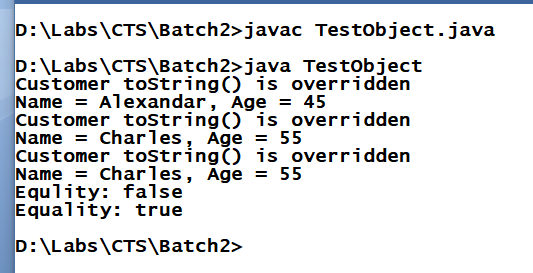
System.out.println("Equality: "+c1.equals(c2));

System.out.println("Equality: "+c2.equals(c3));

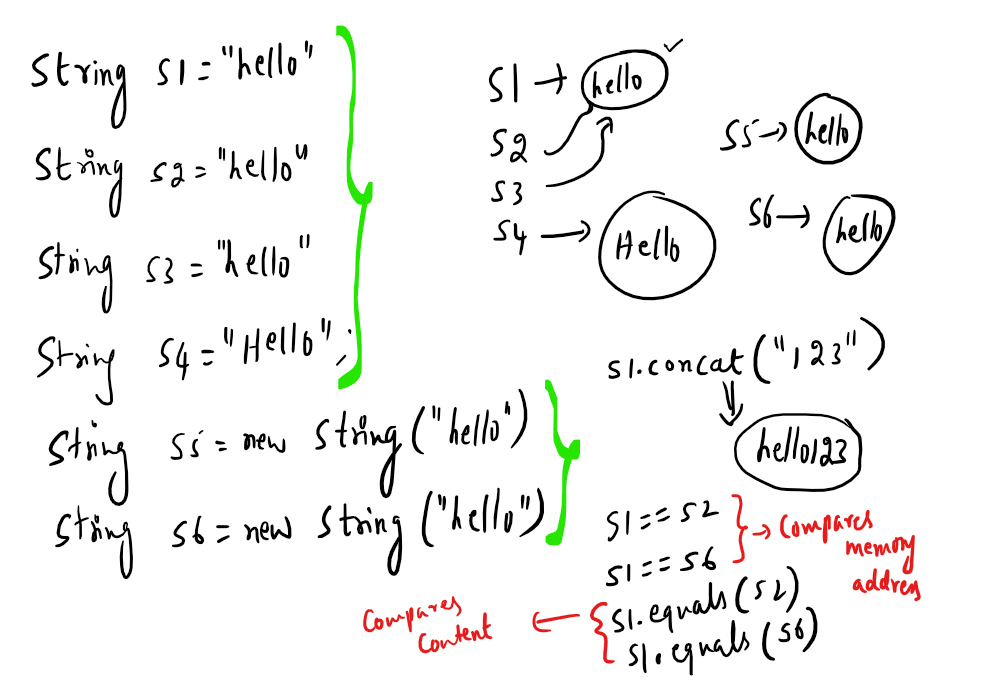
}

}

Object:

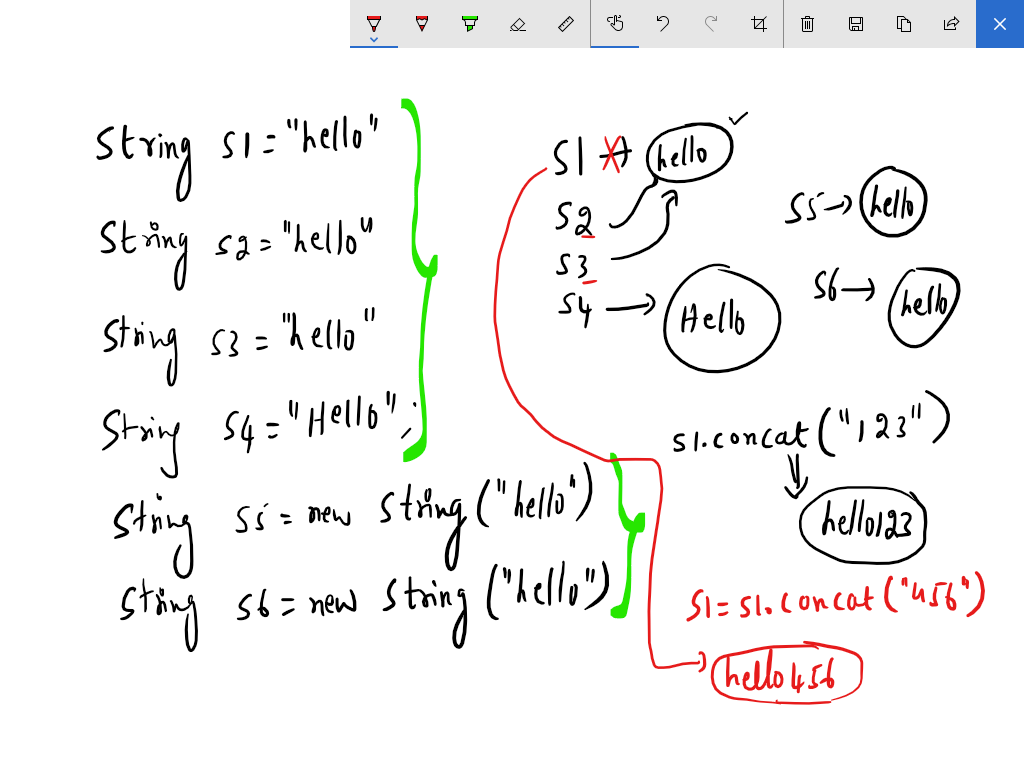


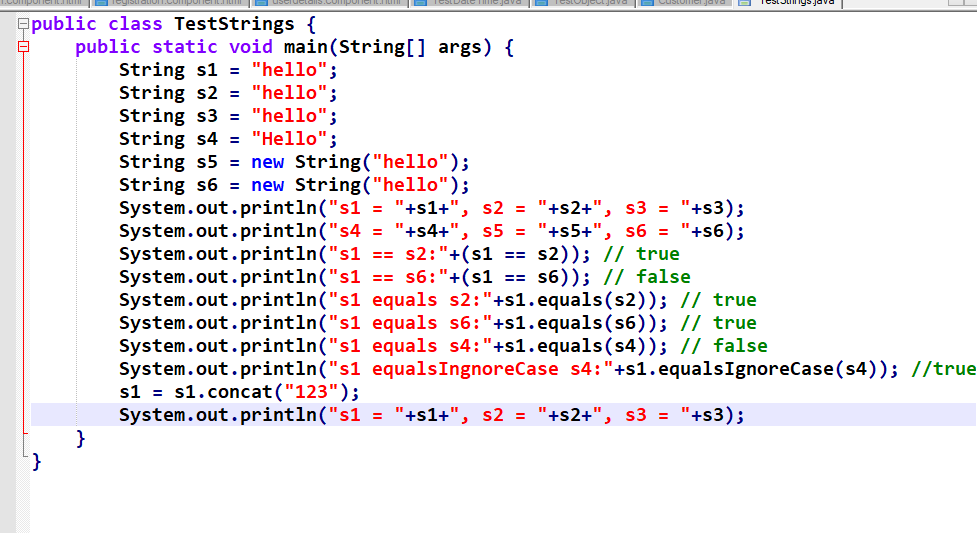
String:

It is used to work on strings, it will create immutable string objects (whose content can’t be modified once it is created)

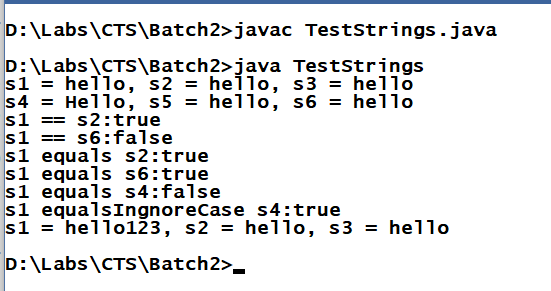
In String class equals() method of Object class has been overridden to compare content of string objects and also toString() has been overridden as well so that when print string object it prints content.

Note: When you concat a string with another string a new string will be created same thing happens when you modify string to upper case or lowercase



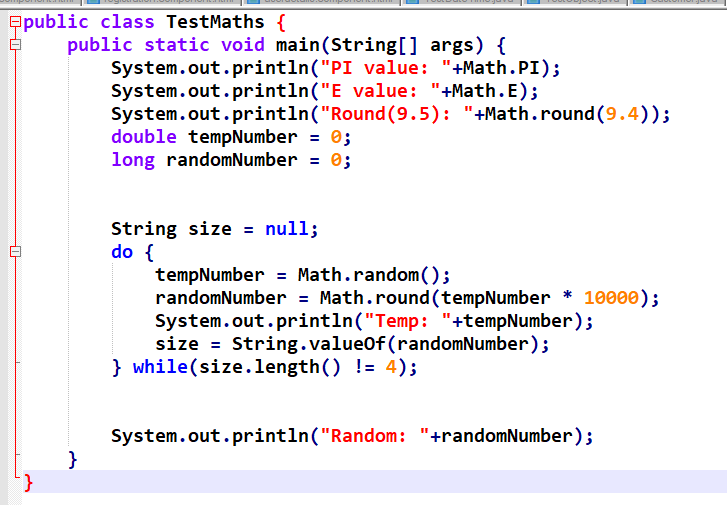


Output:

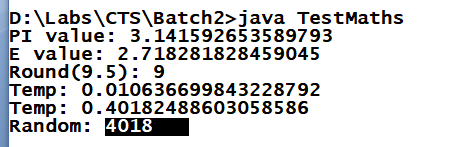


Math class:

It can be used to perform any mathematical operations like round(), PI, ceil(), random(), floor(), E, abs() and etc



Output:



Collection Framework

What is Collection?

It is a container which can have group of elements of different datatypes.

Array is a container which can have group of elements of same datatypes.

Some of the problems you would face if you array.

* Array size is fixed, i.e., int[] a = new int[5]; here you can only store 5 elements maximum, but at runtime if you want to store more than 5 elements then it raises ArrayIndexOutOfBoundsException.
* Array can store only similar type of elements
* Manipulation of an array has to be taken care by programmers like sorting, searching, iterating, storing and etc.

Collection Framework - It provides set of API’s which can maintain elements in various way and it also provides lot of best algorithms for searching, sorting, iterating, removing and many more.

Advantage of Collections

* Its size is not fixed, it increases at runtime.
* It can store any type of elements

In Collection you have a ROOT interface called Collection, which provides all the common methods to maintain the elements like: add(), remove(), iterator(), clear(), size() and so on.

Collection is extended by 3 other interfaces called List, Set and Queue to specify few more rules on the elements.

List: It specifies all the elements must be in maintained sequential order and indexed, it supports duplicate elements.

Set: It specifies all the elements must be unique no duplicates are allowed.

Queue: It is used to achieve FIFO order or based on priority means elements are processed after removing from the queue.

Set is implemented by HashSet, LinkedHashSet and TreeSet

HashSet: Order is not maintained but retrieval is faster.

LinkedHashSet: Order is insertion order

TreeSet: Sorted Order.

All three supports only unique elements.

List is implemented by ArrayList and LinkedList, it supports index, supports duplicates and order is insertion order.

Queue is implemented by LinkedList & PriorityQueue, use Queue when you want to process elements while removing.

PriorityQueue removes in Sorted Order.

LinkedList removes in FIFO.

List of methods in Collection:

add(Object)

remove(Object)

iterator()

size()

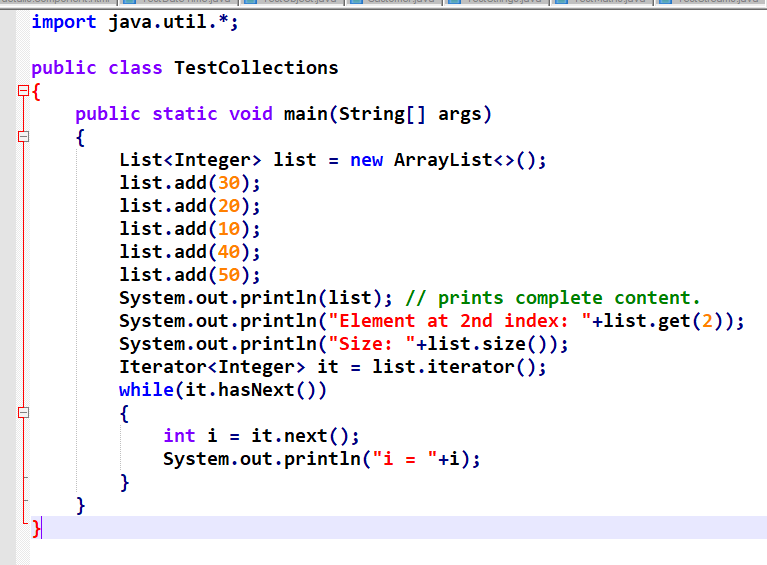
clear() and so on.

There is another API called Collections which is a class that provides operations like swap(), shuffle(), reverse(), sort and many more.

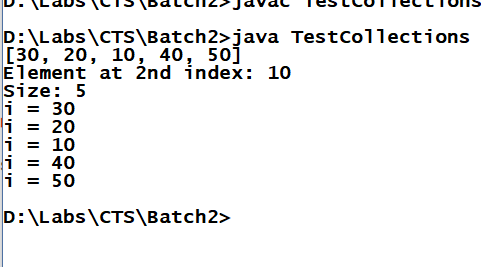
Iterator is an interface used to traverse each & every item from the collection.

All these API’s are present in java.util.\* package.

TestCollections.java



Output:

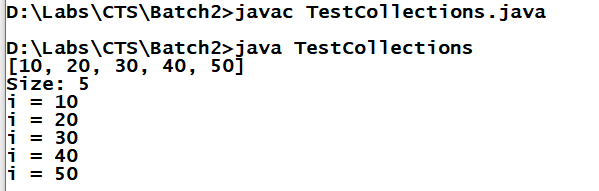


Change the reference from List to Set to Queue to observer different Output:

TestCollections.java

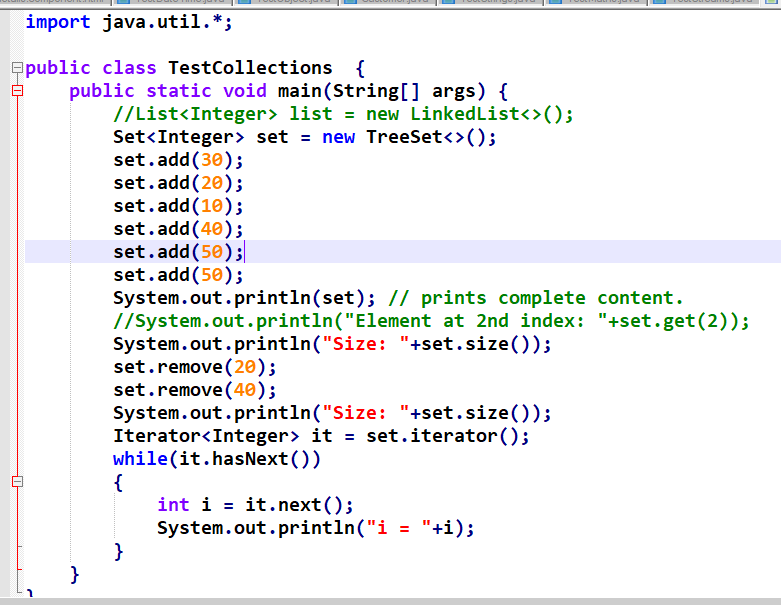


Output:

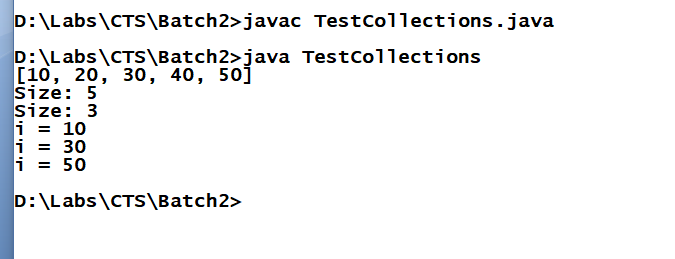


Since Set doesn’t support duplicates 50 is added only once.

