Build an InternsManagement System Using JSE

This Lab provides a step-by-step guide for building InternsManagement System using Java Standard Edition

Prerequisites:

- Spring Tool Suite 3/4
- Java 1.8
- Basic understanding of design patterns

The Spring Tool Suite is an open-source, Eclipse-based IDE distribution that provides a superset of the Java EE distribution of Eclipse

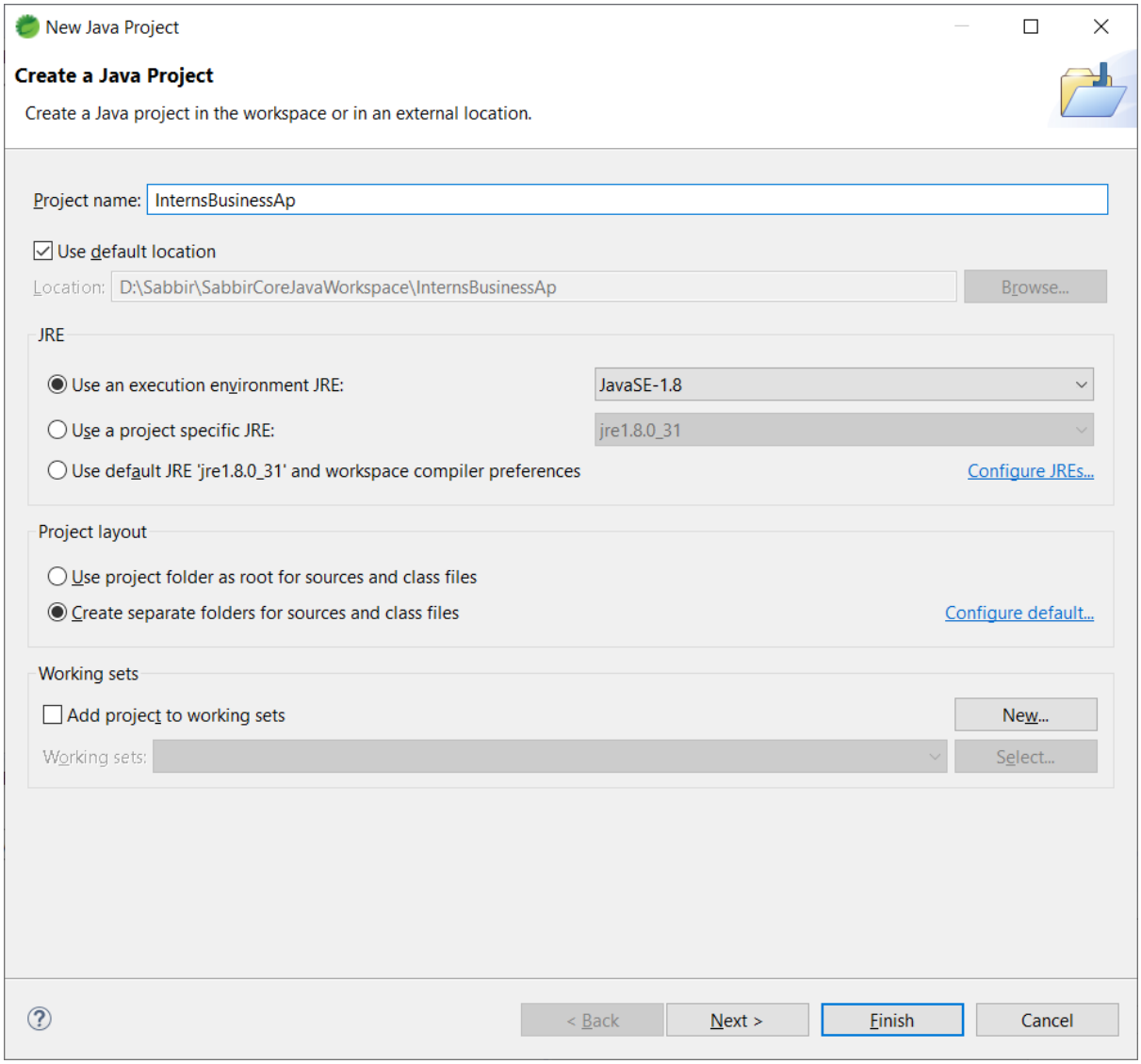
Business Scenario:

An application has to be developed for management of interns hired by Yash technologies. The application will provide following services:

1. Based on technical evaluation, intern will be hired and details will be entered in system. Intern's technical level will be decided based on college semester marks. Interns will have to go through training based on levels and clear certifications.
2. Intern's details can be retrieved by HR to assign projects to intern.
3. Intern's personal details can be updated.
4. Intern's level can be updated based on semester scores.
5. Intern's details will be removed from system if intern has completed internship.

Let’s start creating application. We will create application using MVC design pattern.

Create a java project InternsBusinessApp in Eclipse STS,



Create below packages,

Package Name	Purpose
com.yash.entity	Domain class representing real world entity or physical entity to create domain objects
com.yash.controller	Controller class for controlling flow of application
com.yash.service	Business service classes to implement business logic
com.yash.dao	Classes