If you want to remove items from Set you can directly pass the element inside remove as below



Output:



Note:

In List implementations remove(int) means remove by index, remove(Object) means remove the object.

In Set or Queue implementations there is no index based so remove(int) is not present which is why you use remove(Object).

Most of the times you would be maintaining complex objects in the collection, so you must create a Collection instance that maintains complex objects and you can use Iterator to traverse on each and every complex object to perform search, update or remove kind work.

Employee.java

public class Employee

{

private int id;

private String name;

private double salary;

public Employee() {}

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

this.id = id;

this.name = name;

this.salary = salary;

}

public void setName(String name) {

this.name = name;

}

public void setSalary(double salary) {

this.salary = salary;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public double getSalary() {

return salary;

}

}

TestCollections.java

import java.util.\*;

public class TestCollections {

public static void main(String[] args) {

List<Employee> list = new ArrayList<>();

for(int i = 100; i <= 105; i++) {

Employee emp = new Employee(i, "Name"+i, 5 \* i);

list.add(emp);

}

Iterator<Employee> it = list.iterator();

System.out.println("Size: "+list.size());

while(it.hasNext())

{

Employee e = it.next();

if(e.getId() == 102)

{

it.remove();

}

System.out.println(e.getId()+" "+e.getName()+" "+e.getSalary());

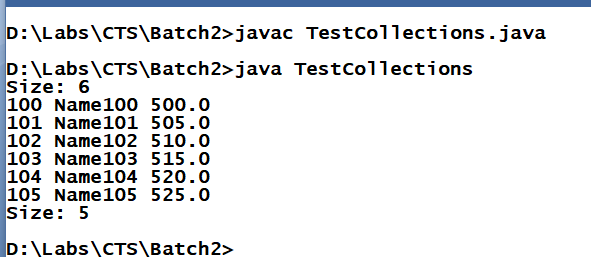
}

System.out.println("Size: "+list.size());

}

}

Output:



Using Queue implementations LinkedList and PriorityQueue

TestQueues.java

import java.util.\*;

public class TestQueues

{

public static void main(String[] args)

{

Queue<String> queue = new PriorityQueue<String>();

queue.add("ABC");

queue.add("abc");

queue.add("Abc");

queue.add("ABc");

queue.add("aBC");

System.out.println("Size: "+queue.size());

String elementPolled = queue.poll();

System.out.println("Element removed: "+elementPolled);

elementPolled = queue.poll();

System.out.println("Element removed: "+elementPolled);

elementPolled = queue.poll();

System.out.println("Element removed: "+elementPolled);

elementPolled = queue.poll();

System.out.println("Element removed: "+elementPolled);

elementPolled = queue.poll();

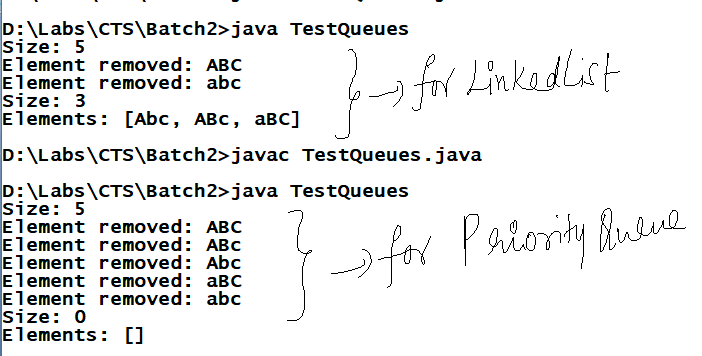
System.out.println("Element removed: "+elementPolled);

System.out.println("Size: "+queue.size());

System.out.println("Elements: "+queue);

}

}

Output:  


In LinkedList we have polled only 2 elements but in PriorityQueue we have polled all the elements which is why the first output is showing size 3 and the next output is showing size as 0.

Map: It is works like Collection but it is not part of Collection, it also has some implementations like Set interface.

Map is an interface which can store elements as key and value, using key you can store, retrieve and remove element, here key is unique and value can be duplicate.

Map has 3 implementations HashMap, LinkedHashMap and TreeMap, they all work like Set implementations HashSet, LinkedHashSet and TreeSet but Set stores only elements whereas Map stores keys and values.

Some of the methods of Map are:

put(key, value): It is used to store elements using a key

get(key): It is used to retrieve element based on key

remove(key): It is used to remove element based on key

keySet(): It returns Set of keys and stores in Set interface

values(): It returns Collection of values and stores in Collection interface

entrySet(): It returns both keys and values and stores in a Set interface.

The three implementations of Map are HashMap, TreeMap and LinkedHashMap, apart from these 3 implementations there is an Hashtable which implements Map but it is a older implementation of Map which works exactly like HashMap.

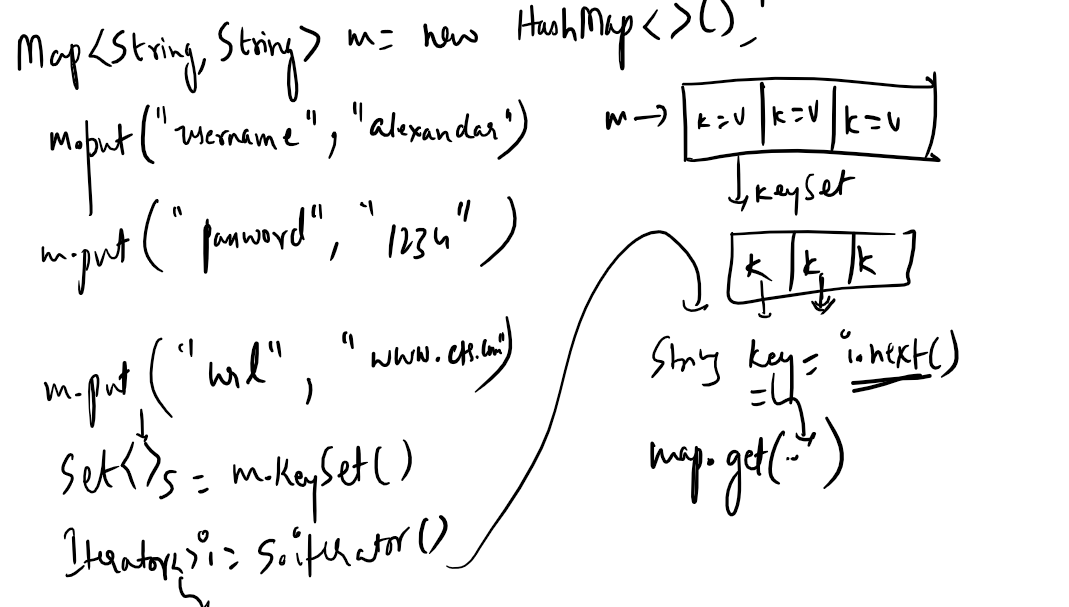
Totally there are 4 implementations HashMap, Hashtable, LinkedHashMap and TreeMap

HashMap: order is not maintained but retrieval is faster, it supports null

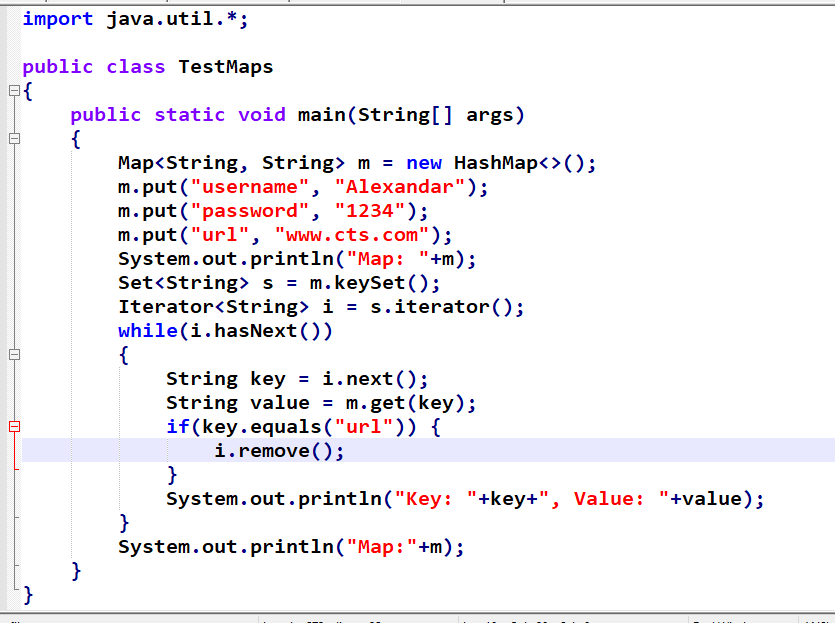
Hashtable: Same like HashMap but is a older API and it doesn’t support null

TreeMap: Sorts the key

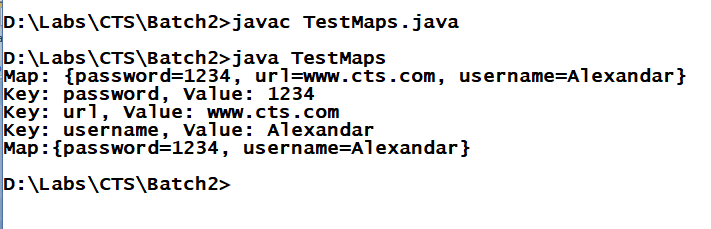
LinkedHashMap: Insertion order is maintained.



TestMaps.java



Output:



Database: Collection of logically related data.

List of databases you have are:

MySQL

IBM DB2

Oracle

Sybase

Derby and so on.

All these databases are called as relational (table) database which stores data in table format.

Some of the important points on database

* Every database understand a language called SQL
* But every database has its own SQL structure/datatypes
* example In oracle you have varchar2 and varchar for strings, whereas in MySQL you have only varchar for strings, same way you may have float, double, integer types in one database but you may not see the same in other database.
* It means SQL is database dependent language.

In MySQL you must create database and use that database to create any tables:

Usage of create database, use and show tables:

>>create database demo;

>>use demo;

>>show tables;

Hibernate Framework

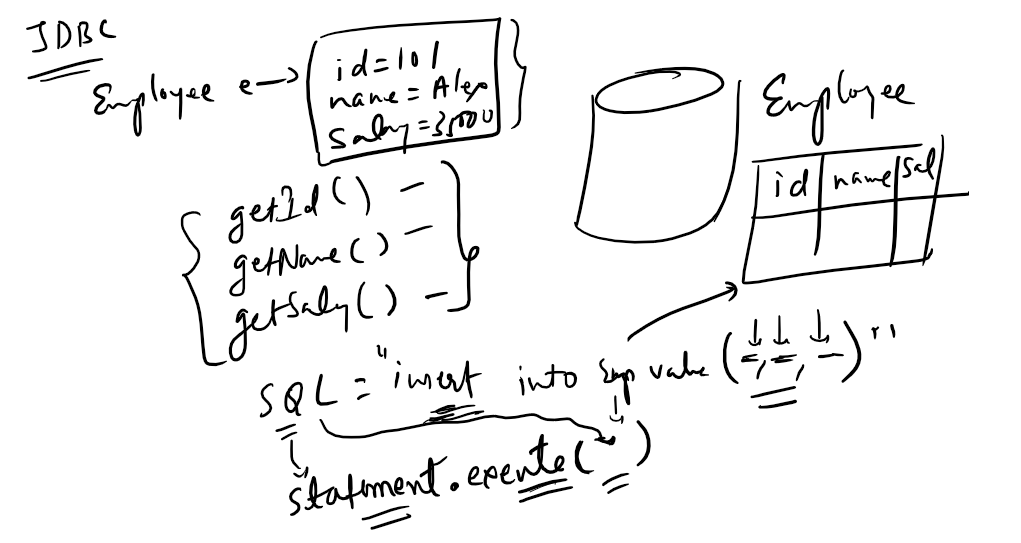
Hibernate framework is Java framework to interact with any database.

What is Framework?

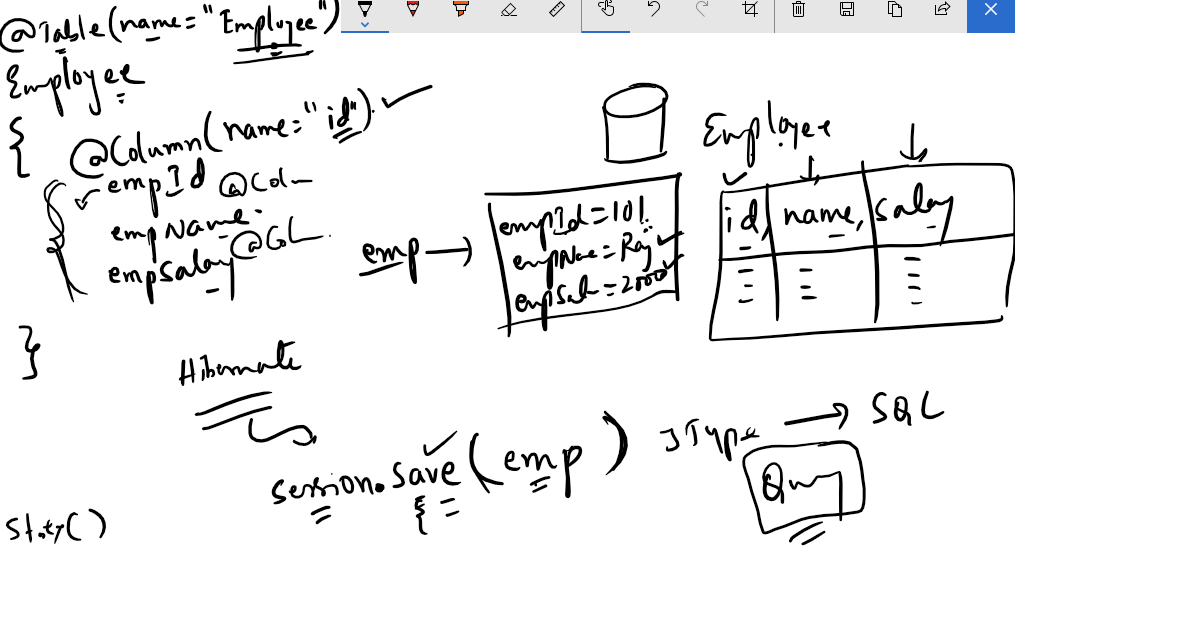
* Simplifies the development process by reducing developer work
* It is like a ready-made application which will do lot of tasks which developers don’t have to concentrate on like type conversions, exception handling, validation, Object creation, security and many more.
* Some of the list of frameworks are:
  + Struts >> Web framework (now it is not used for new projects)
  + Spring >> Application framework (develop web application, desktop application, enterprise application)
  + Hibernate >> it is mainly for database operations

Some important points in Hibernate.  
- Hibernate framework is used to map java objects directly to relational database table which is why it is called as ORM framework i.e, Object-Relational-Mapping.

In JDBC you need to convert Java Objects to SQL types and execute query as below:

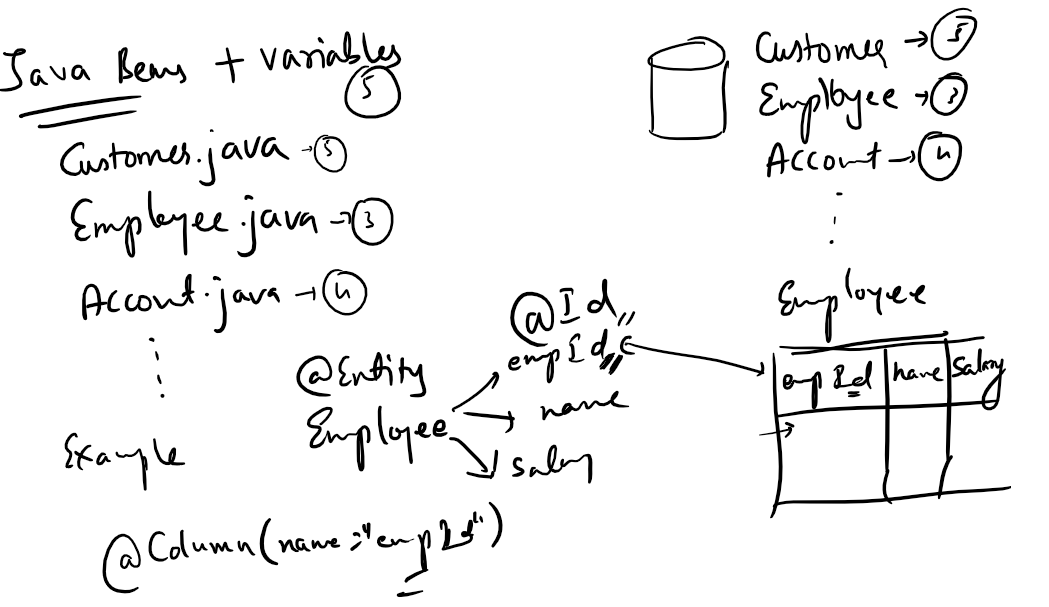


But in hibernate you can directly store java objects so that the framework takes care of converting java types to SQL types and mapping the values to appropriate column



* Hibernate generates the query according to the target database so that you don’t have to write SQL query which is database dependent so the Hibernate is database independent because it generates the query according to the database.

How to write programs using Hibernate to operate on database.



* You must create a java bean which will have @Entity annotation marked, which tells framework that the Java object is a representation of database table.

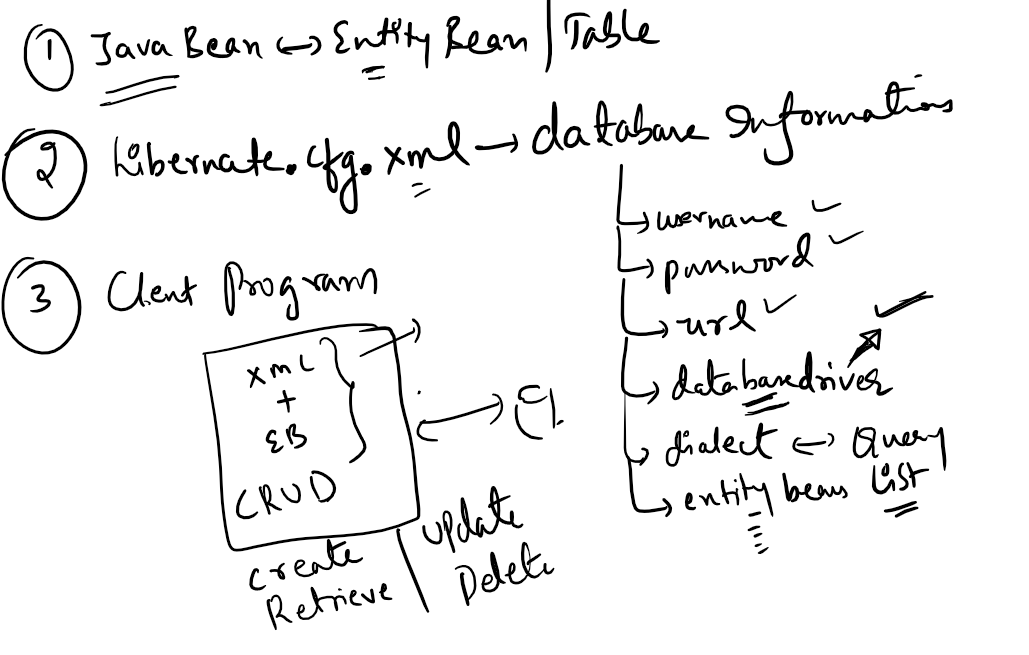
@Table on a class is optional when table name and class name are same.

@Id is mandatory on a variable which maps to the primary key column.

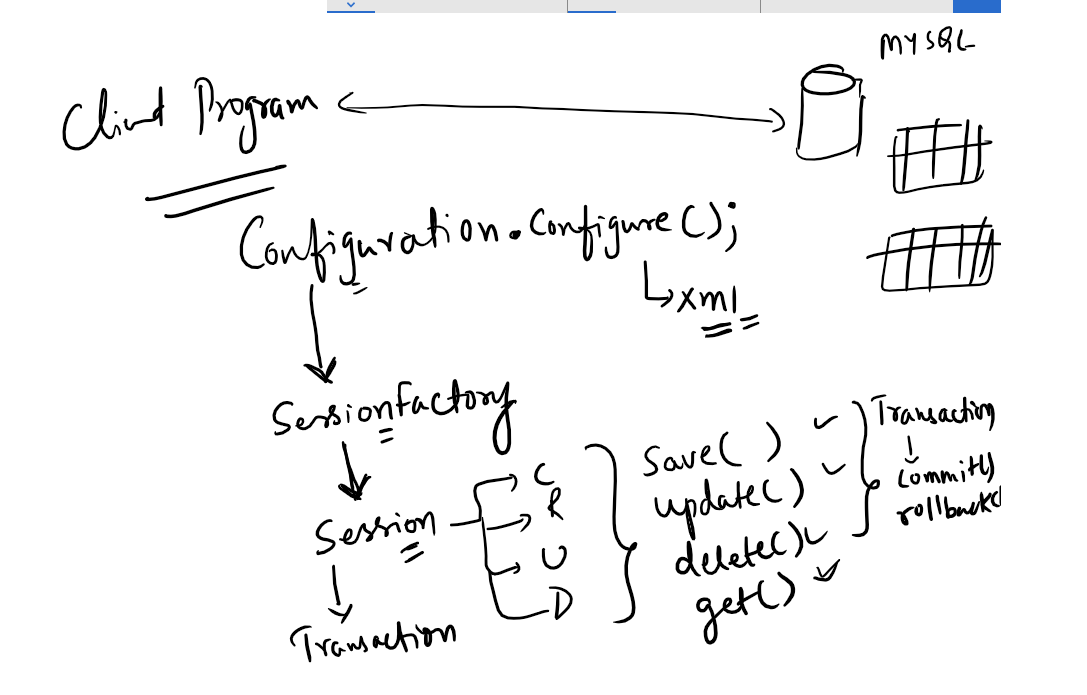
@Column is optional if the variable name is same as the column name of a table.

In Hibernate one java object is equal one SQL record,

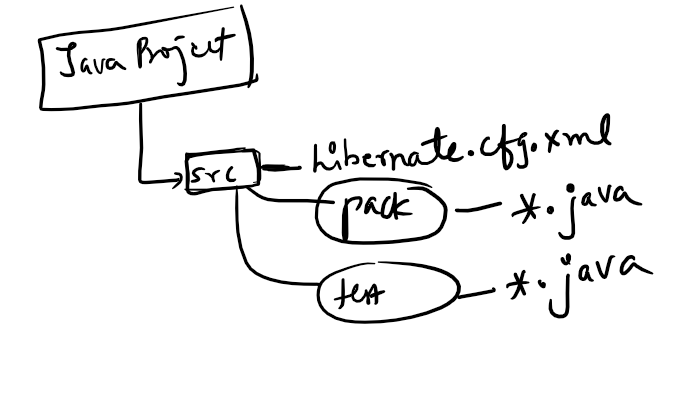
A variable in the java object is equal a column of an SQL record.



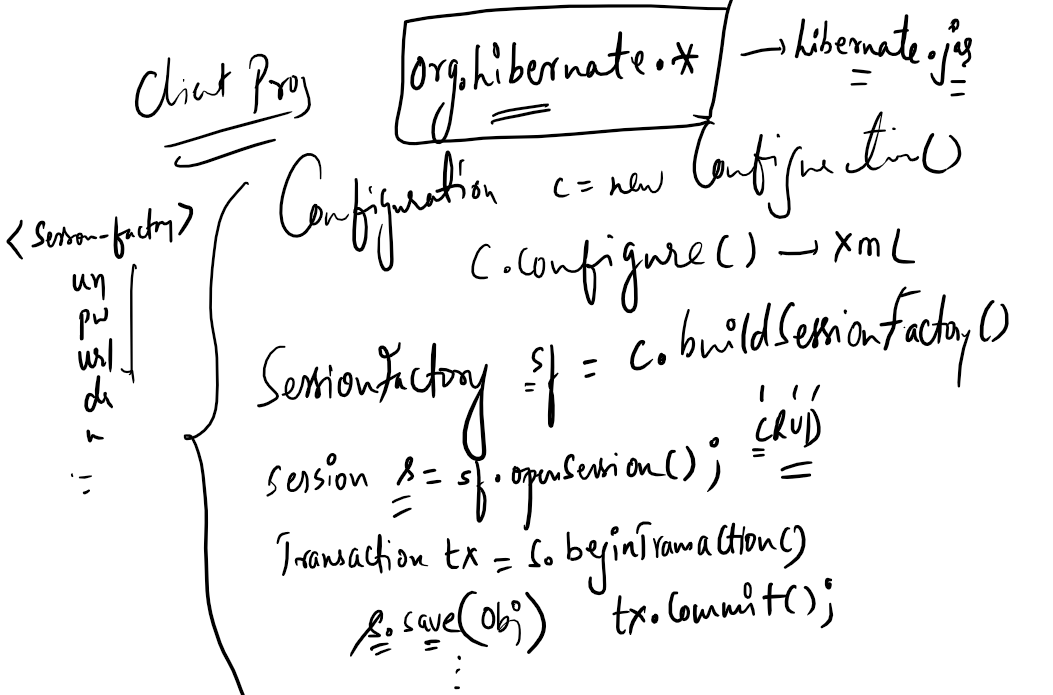
How the client program is written



* Configuration instance will load the hibernate.cfg.xml file using configure() method.
* SessionFactory is created from Configuration instance which provides Connection to the database
* Session is created to perform CRUD operations from SessionFactory.
* Transaction must be created for Insert, Update and Delete operations.



Client programs and Java beans will be present in src folder only



Employee.java

**package** pack;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

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

@Id

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

**private** String name;

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

**public** Employee() {

**super**();

}

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

**super**();

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

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

**return** salary;

}

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

**this**.salary = salary;

}

}

hibernate.cfg.xml

<?xml version=*'1.0'* encoding=*'utf-8'*?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name=*"connection.driver\_class"*>com.mysql.jdbc.Driver</property>

<property name=*"connection.url"*>jdbc:mysql://localhost:3306/demo</property>

<property name=*"connection.username"*>root</property>

<property name=*"connection.password"*>root</property>

<!-- SQL dialect -->

<property name=*"dialect"*>org.hibernate.dialect.MySQL5Dialect</property>

<!-- Echo all executed SQL to stdout -->

<property name=*"show\_sql"*>false</property>

<!-- Create table if doesn't exist else Update the table -->

<property name=*"hbm2ddl.auto"*>update</property>

<mapping class=*"pack.Employee"*/>

</session-factory>

</hibernate-configuration>

TestHibernate.java

**package** pack;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**import** org.hibernate.cfg.Configuration;

**public** **class** TestHibernate {

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

Configuration c = **new** Configuration();

c.configure();

SessionFactory sf = c.buildSessionFactory();

Session s = sf.openSession();

/\*

Transaction tx = s.beginTransaction();

Employee emp = new Employee(666, "Daniels", 45000);

s.save(emp);

tx.commit();

\*/

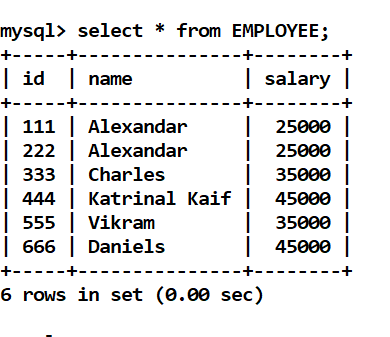
s.close();

sf.close();

System.***out***.println("\*\*\*\*\*\*DONE\*\*\*\*\*\*");

}

}

Output:  


How to retrieve, update and delete using Session instance.

Usage:

Employee e = s.get(Employee.class, 111)

Usage:

Session s = sf.openSession();

Transaction tx = s.beginTransaction();

Employee e = s.get(Employee.class, 111);

Employee e2 = s.get(Employee.class, 444);

e.setName(“newName”);

e.setSalary(newSalary);

tx.commit;

//close session & session factory.

You can use java.util.Scanner class, to take input from keyboard.  
How to create Scanner Object

Scanner sc = new Scanner(System.in);

// to read int

int value = sc.nextInt();

// to read String

String value = sc.next() or sc.nextLine()

// to read double

double value = sc.nextDouble();

// to read float

float value = sc.nextFloat();

// at the end you need to close your input stream

sc.close();

You can have a reusable code to get a SessionFactory as below:

HibernateUitity.java

**package** pack;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

**public** **class** HibernateUtility {

**public** **static** SessionFactory getSessionFactory() {

SessionFactory sf = **null**;

Configuration cfg = **new** Configuration();

cfg.configure();

sf = cfg.buildSessionFactory();

**return** sf;

}

}

TestCRUD.java

**package** pack;

**import** java.util.Scanner;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**public** **class** TestCRUD {

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

System.***out***.println("Doing update operation by passing id at runtime");

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

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Transaction tx = s.beginTransaction();

System.***out***.println("Enter an id:");

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

sc.nextLine();

Employee e = s.get(Employee.**class**, tempId);

**if**(e != **null**) {

System.***out***.println("Old name: "+e.getName());

System.***out***.println("Enter name to update");

String tempName = sc.nextLine();

e.setName(tempName);

System.***out***.println("New name: "+e.getName());

s.update(e);

tx.commit();

} **else** {

System.***err***.println("Sorry, Id "+tempId+" not found!");

}

s.close();

sf.close();

sc.close();

}

}

Output:  
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnectionAccess@38a1c423] for (non-JTA) DDL execution was not in auto-commit mode; the Connection 'local transaction' will be committed and the Connection will be set into auto-commit mode.

Enter an id:

222

Old name: Bob

Enter name to update

Bobby

New name: Bobby

Mar 05, 2018 10:29:02 AM

Same way you can perform delete operation using session.delete(obj);  
example: s.delete(e);

Note: To perform update or delete you must retrieve the object

Criteria API  
It is an instance to get collection of entities based on some conditions (criteria), In Criteria you get a Collection of List type which maintains collection of entities.

How to instantiate Criteria?  
Criteria crt = s.createCriteria(class);

List list = crt.list();

Example:

Criteria crt = s.createCriteria(Employee.class);

List list = crt.list(); // now you get all the records because there is no condition or criteria specified.

How to specify Criteria?  
1st Scenario:

Criteria crt = s.createCriteria(Employee.class);

crt.add(Restrictions.lt(“salary”,25000));

List list = crt.list(); // now you get only those records whose salary is less than 25000

2nd Scenario:

Criteria crt = s.createCriteria(Employee.class);

crt.add(Restrictions.between(“id”, 100, 200));

crt.add(Restrictions.gt(“salary”,30000));

List list = crt.list();

Now you can use for loop or Iterator to iterate the List

TestCriteria.java

**package** pack;

**import** java.util.Iterator;

**import** java.util.List;

**import** org.hibernate.Criteria;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.criterion.Restrictions;

**public** **class** TestCriteria {

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

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Criteria crt = s.~~createCriteria~~(Employee.**class**);

crt.add(Restrictions.*between*("id", 100, 600));

crt.add(Restrictions.*gt*("salary", 30000.0)); // double->Double

List list = crt.list();

Iterator it = list.iterator();

**while**(it.hasNext()) {

Employee e = (Employee) it.next();

System.***out***.println("Id = "+e.getId()+", Name = "+e.getName()

+", Salary = "+e.getSalary());

}

s.close();

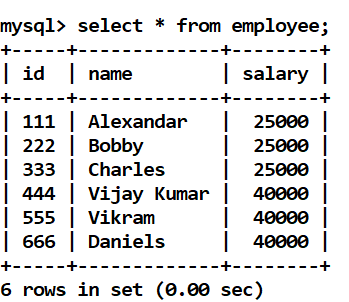
sf.close();

}

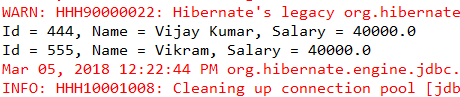
}

Output:

Records in the table

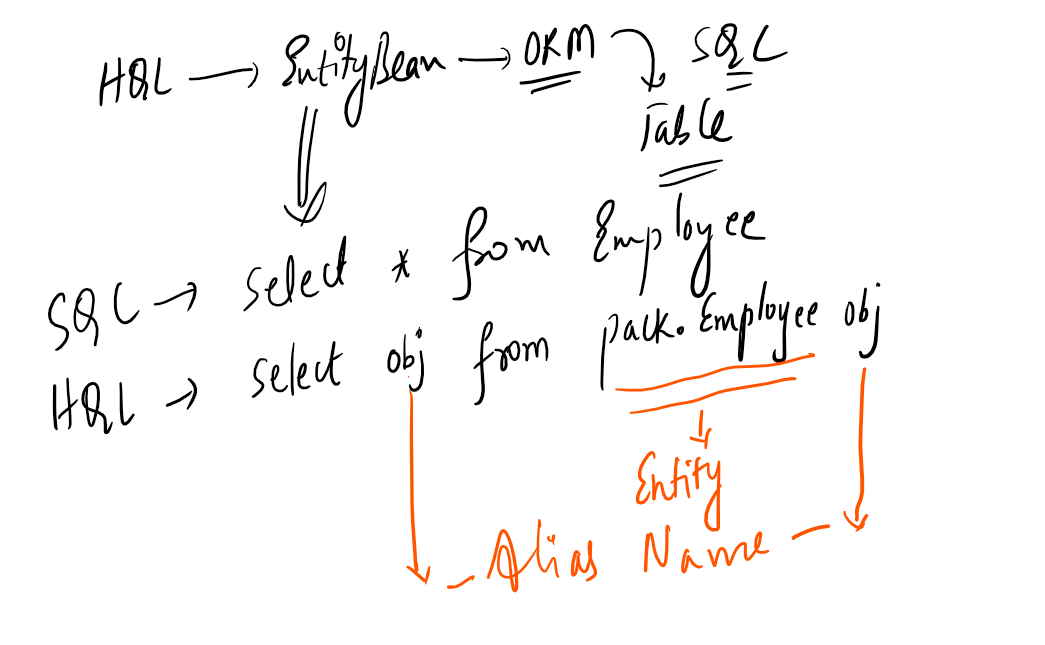


Console output



Hibernate Query Language(HQL)

It is a language used to query the Entity Bean such that the framework will query the table using SQL which is understood by the database.



Here obj is the alias name if you use alias name in between select and from then it is considered as all the properties of entity bean.

Usage:

select obj.name,obj.salary from pack.Employee obj

Now you are going get only name and salary of Employee object.

How to execute Hibernate Query?

You can use Query interface to pass the query.

Usage:

String hql = “select obj from pack.Employee obj”

Query qry = s.createQuery(hql);

List list = qry.list(); // same like criteria

Difference between HQL and Criteria

Criteria allow you to retrieve all the entity properties, you cannot retrieve only few properties from the entity bean, it always queries all the columns.

Whereas HQL can filter the entity bean properties so that it can query only few columns which is important because you don’t want to show some properties to some users for ex: hiding salary from other users.

TestHQL.java

**package** pack;

**import** java.util.Iterator;

**import** java.util.List;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.query.Query;

**public** **class** TestHQL {

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

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

String hibernateQuery = "select obj from pack.Employee obj where obj.salary > 30000";

Query query = s.createQuery(hibernateQuery);

List list = query.list();

Iterator it = list.iterator();

**while**(it.hasNext()) {

Employee e = (Employee) it.next();

System.***out***.println(e.getId()+" "+e.getName()+" "+e.getSalary());

}

System.***out***.println("\*\*\*\* To retrieve only name of Employee \*\*\*\*");

String secondQuery = "select obj.name from pack.Employee obj";

Query query2 = s.createQuery(secondQuery);

List list2 = query2.list();

Iterator it2 = list2.iterator();

**while**(it2.hasNext()) {

String empNames = (String)it2.next();

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

}

sf.close();

s.close();

}

}

Output:

INFO: HHH000397: Using ASTQueryTranslatorFactory

444 Vijay Kumar 40000.0

555 Vikram 40000.0

666 Daniels 40000.0

\*\*\*\* To retrieve only name of Employee \*\*\*\*

Alexandar

Bobby

Charles

Vijay Kumar

Vikram

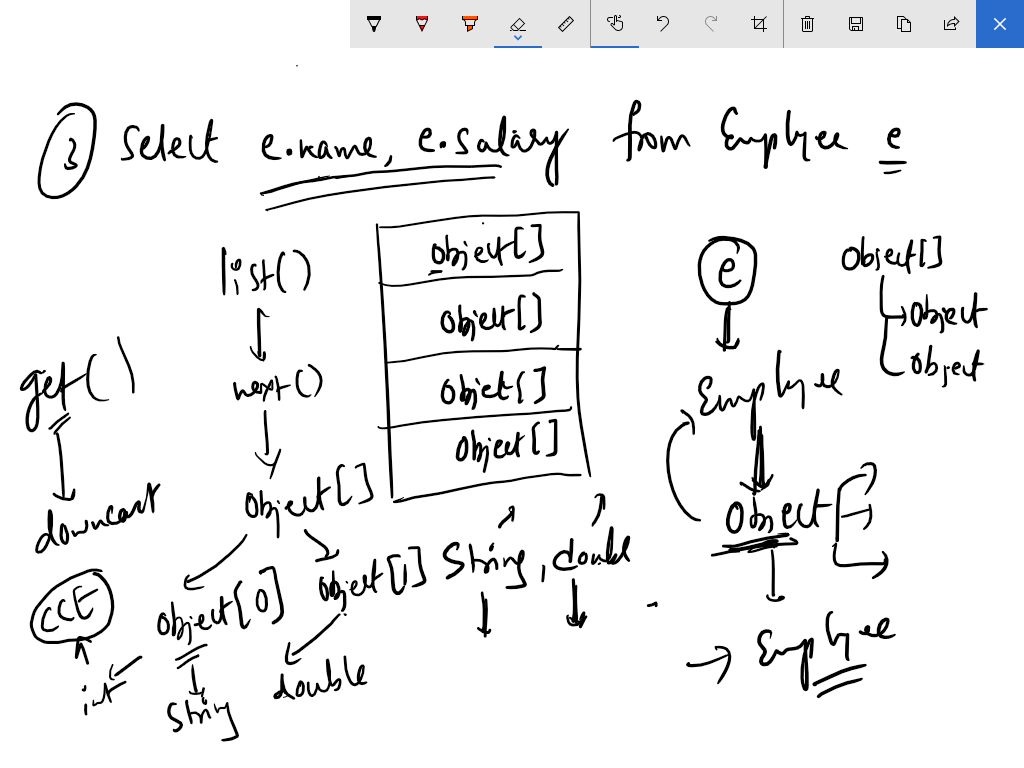
Daniels

Mar 05, 2018 12:46:21 PM

How to get only few properties from an Entity

Usage: select e.name, e.salary from pack.Employee e;

Since name is String and salary is double the list would be an array of Object i.e, Object[]



Which means on invoking next() from the Iterator you get Object[] in that there will be 2 elements which will be in Object type, which has to be retrieved from the index by properly converting each index to the right type.

TestHQL.java

**package** pack;

**import** java.util.Iterator;

**import** java.util.List;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.query.Query;

**public** **class** TestHQL {

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

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

String hql = "select e.name, e.salary from pack.Employee e";

Query query = s.createQuery(hql);

List list = query.list();

Iterator it = list.iterator();

**while**(it.hasNext()) {

Object[] obj = (Object[]) it.next();

String name = (String)obj[0];

**double** salary = (**double**)obj[1];

System.***out***.println("Name = "+name+", Salary = "+salary);

}

sf.close();

s.close();

}

}

Output:

INFO: HHH000397: Using ASTQueryTranslatorFactory

Name = Alexandar, Salary = 25000.0

Name = Bobby, Salary = 25000.0

Name = Charles, Salary = 25000.0

Name = Vijay Kumar, Salary = 40000.0

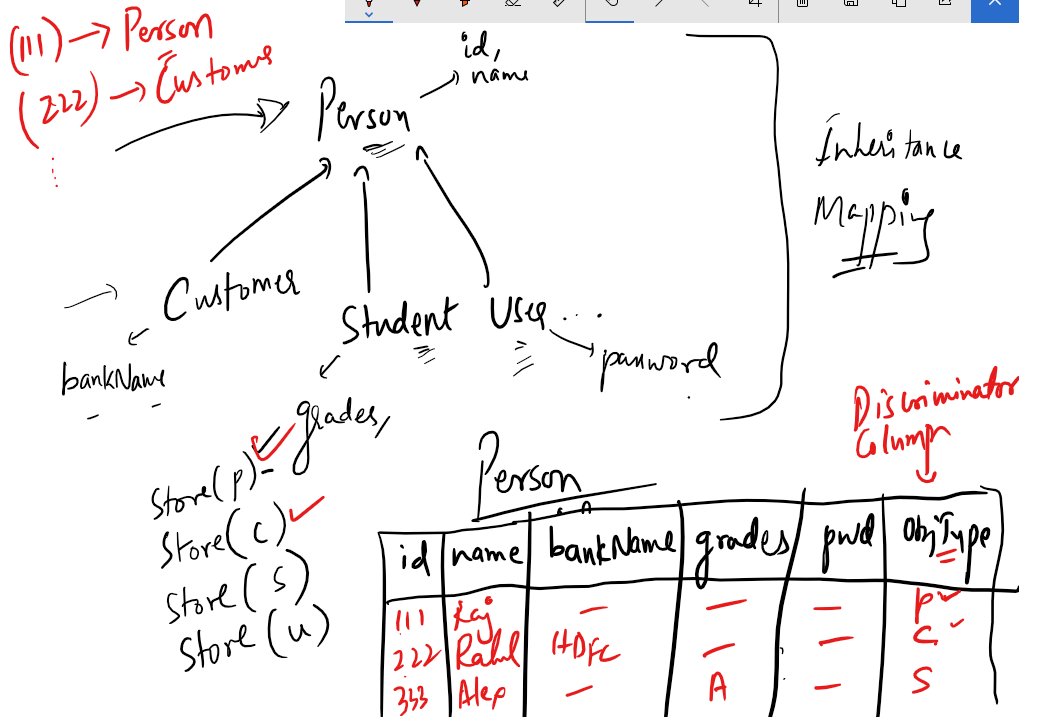
Name = Vikram, Salary = 40000.0

Name = Daniels, Salary = 40000.0

Mar 05, 2018 3:43:38 PM

Inheritance Mapping

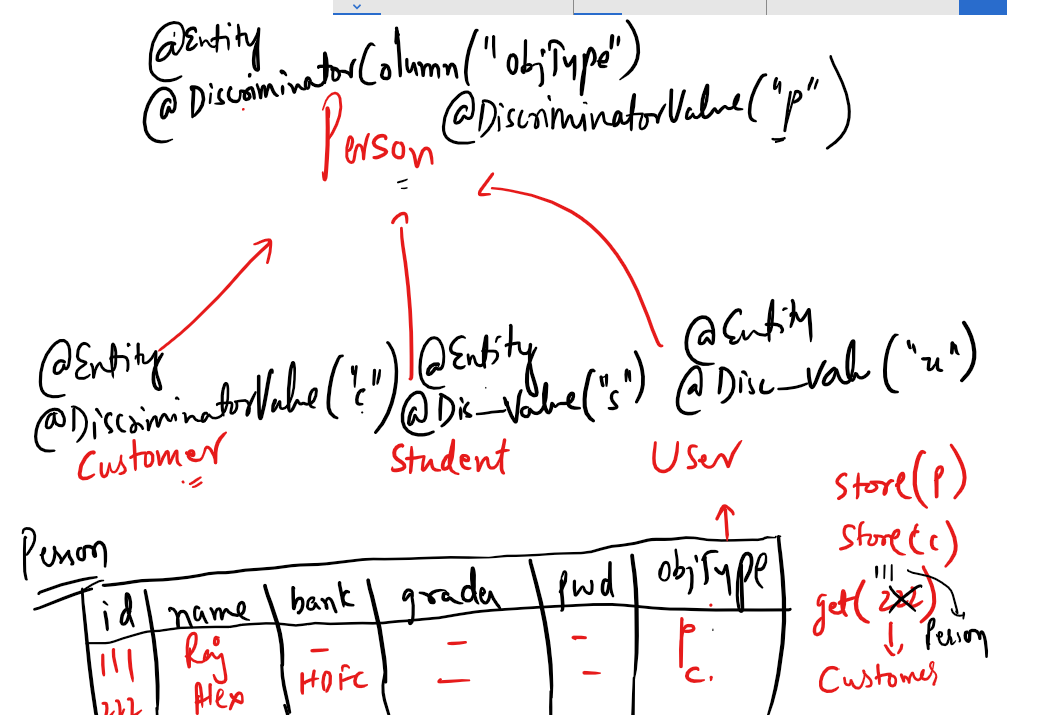
It allows you to store different types of Objects in the same table and categorizing each object in a table using one common column called discriminator column which is used to specify the type object you have in a table.



Now hibernate uses the discriminator column to return or store the right type object, for hibernate to categorize the right type of Object you must have an annotation called @DiscriminatorColumn and @DiscriminatorValue on the entity classes.

@DiscriminatorColumn comes on the class that is mapped to the table i.e., Person class but the subclasses will not have @DiscriminatorColumn annotations but it will have @Entity as it has to be stored.

@DiscriminatorValue comes on the classes whose object has to be categorized which will tell the value for the discriminator column



In hibernate table is automatically created if it doesn’t exist, it is because of

<property name=*"hbm2ddl.auto"*>update</property>

It specifies that create table if it’s not existing.

hibernate.cfg.xml

<?xml version=*'1.0'* encoding=*'utf-8'*?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name=*"connection.driver\_class"*>com.mysql.jdbc.Driver</property>

<property name=*"connection.url"*>jdbc:mysql://localhost:3306/demo</property>

<property name=*"connection.username"*>root</property>

<property name=*"connection.password"*>root</property>

<!-- SQL dialect -->

<property name=*"dialect"*>org.hibernate.dialect.MySQL5Dialect</property>

<!-- Echo all executed SQL to stdout -->

<property name=*"show\_sql"*>false</property>

<!-- Create table if doesn't exist else Update the table -->

<property name=*"hbm2ddl.auto"*>update</property>

<mapping class=*"pack.Employee"*/>

<mapping class=*"pack.Person"*/>

<mapping class=*"pack.Customer"*/>

<mapping class=*"pack.Student"*/>

</session-factory>

</hibernate-configuration>

Person.java

**package** pack;

**import** javax.persistence.DiscriminatorColumn;

**import** javax.persistence.DiscriminatorValue;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

@DiscriminatorColumn(name = "objType")

@DiscriminatorValue(value = "p")

**public** **class** Person {

@Id

**private** **int** uid;

**private** String name;

**public** Person() {

**super**();

}

**public** Person(**int** uid, String name) {

**super**();

**this**.uid = uid;

**this**.name = name;

}

**public** **int** getUid() {

**return** uid;

}

**public** **void** setUid(**int** uid) {

**this**.uid = uid;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

}

Customer.java

**package** pack;

**import** javax.persistence.DiscriminatorValue;

**import** javax.persistence.Entity;

@Entity

@DiscriminatorValue(value = "c")

**public** **class** Customer **extends** Person {

**private** String bankName;

**public** Customer(**int** uid, String name, String bankName) {

**super**(uid, name);

**this**.bankName = bankName;

}

**public** Customer() {

**super**();

}

**public** String getBankName() {

**return** bankName;

}

**public** **void** setBankName(String bankName) {

**this**.bankName = bankName;

}

}

Student.java

**package** pack;

**import** javax.persistence.DiscriminatorValue;

**import** javax.persistence.Entity;

@Entity

@DiscriminatorValue(value = "s")

**public** **class** Student **extends** Person {

**private** String grades;

**public** Student(**int** uid, String name, String grades) {

**super**(uid, name);

**this**.grades = grades;

}

**public** Student() {

**super**();

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

}

**public** String getGrades() {

**return** grades;

}

**public** **void** setGrades(String grades) {

**this**.grades = grades;

}

}

TestInheritanceMapping.java

**package** pack;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**public** **class** TestInheritanceMapping {

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

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Transaction tx = s.beginTransaction();

// create 3 different type of objects

Person p = **new** Person(111, "Raj");

Customer c = **new** Customer(222, "Rahul", "HDFC");

Student std = **new** Student(333, "Alex", "A");

s.save(p);

s.save(c);

s.save(std);

tx.commit();

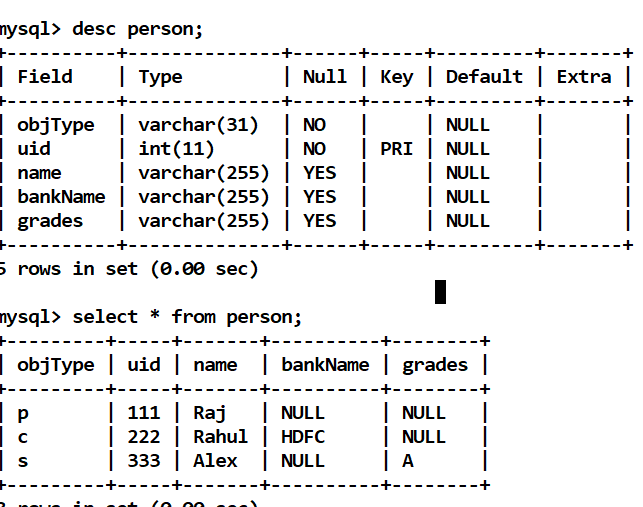
s.close();

sf.close();

}

}

Output:



Hibernate can properly instantiate the object by checking the discriminator value of discriminator column internally when you try to retrieve the Object using get() or list() .

**package** pack;

**import** java.util.Iterator;

**import** java.util.List;

**import** org.hibernate.Criteria;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**public** **class** TestDiscriminating {

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

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Criteria crt = s.~~createCriteria~~(Person.**class**);

List list = crt.list();

Iterator it = list.iterator();

**while**(it.hasNext()) {

Person p = (Person)it.next();

**if**(p **instanceof** Customer) {

Customer c = (Customer)p;

System.***out***.println("Id = "+c.getUid()+", Name = "+c.getName()

+", Bank = "+c.getBankName());

} **else** **if**(p **instanceof** Student) {

Student s1 = (Student)p;

System.***out***.println("Id = "+s1.getUid()+", Name = "+s1.getName()

+", Grades = "+s1.getGrades());

} **else** {

System.***out***.println("Id = "+p.getUid()+" Name = "+p.getName());

}

}

s.close();

sf.close();

}

}

Output:

WARN: HHH90000022: Hibernate's legacy org.hibernate.Criteria API is deprecated; use the JPA javax.persistence.criteria.CriteriaQuery instead

Id = 111 Name = Raj

Id = 222, Name = Rahul, Bank = HDFC

Id = 333, Name = Alex, Grades = A

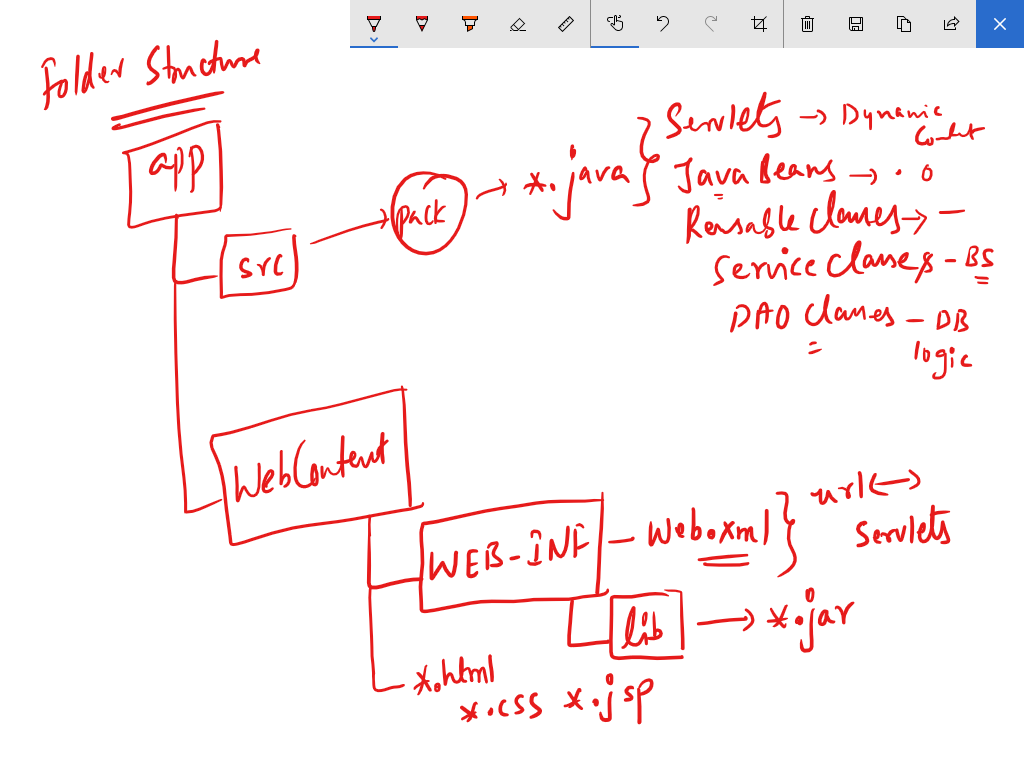
Mar 05, 2018 5:23:11 PM

Servlets: They are java programs which are executed by server to develop the dynamic web pages.

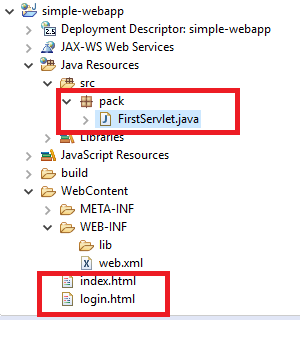
Servlets are called by server itself when user sends the request to the server.

User can send request through URL so that a corresponding Servlet which is configured for the URL get called by server.

Servlets will be stored at server side and each servlets are mapped to a particular URL



Create index.html, login.html inside Webcontent folder and FirstServlet in a package pack



FirstServlet

**package** pack;

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** javax.servlet.ServletConfig;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

/\*\*

\* Servlet implementation class FirstServlet

\*/

**public** **class** FirstServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **int** i = 0;

**private** **int** j = 0;

**public** **void** init(ServletConfig config) **throws** ServletException {

i++;

}

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

j++;

PrintWriter pw = response.getWriter();

System.***out***.println("doGet() is called");

pw.print("<html><body>");

pw.print("<p>Value of i = "+i+", Value of j = "+j+"</p></body></html>");

}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

j++;

PrintWriter pw = response.getWriter();

System.***out***.println("doPost() is called");

pw.print("<html><body>");

pw.print("<p>Value of i = "+i+", Value of j = "+j+"</p></body></html>");

}

}

index.html

<body>

<h2>This is welcome page</h2>

<a href = *"login.html"*>Click Here</a>

<hr>

<a href = *"FirstServlet"*>Call Servlet</a>

</body>

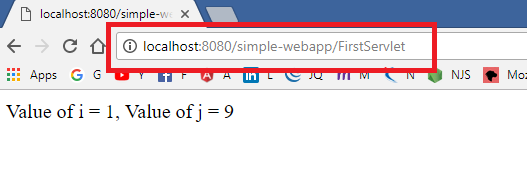
login.html

<body>

<h3>This is login page</h3>

</body>

Output:



RequestDispatcher: It is an interface used to call other web resources like HTML, JSP and Servlets

It provides some methods like

forward(request, response) and

include(request, response)

How to create RequestDispatcher object

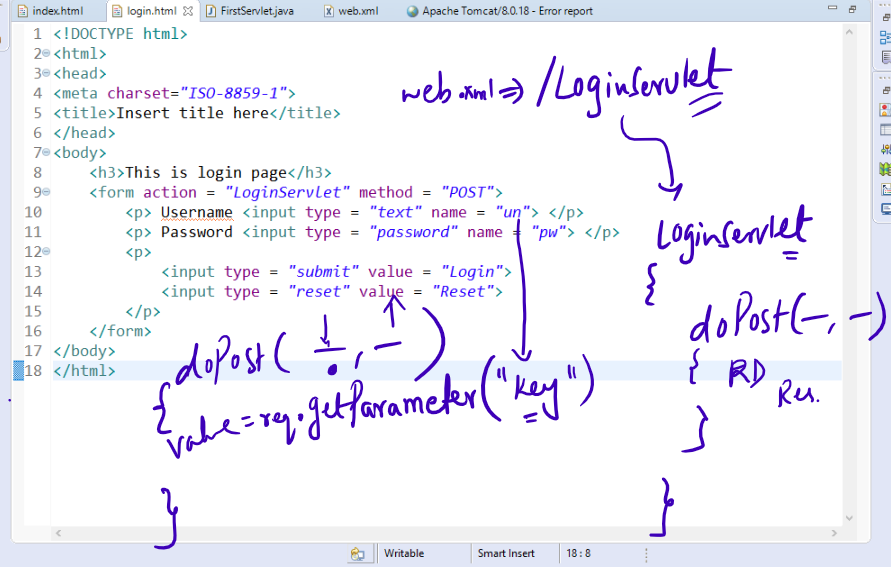
RequestDispatcher rd = request.getRequestDisptacher(“url”)

rd.forward(request, response);

Requirement: To create login form and submit to the Servlet.

Files required are

login.html



LoginServlet.java

**package** pack;

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** javax.servlet.RequestDispatcher;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** LoginServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

String user = request.getParameter("un");

String pass = request.getParameter("pw");

PrintWriter pw = response.getWriter();

**if**(user.equals("Admin") && pass.equals("cts")) {

/\*pw.print("<html><body><p>Success</p></body></html>");\*/

RequestDispatcher rd = request.getRequestDispatcher("success.html");

pw.print("<html><body>Forward Content</body></html>");

rd.forward(request, response);

} **else** {

/\*pw.print("<html><body><p>Authentication Failed!</p></body></html>");\*/

RequestDispatcher rd = request.getRequestDispatcher("login.html");

pw.print("<html><body>Authenticatio Error</body></html>");

rd.include(request, response);

}

}

}

success.html

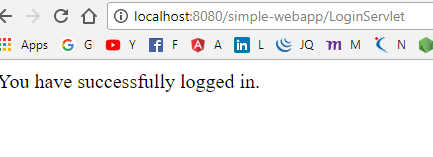
<body>

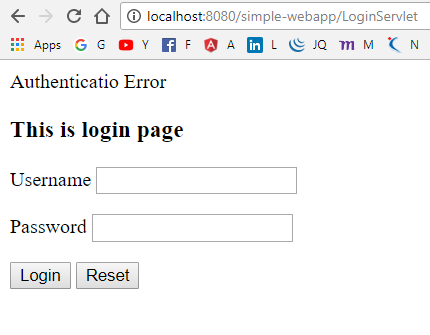
You have successfully logged in.

</body>

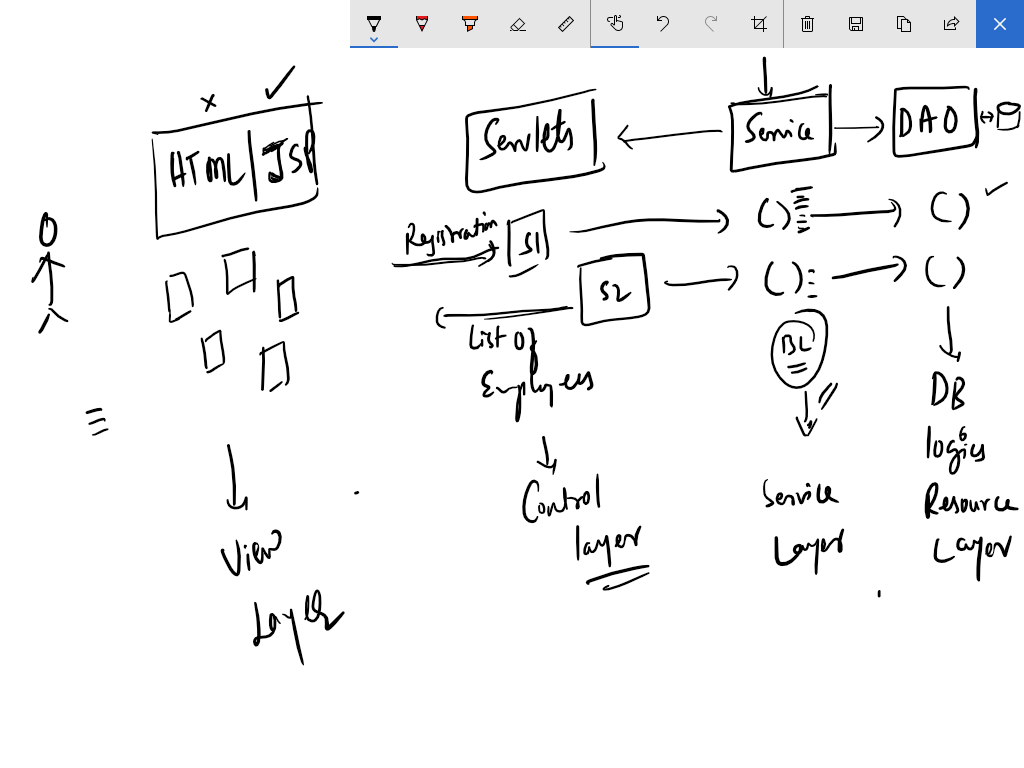
Output:

Right username and password

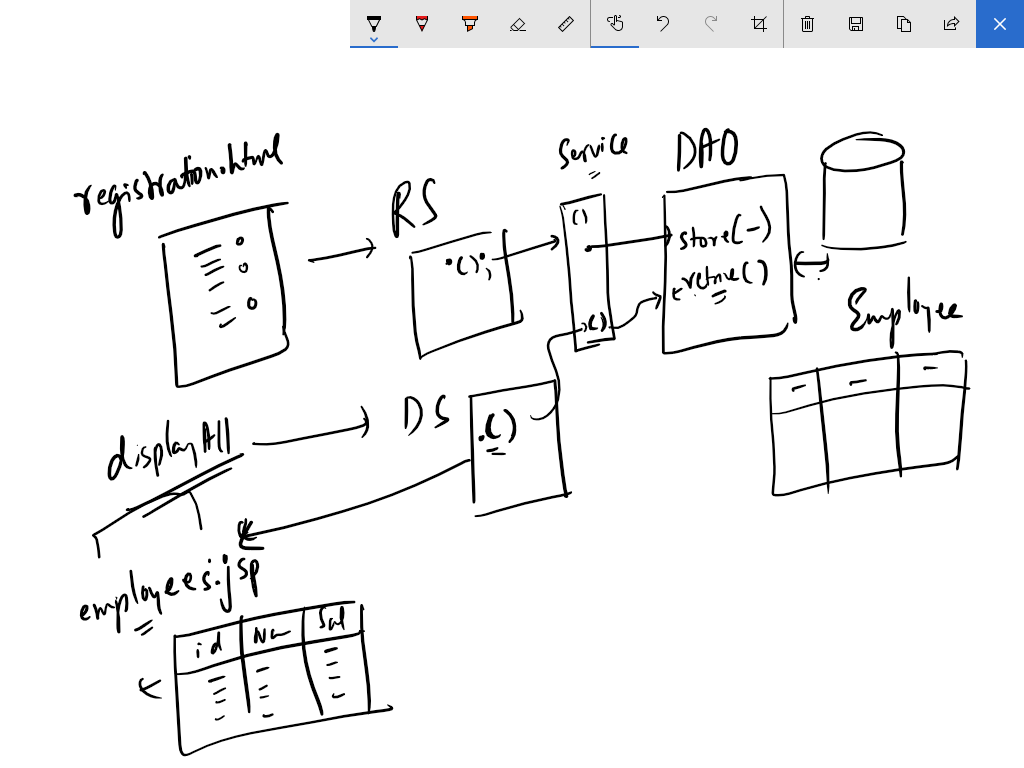
  
Wrong username and password



MVC in multi-layer architecture



Requirement:



Solution:

registration.html >> form for id, name and salary

employees.jsp >> displays employee information’s in a tabular form

index.html >> link for displayAll, registration

2 Servlets >> RegistrationServlets, DisplayServlet

EmployeeService.java >> business logics

EmployeeDAO.java >> db logics

Pre-requisites:

Add Hibernate Jars to lib folder

Add mysql-connector jar to lib folder

Add hibernate.cfg.xml file into src folder

Now: you can have 2 packages cts.model and cts.controller where cts.model will have

Employee.java >> Entity Bean

EmployeeService >> Business Service class

EmployeeDAO >> CRUD operations

HibernateUtility >> Reusable class for Session Factory

DiplayServlet >> calls displayAll of EmployeeService and gets the List of employees which is sent to employees.jsp

RegistrationServlet >> receive employee information’s from HTML and call insert() method of EmployeeService.

Employee.java

**package** cts.model;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

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

@Id

**private** Integer id;

**private** String name;

**private** Double salary;

**public** Employee(Integer id, String name, Double salary) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

**public** Employee() {

**super**();

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

}

**public** Integer getId() {

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** Double getSalary() {

**return** salary;

}

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

**this**.salary = salary;

}

}

EmployeeDAO.java

**package** cts.model;

**import** java.util.List;

**import** org.hibernate.Criteria;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**public** **class** EmployeeDAO {

// store logic

**public** **int** store(Employee emp) {

**int** status = 0;

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Transaction tx = s.beginTransaction();

s.save(emp);

tx.commit();

status = 1;

s.close();

sf.close();

**return** status;

}

// display all

**public** List retrieveAll() {

List list = **null**;

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Criteria crt = s.~~createCriteria~~(Employee.**class**);

list = crt.list();

**return** list;

}

}

EmployeeService.java

**package** cts.model;

**import** java.util.List;

**public** **class** EmployeeService {

**private** EmployeeDAO dao = **null**;

**public** EmployeeService() {

dao = **new** EmployeeDAO();

}

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

Employee emp = **new** Employee(id, name, salary);

**int** status = dao.store(emp);

**return** status;

}

**public** List displayAll() {

List list = dao.retrieveAll();

//sort logic

**return** list;

}

}

DisplayServlet.java

**package** cts.controller;

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** java.util.List;

**import** javax.servlet.RequestDispatcher;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** cts.model.Employee;

**import** cts.model.EmployeeService;

/\*\*

\* Servlet implementation class DisplayServlet

\*/

**public** **class** DisplayServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

EmployeeService service = **new** EmployeeService();

List list = service.displayAll();

/\*PrintWriter out = response.getWriter();

out.println("<html><body>");

for(Object obj : list) {

Employee emp = (Employee)obj;

out.println("<p>");

out.print("Id = "+emp.getId()+", Name = "+emp.getName()+", Salary = "+emp.getSalary());

out.println("</p>");

out.print("<hr>");

}

out.println("</body></html>");\*/

request.setAttribute("allEmps", list);

RequestDispatcher rd = request.getRequestDispatcher("employees.jsp");

rd.forward(request, response);

}

}

RegisterServlet.java

**package** cts.controller;

**import** java.io.IOException;

**import** javax.servlet.RequestDispatcher;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** cts.model.EmployeeService;

/\*\*

\* Servlet implementation class RegistrationServlet

\*/

**public** **class** RegistrationServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

// 3 parameters from html form i, n and s

**int** id = Integer.*parseInt*(request.getParameter("i"));

String name = request.getParameter("n");

**double** salary = Double.*parseDouble*(request.getParameter("s"));

EmployeeService service = **new** EmployeeService();

**int** status = service.insert(id, name, salary);

**if**(status == 1) {

// Request dispatcher forwards to login.html

RequestDispatcher rd = request.getRequestDispatcher("success.html");

rd.forward(request, response);

} **else** {

// Request dispatcher forwards to failure.html

RequestDispatcher rd = request.getRequestDispatcher("login.html");

rd.forward(request, response);

}

}

}

index.html

<body>

<h2>This is welcome page</h2>

<a href = *"login.html"*>Login</a>

<hr>

<a href = *"register.html"*>Register</a>

<br>

<a href = *"DisplayServlet"*>Display All Employees</a>

</body>

register.html

<body>

<h2>Registration Form!</h2>

<form action = *"RegistrationServlet"* method = *"post"*>

<p>

Enter Id <input type = *"text"* name = *"i"*>

</p>

<p>

Enter Name <input type = *"text"* name = *"n"*>

</p>

<p>

Enter Salary <input type = *"text"* name = *"s"*>

</p>

<p>

<input type = *'submit'* value = *"Register"*>

<input type = *"reset"* value = *"Reset"*>

</p>

</form>

</body>

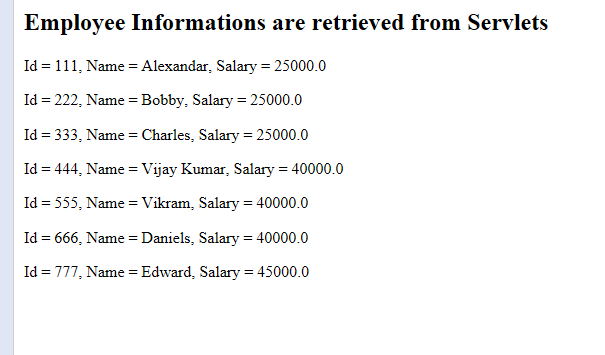
success.html

<body>

You have successfully logged in.

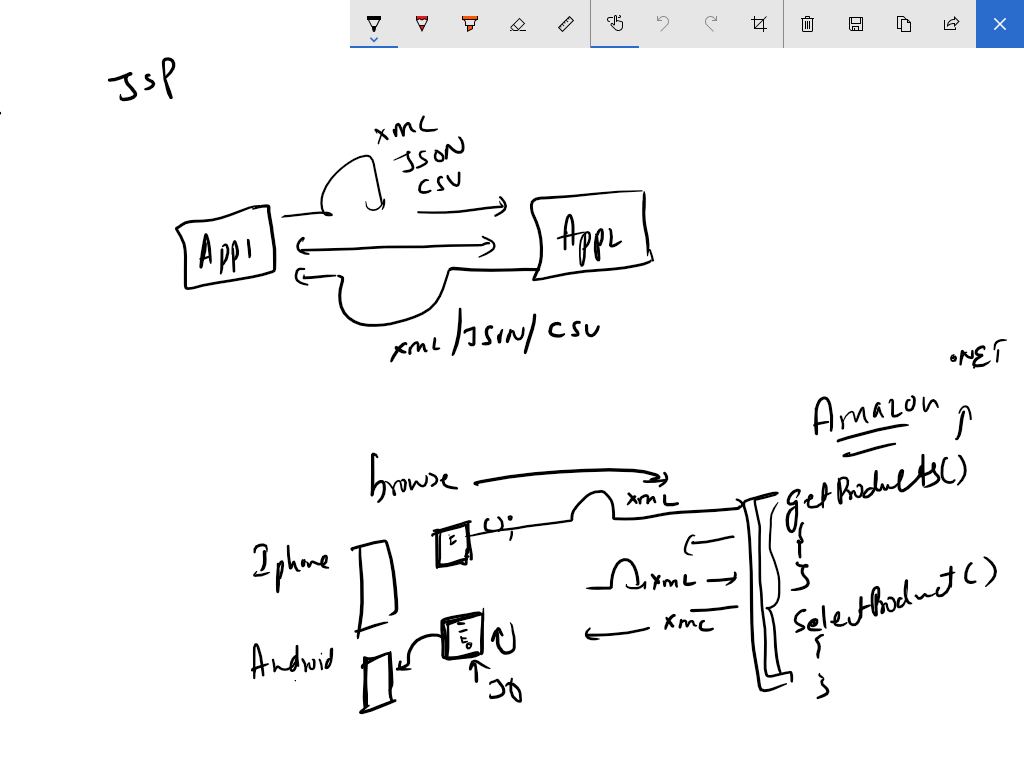
</body>

Output: On Clicking display you get below result



Web Services:

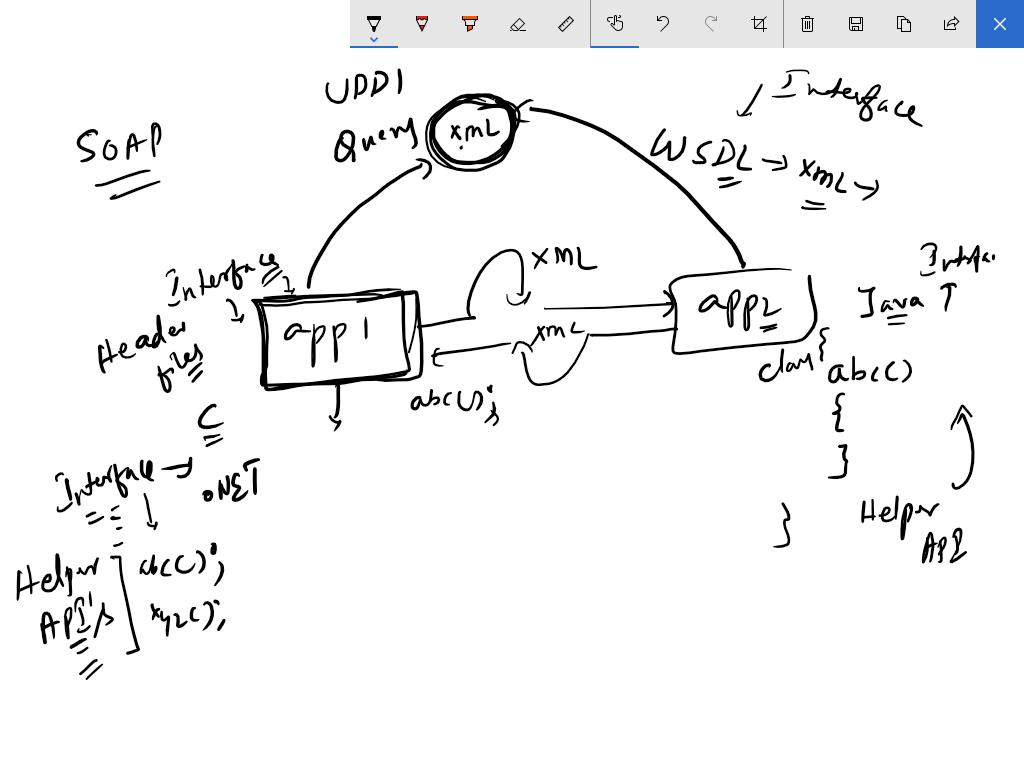
These are logics which can be called from any programming languages, it is for heterogeneous applications used for application integrations.

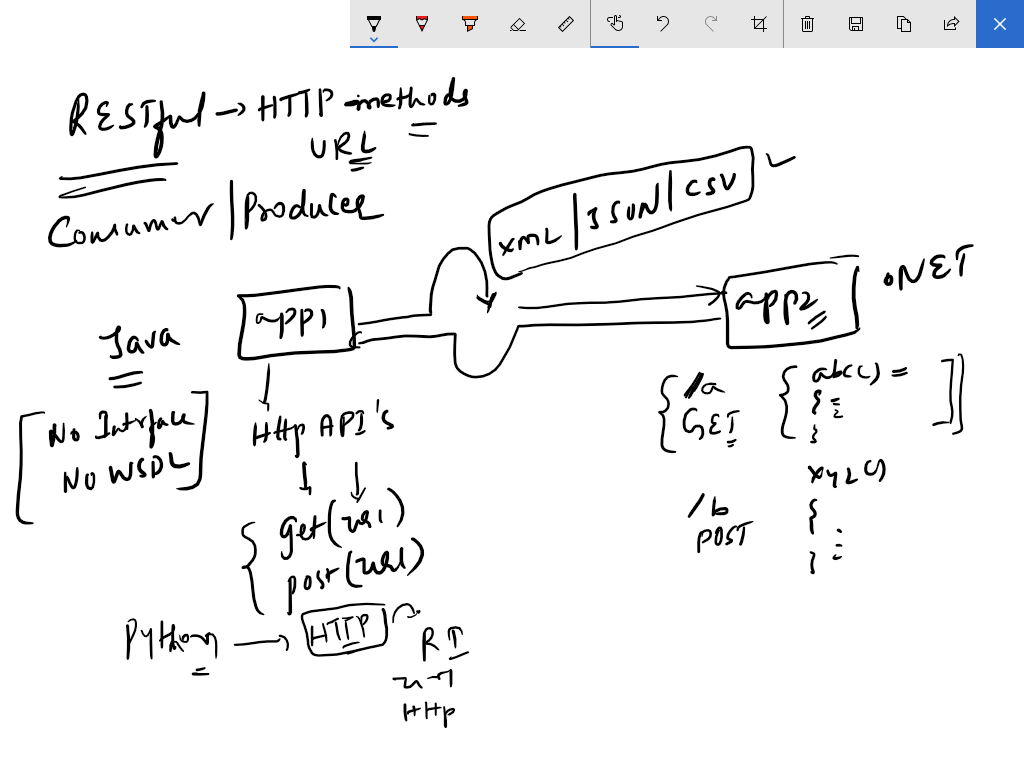


Example: Amazon application already has logics of selectProduct(), getProduct(), checkout() and many more which are written some specific language, these logics are called by different mobile applications written different languages because amazon has exposed their logics as webservices.

There are two types of webservices

1. SOAP (Simple Object Access Protocol)
2. RESTful (Representational State Transfer)





In SOAP webservices the client needs Interface and he must query the WSDL to get the interface the client can understand, moreover the SOAP Webservice rule is that the dataformat must always be in XML format

In RESTful webservices the client need not to have interface and no need of WSDL, the services are exposed with an URL and HTTP methods so the client can use its HTTP API’s to call the services using URL and HTTP methods.

The reason RESTful is termed as Representational State Transfer is because it represents the data/state in XML/JSON/CSV which can be transferred over the internet between the applications.

In Java you can create RESTful API’s using Jersey libraries

jaxrs-ri-2.x.x

Every language programming can implement SOAP and RESTful API’s same way Java also have specified SOAP and RESTful API’s.

Since there are 2 types of Webservice, Java has provided two specifications

1. jax-ws - SOAP webservice
2. jax-rs - RESTful webservice

How to expose the logics you develop in Java as RESTful services

There are some of the annotations provided by jax-rs to implement REST API’s which are:

@Path: it is to configure the URL for classes & methods

@GET, @POST, @PUT, @DELETE: It is to configure the methods so that the client must use appropriate HTTP methods to call, all these annotations are method designators which has to be used on methods.

@Produces: It is a method designator which produces different data formats like XML/JSON/TEXT/HTML and many more so that client could receive such data format.

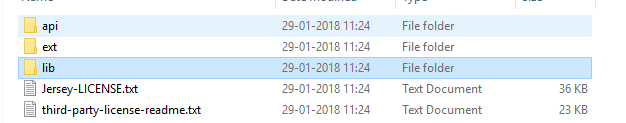
@Consumes: It is a method designator which accepts data formats like XML/JSON/TEXT/HTML and many more from the client

In Java RESTful Webservice uses a FrontController called ServletContainer which will search all the webservices you have created in your application, this FrontController must be configured in a web.xml, most of the times FrontController’s are Servlets

All the annotations are available in Jersey Jars which has to be added to lib folder.

Note: jaxrs-ri-2.25.x is the jar file currently works fine

Once you unzip the jaxrs-ri-2.25.x.zip you would get some folders as below:



Each folders have some Jars which must be copied to lib folder of your dynamic web project.

How to develop RESTful API’s

* Create a Dynamic Web Project >> 2.5
* Add jars to lib folder of your project
* Copy the front controller configuration into web.xml (shown later)
* Create class having webservices in the proper package mentioned in the FrontController.

web.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"* id=*"WebApp\_ID"* version=*"2.5"*>

<display-name>restful-demo</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>A</servlet-name>

<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>

<init-param>

<param-name>jersey.config.server.provider.packages</param-name>

<param-value>com.org</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>A</servlet-name>

<url-pattern>/rest/\*</url-pattern>

</servlet-mapping>

</web-app>

Customer.java >> A JavaBean which represents XML structure also

**package** com.org;

**import** javax.xml.bind.annotation.XmlRootElement;

**import** javax.xml.bind.annotation.XmlType;

@XmlRootElement

@XmlType(propOrder = {"customerId", "customerName", "age"})

**public** **class** Customer {

**private** **int** customerId;

**private** String customerName;

**private** **int** age;

**public** Customer(**int** customerId, String customerName, **int** age) {

**super**();

**this**.customerId = customerId;

**this**.customerName = customerName;

**this**.age = age;

}

**public** Customer() {

**super**();

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

}

**public** **int** getCustomerId() {

**return** customerId;

}

**public** **void** setCustomerId(**int** customerId) {

**this**.customerId = customerId;

}

**public** String getCustomerName() {

**return** customerName;

}

**public** **void** setCustomerName(String customerName) {

**this**.customerName = customerName;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

}

DemoREST.java

**package** com.org;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

@Path("/demo")

**public** **class** DemoREST {

@Path("/simple")

@GET

@Produces(MediaType.***APPLICATION\_XML***)

**public** String test() {

System.***out***.println("test() is called");

**return** "<greeting>Hello RESTFul</greeting>";

}

//employee object wants to be returned to the client

@GET

@Path("/getCustomer")

@Produces(MediaType.***APPLICATION\_XML***)

**public** Customer sendCustomer() {

System.***out***.println("sendCustomer is called");

Customer c = **new** Customer(111, "Rahul", 34);

**return** c;

}

}

In Postman client you can enter the path and choose HTTP methods.

Example:

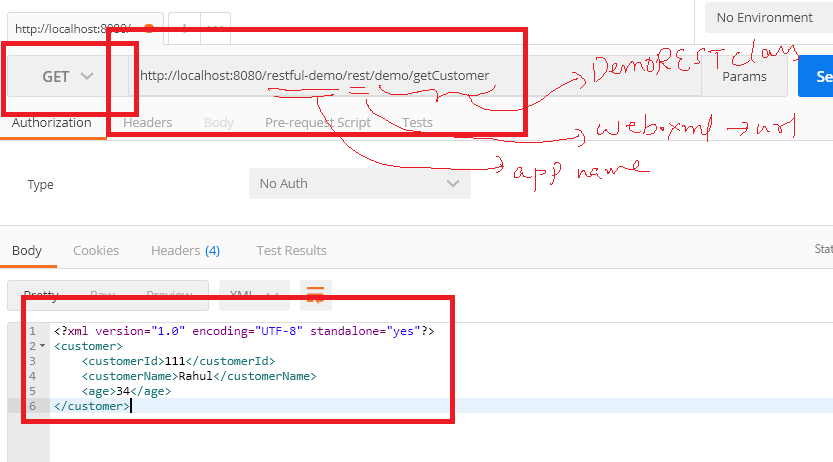
Url for an application name demo-app would be

<http://ip:port/demo-app/path/path/path>

Path are configured in web.xml, and @Path annotation

Example: in web.xml you have used /rest/\* >> so any url that starts with rest is received by front-controller.

Output:



In the URL

* restful-demo is the application name
* rest is configured in web.xml for fron-controller
* demo is configured in DemoREST class as a @Path(“/demo”)
* getCustomer is configured in DemoREST class method called sendCustomer as a @Path(“getCustomer”)

Client can send XML/JSON messages to the Webserivce so that web service can consume that and convert to the appropriate data-structure.

DemoREST.java

**package** com.org;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.POST;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

@Path("/demo")

**public** **class** DemoREST {

@Path("/simple")

@GET

@Produces(MediaType.***APPLICATION\_XML***)

**public** String test() {

System.***out***.println("test() is called");

**return** "<greeting>Hello RESTFul</greeting>";

}

//employee object wants to be returned to the client

@GET

@Path("/getCustomer")

@Produces(MediaType.***APPLICATION\_XML***)

**public** Customer sendCustomer() {

System.***out***.println("sendCustomer is called");

Customer c = **new** Customer(111, "Rahul", 34);

**return** c;

}

@Path("/storeCustomer")

@POST

@Consumes(MediaType.***APPLICATION\_XML***)

@Produces(MediaType.***TEXT\_PLAIN***)

**public** String recieveCustomer(Customer c) {

//call to Service or DAO classes

System.***out***.println("Id = "+c.getCustomerId());

System.***out***.println("Name = "+c.getCustomerName());

System.***out***.println("Age = "+c.getAge());

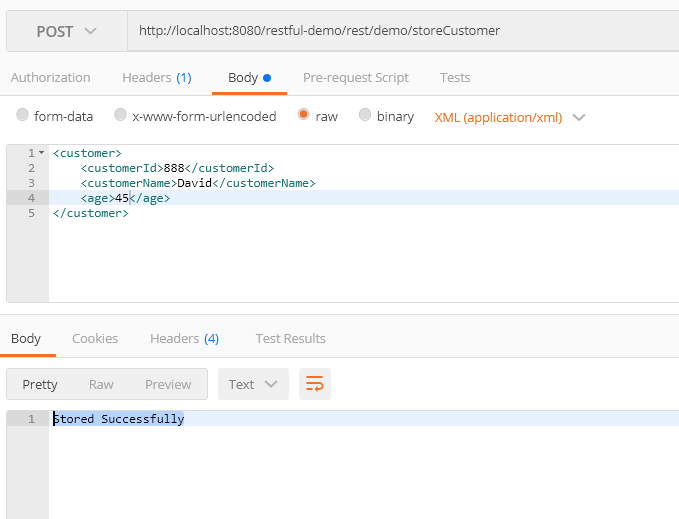
**return** "Stored Successfully";

}

}

The @Consumes(MediaType.APPLICATION\_XML) will convert XML message to Customer object and it will be passed as parameter if XML structure the client sends matches to Customer class XML structure.

Sending XML message from Postman client



After submitting xml message client receives a text message.

Console Output: >> proves that XML message is converted to Customer object

Id = 888

Name = David

Age = 45

Just like XML you can also represent the data in JSON/CSV

XML format for Customer object

<customer>

<customerId>888</customerId>

<customerName>David</customerName>

<age>45</age>

</customer>

JSON format for Customer object

{“customerId”:888, “customerName”:”David”, “age”:45}

CSV format for Customer object

888; David; 45

How to make a service to consume or produce multiple dataformats

public class DemoREST {

…

@Consumes(value = {MediaType.APPLICATION\_XML, MediaType.APPLICATION\_JSON})

@Produces(MediaType.TEXT\_PLAIN)

public String storeCustomer(Customer c) {…}

…

@Produces(value = {MediaType.APPLICATION\_XML, MediaType.APPLICATION\_JSON})

public String recieveCustomer() {…}

…

}

In recieveCustomer() the client would receive XML because of FCFS priority, but client can request for JSON from its request header.

i.e, it can use one of the request header property called Accept:”application/json”

Note: Request and Response will have two sections HEADER and BODY.

Request HEADER will have url, accept, content-type and so on.

Response HEADER will have content-type, cookie, content-length, status and so on

BODY will have content

Using POST, GET, PUT and DELETE request method designators on some methods

POST: It must be used on a method which wants to create a new resource

GET: It must be used on a method which doesn’t create/modify the resource rather it retrieves the data

PUT: It must be used on a method which updates the existing resource

DELETE: It must be used on a method that deletes the resource

Requirement:   
Performing CRUD operations - store and retrieve all

Steps to be performed

* Add hibernate jars to lib folder
* Add mysql-connector.jar to lib folder
* Add hibernate.cfg.xml file
* Create EmployeeService, EmployeeDAO and Employee class
* A web service methods call EmployeeService methods to perform Store and Retrieve.

Employee.java

**package** com.org;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

**import** javax.xml.bind.annotation.XmlRootElement;

@Entity

@XmlRootElement

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

@Id

**private** Integer id;

**private** String name;

**private** Double salary;

**public** Employee(Integer id, String name, Double salary) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

**public** Employee() {

**super**();

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

}

**public** Integer getId() {

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** Double getSalary() {

**return** salary;

}

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

**this**.salary = salary;

}

}

EmployeeDAO.java

**package** com.org;

**import** java.util.List;

**import** org.hibernate.Criteria;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**public** **class** EmployeeDAO {

// store logic

**public** **int** store(Employee emp) {

**int** status = 0;

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Transaction tx = s.beginTransaction();

s.save(emp);

tx.commit();

status = 1;

s.close();

sf.close();

**return** status;

}

// display all

**public** List retrieveAll() {

List list = **null**;

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Criteria crt = s.~~createCriteria~~(Employee.**class**);

list = crt.list();

**return** list;

}

**public** Employee update(**int** id, **double** salary) {

Employee e = **null**;

SessionFactory sf = HibernateUtility.*getSessionFactory*();

Session s = sf.openSession();

Transaction tx = s.beginTransaction();

e = s.get(Employee.**class**, id);

**if**(e != **null**) {

e.setSalary(salary);

s.update(e);

tx.commit();

}

s.close();

sf.close();

**return** e;

}

}

EmployeeService.java

**package** com.org;

**import** java.util.List;

**public** **class** EmployeeService {

**private** EmployeeDAO dao = **null**;

**public** EmployeeService() {

dao = **new** EmployeeDAO();

}

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

Employee emp = **new** Employee(id, name, salary);

**int** status = dao.store(emp);

**return** status;

}

**public** **int** insertObject(Employee emp) {

**int** status = dao.store(emp);

**return** status;

}

**public** List displayAll() {

List list = dao.retrieveAll();

//sort logic

**return** list;

}

**public** Employee updateSalary(**int** id, **double** salary) {

**return** dao.update(id, salary);

}

}

DemoREST.java

**package** com.org;

**import** java.util.List;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.POST;

**import** javax.ws.rs.PUT;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.PathParam;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

@Path("/demo")

**public** **class** DemoREST {

**private** **static** EmployeeService *service* = **new** EmployeeService();

@Path("/simple")

@GET

@Produces(MediaType.***APPLICATION\_XML***)

**public** String test() {

System.***out***.println("test() is called");

**return** "<greeting>Hello RESTFul</greeting>";

}

//employee object wants to be returned to the client

@GET

@Path("/getCustomer")

@Produces(MediaType.***APPLICATION\_XML***)

**public** Customer sendCustomer() {

System.***out***.println("sendCustomer is called");

Customer c = **new** Customer(111, "Rahul", 34);

**return** c;

}

@Path("/storeCustomer")

@POST

@Consumes(MediaType.***APPLICATION\_XML***)

@Produces(MediaType.***TEXT\_PLAIN***)

**public** String recieveCustomer(Customer c) {

//call to Service or DAO classes

System.***out***.println("Id = "+c.getCustomerId());

System.***out***.println("Name = "+c.getCustomerName());

System.***out***.println("Age = "+c.getAge());

**return** "Stored Successfully";

}

@Path("/allEmployees")

@GET

@Produces(MediaType.APPLICATION\_XML)

**public** List<Employee> getAllEmployees() {

return service.displayAll();

}

@Path("/storeEmp")

@POST

@Consumes(MediaType.***APPLICATION\_XML***)

@Produces(MediaType.***TEXT\_PLAIN***)

**public** String storeEmployee(Employee emp) {

**int** count = *service*.insertObject(emp);

**return** count+" Record added";

}

@Path("/update/{i}/{s}") //update/555/87000

@PUT

@Produces(MediaType.***APPLICATION\_XML***)

**public** Employee updateEmployeeSalary(@PathParam("i")**int** id,

@PathParam("s") **double** salary) {

Employee e = *service*.updateSalary(id, salary);

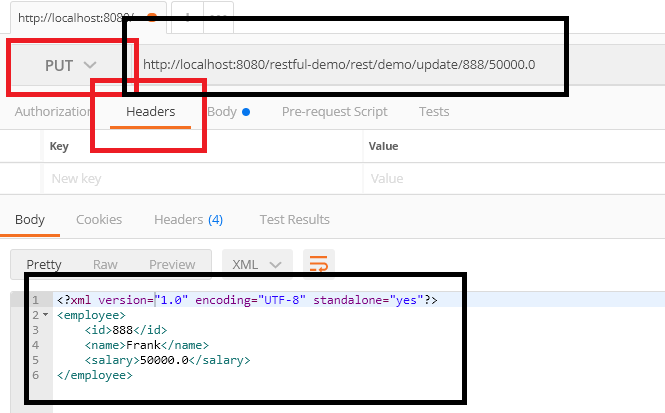
**return** e;

}

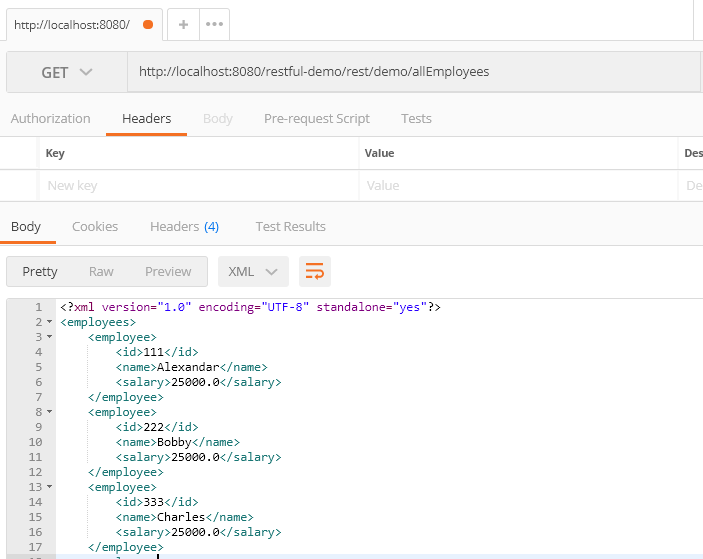
}

Output:

Sending request to update employee information



Getting all the employees



Working on request and response

* Client can send an Accept header and decide what content he needs
* Webservice can also send a Response instance which can have status, content, cookie and many more.

A Response instance can be created using Response class as below

Response.status(200).entity(empObject)

.cookie(cookieObject)

.build();

Some Response status are:

200 >> success

201 >> created

204 >> no content

409 >> conflict

SimpleREST.java

**package** com.org;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

**import** javax.ws.rs.core.Response;

@Path("/simple")

**public** **class** SimpleREST {

@Path("/content")

@GET

@Produces(MediaType.***TEXT\_PLAIN***)

**public** Response getSomeContent1() {

Response response = **null**;

Customer customer = **new** Customer(111, "Rahul", 35);

String content = customer.getCustomerId()+";"+customer.getCustomerName()+";"

+customer.getAge();

response = Response.*status*(200).entity(content).build();

**return** response;

}

@GET

@Path("/content")

@Produces(MediaType.***APPLICATION\_XML***)

**public** Response getSomeContent2() {

Response response = **null**;

Customer customer = **new** Customer(111, "Rahul", 35);

response = Response.*status*(200).entity(customer).build();

**return** response;

}

@GET

@Path("/content")

@Produces(MediaType.***APPLICATION\_JSON***) // {}

**public** Response getSomeContent3() {

Response response = **null**;

Customer customer = **new** Customer(111, "Rahul", 35);

response = Response.*status*(200).entity(customer).build();

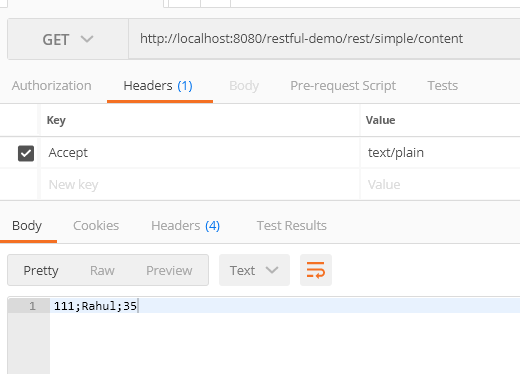
**return** response;

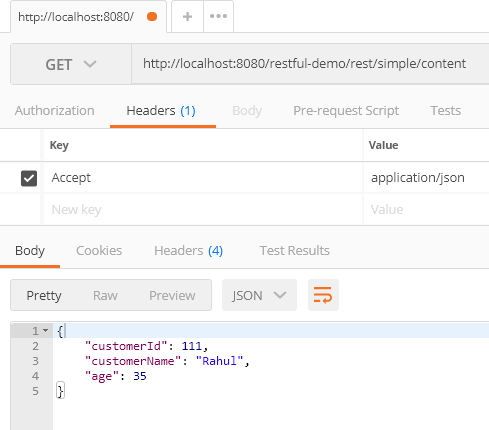
}

}

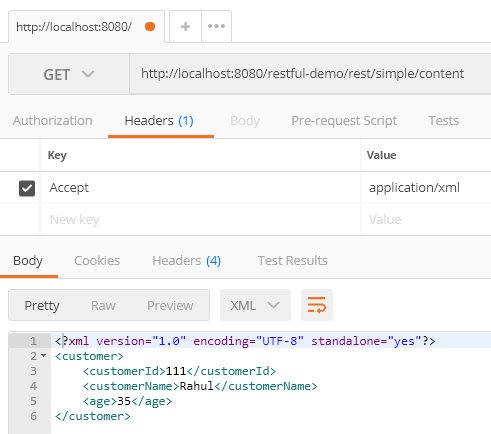
Note: In the above RESTful service three methods take same URL and HTTP methods, but method invocations can be resolved from the Request Accept header.

Output:



If Accept Header is JSON 

If Accept Header is XML



Subresources in RESTful

Resources are informations

In Most cases in the application you will have multiple resources like Customer Resources, Employee Resources, Account resources and etc you can keep all these resources as sub-resource and access them from the main resource.

Example  


Spring Framework

* Spring is an application framework used to develop variety applications like Desktop, Web, Enterprise, Mobile applications and etc.
* Spring framework allows developers to create an application which would be easily testable and maintainable.
* Since every framework follow certain design patterns developers are forced to use these design pattern.
* Frameworks reduces the development work by providing lot of features like Object Creation, Type Conversion, Exception Handling and many more.

/rest/demo/update/111/35000

public Employee update(@PathParam(“i”) int id){}

<customer>

<customerId>111</customerId>

<customerName>Raj</customerName>

</customer>

In the above url the 111 and 35000 was in string format the @PathParam was converting to int type and assigning id variable, same the XML message <customerId> has 111 which is String but RESTful was taking care of creating Customer object with 111 and Raj as Customer information which means a Type Conversion and Object creation happened here.

Spring has many modules like:   
Spring Core

Spring JDBC

Spring MVC

Spring AOP

Spring Boot

Spring Testing

Spring RESTful

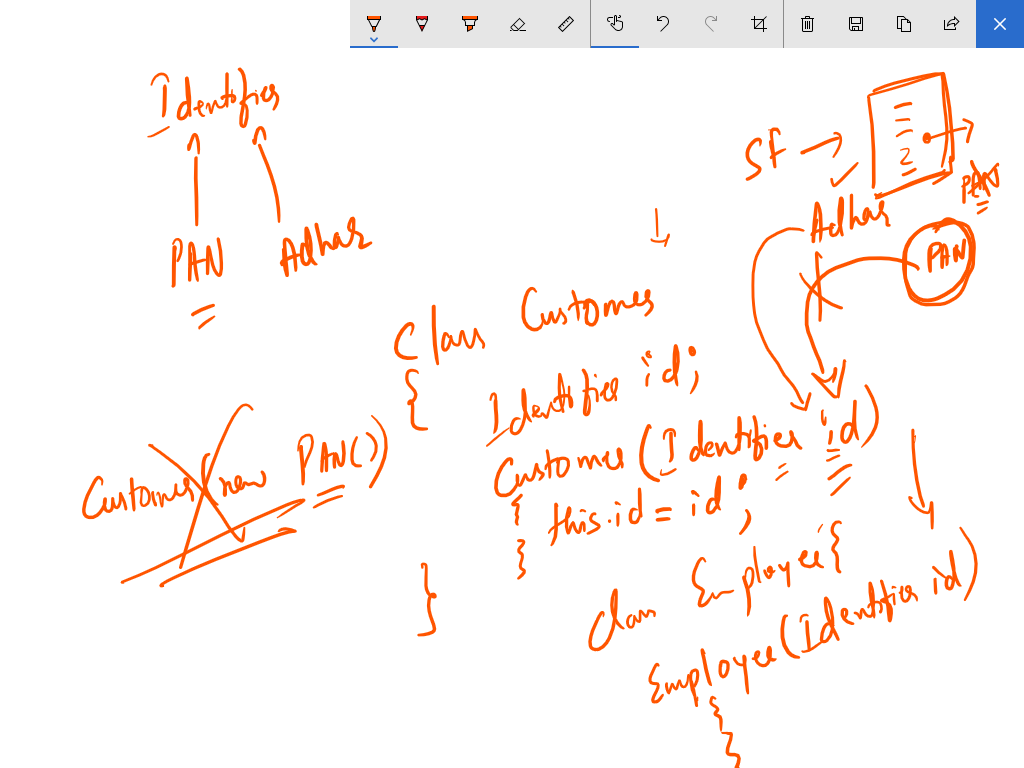
Spring ORM

Spring Core: It is one of the main module in any spring modules which provides a Dependency Injection feature.

What is Dependency Injection?

It supplies the object to the dependent object.

Every object will have some dependencies those dependencies are supplied by framework which is actually called as dependency injection.



Since Customer and Employee are depending on the instance of Identifier you don’t have to instantiate PAN() or Adhar() because it makes complex when the requirement changes as you have to instantiate the object by yourself which is why spring will take care of instantiating the object and supplying the dependency whenever an object needs that dependency, the dependencies will be configured in xml file

How to configure the dependency in spring

* In xml file you must configure the java beans which can be injected/supplied to the dependent object

Usage:

Assume you have a class called PAN then you can configure bean in xml as below:

<beans>

…

<bean id = “idValue” class = “fullyQualifiedClass”>

</bean>

…

</beans>

Once you configured a bean it can be supplied to another object by spring or you can also get the object from the spring container.

How to get the object from the spring container

ApplicationContext ctx = new ClassPathXmlApplicationContext(“xmlFileName.xml”);

// above code references the Spring Container

BeanClass b = (BeanClass)ctx.getBean(“idValue”);

// above code is getting the bean object from the container

You must add some jars in the classpath

commons-logging-1.2.jar

spring-aop.RELEASE.jar

spring-beans.RELEASE.jar

spring-aspects.RELEASE.jar

spring-context.RELEASE.jar

spring-core.RELEASE.jar

spring-jdbc.RELEASE.jar

spring-expression.RELEASE.jar

How to develop a simple spring program to test the bean configurations

beans.xml >> keep inside src

public interface Identifier { void display(); }

public class PAN implements Identifier{ display(){…} }

Now you can configure PAN class inside beans.xml

<beans>

…

<bean id = “b1” class = “pack.PAN”></bean>

…

</beans>

Now you can get or let spring framework to supply it to dependent object

// How to get

ApplicationContext ctx = new ClassPathXmlApplicationContext(“beans.xml”);

Identifer id = (Identifer)ctx.getBean(“b1”);

beans.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>

<bean id = *"b1"* class = *"pack.PAN"*></bean>

</beans>

Identifier.java

**package** pack;

**public** **interface** Identifier {

**void** display();

}

PAN.java

**package** pack;

**public** **class** PAN **implements** Identifier {

@Override

**public** **void** display() {

System.***out***.println("\*\* Displays PAN Information\*\*\*");

}

}

Aadhar.java

**package** pack;

**public** **class** Aadhar **implements** Identifier {

@Override

**public** **void** display() {

System.***out***.println("\*\*\*Displays Aadhar Information\*\*\*");

}

}

TestSprings.java

**package** pack;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** TestSprings {

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

ApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

Identifier id = (Identifier)context.getBean("b1");

id.display();

}

}

Output:

INFO: Loading XML bean definitions from class path resource [beans.xml]

\*\* Displays PAN Information\*\*\*

Modify beans.xml replacing PAN by Aadhar you get below output

INFO: Loading XML bean definitions from class path resource [beans.xml]

\*\*\*Displays Aadhar Information\*\*\*

How to configure beans so that it will be injected to another object

It can be done in 2 ways

* setter injection
* constructor injection

These are setter methods and constructors which are used to supply the object.

Customer has Identifer property then Customer is depending on Identifer object which the spring can supply either through setter method or constructor.

public class Customer {

private Identifer id;

public Customer(Identifier id){ this.id = id; }

public void setId(Identifier id){ this.id = id; }  
}

How to configure this in xml file

Spring Framework can supply the Identifier object using setter injection

<bean id = “b1” class = “pack.Aadhar”></bean>

<bean id = “c1” class = “pack.Customer”>

<!-- supplying through setter using property tag -->

<property name = “id” ref = “b1”>

</bean>

Spring framework can also supply the object using constructor injection

<bean id = “b1” class = “pack.Aadhar”></bean>

<bean id = “c1” class = “pack.Customer”>

<constructor-arg index = “0” ref = “b1”></bean>

</bean>

Here the constructor first parameter takes <bean> b1 instance as a parameter.

Note: When you do setter injection the framework instantiates the object calling default constructor but if you use constructor injection then framework instantiates by calling parameterized constructor.

beans.xml  
<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>

<bean id = *"b1"* class = *"pack.PAN"*></bean>

<bean id = *"c1"* class = *"pack.Customer"*>

<!-- <property name = "id" ref = "b1"></property> -->

<constructor-arg index = *"0"* ref = *"b1"*></constructor-arg>

</bean>

</beans>

Customer.java

**package** pack;

**public** **class** Customer {

**private** Identifier id;

**public** Customer() {

**super**();

System.***out***.println("Customer() is called");

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

}

**public** Customer(Identifier id) {

**super**();

System.***out***.println("Customer(Identifier) is called");

**this**.id = id;

}

**public** Identifier getId() {

**return** id;

}

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

System.***out***.println("setId() is called");

**this**.id = id;

}

**public** **void** getInformations() {

id.display();

}

}

TestSprings.java

**package** pack;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** TestSprings {

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

ApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

/\*Identifier id = (Identifier)context.getBean("b1");

id.display();\*/

Customer c = (Customer)context.getBean("c1");

c.getInformations();

}

}

Output:

INFO: Loading XML bean definitions from class path resource [beans.xml]

Customer(Identifier) is called

\*\* Displays PAN Information\*\*\*

Collection injection: You can supply collection to a java bean when a bean is depending on collection of data.

For example a Bank class may have locations which specifies all the locations the bank branch is existing, instead of having collection of locations specified in a Bank class you can specify in xml file so that any new locations can be added in the xml file will be reflected to the Bank class.

public class Bank {

private String bankName;

private List<String> locations;

setters & getters or argument constructor

}

In xml file you can configure as below:

<bean id = “bank” class = “pack.Bank”>

<property name = “bankName” value = “ICICI”>

<property name = “locations”>

<list>

<value>Bangalore</value>

<value>Chennai</value>

</list>

</property>

</bean>

Note: <list> and <value> tags can be written in <constructor-arg> also if you want to constructor injection.

In Collection you can use <set> tag also if you want only unique values, but if you use <list> tag it can maintain duplicates

beans.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>

<bean id = *"b1"* class = *"pack.PAN"*></bean>

<bean id = *"c1"* class = *"pack.Customer"*>

<!-- <property name = "id" ref = "b1"></property> -->

<constructor-arg index = *"0"* ref = *"b1"*></constructor-arg>

</bean>

<bean id = *"bank"* class = *"pack.Bank"*>

<property name = *"bankName"* value = *"ICICI"*></property>

<property name = *"locations"*>

<list>

<value>Bangalore</value>

<value>Chennai</value>

<value>New Delhi</value>

</list>

</property>

</bean>

</beans>

Bank.java

**package** pack;

**import** java.util.List;

**public** **class** Bank {

**private** String bankName;

**private** List<String> locations;

**public** Bank() {

**super**();

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

}

**public** Bank(String bankName, List<String> locations) {

**super**();

**this**.bankName = bankName;

**this**.locations = locations;

}

**public** String getBankName() {

**return** bankName;

}

**public** **void** setBankName(String bankName) {

**this**.bankName = bankName;

}

**public** List<String> getLocations() {

**return** locations;

}

**public** **void** setLocations(List<String> locations) {

**this**.locations = locations;

}

}

TestSprings.java

**package** pack;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** TestSprings {

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

ApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

/\*Identifier id = (Identifier)context.getBean("b1");

id.display();\*/

/\*

Customer c = (Customer)context.getBean("c1");

c.getInformations();\*/

Bank bank1 = (Bank) context.getBean("bank");

System.***out***.println(bank1.getLocations());

}

}

Output:

[Bangalore, Chennai, New Delhi]

You can use an annotation called @Autowired instead of <property ref> if you want a bean to get the dependency injected, this a new approach introduced in Spring 3.x earlier the dependency injection was happening through <property ref> or <constructor-arg ref>, but they are still used because @Autowired is not better everytime, example a class needs some list of values then @Autowired cannot configure list of values we need to go for xml configurations only.

@Component

public class Address{}

public class Customer{

@Autowired

private Address address;

}

Since Address is not configured in XML file the framework needs those classes which will have @Component such that it can be instantiated.

The framework will scan those classes having @Component using a configuration in xml file.

<component-scan base-package = “pack”>

beans.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>

<context:component-scan base-package=*"pack"*></context:component-scan>

<bean id = *"b1"* class = *"pack.PAN"*></bean>

<bean id = *"c1"* class = *"pack.Customer"*>

<!-- <property name = "id" ref = "b1"></property> -->

<constructor-arg index = *"0"* ref = *"b1"*></constructor-arg>

</bean>

<bean id = *"bank"* class = *"pack.Bank"*>

<property name = *"bankName"* value = *"ICICI"*></property>

<property name = *"locations"*>

<list>

<value>Bangalore</value>

<value>Chennai</value>

<value>New Delhi</value>

</list>

</property>

</bean>

</beans>

Address.java

**package** pack;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** Address {

**public** Address() {

System.***out***.println("Address() is created");

}

**public** **void** display() {

System.***out***.println("display() of Address is called");

}

}

Customer.java

**package** pack;

**import** org.springframework.beans.factory.annotation.Autowired;

**public** **class** Customer {

**private** Identifier id;

@Autowired

**private** Address address;

**public** Customer() {

**super**();

System.***out***.println("Customer() is called");

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

}

**public** Customer(Identifier id) {

**super**();

System.***out***.println("Customer(Identifier) is called");

**this**.id = id;

}

**public** Identifier getId() {

**return** id;

}

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

System.***out***.println("setId() is called");

**this**.id = id;

}

**public** **void** getInformations() {

id.display();

address.display();

}

**public** Address getAddress() {

**return** address;

}

**public** **void** setAddress(Address address) {

**this**.address = address;

}

}

TestSprings.java

**package** pack;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** TestSprings {

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

ApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

/\*Identifier id = (Identifier)context.getBean("b1");

id.display();\*/

/\*

Customer c = (Customer)context.getBean("c1");

c.getInformations();\*/

/\*Bank bank1 = (Bank) context.getBean("bank");

System.out.println(bank1.getLocations());\*/

Customer c = (Customer)context.getBean("c1");

c.getInformations();

}

}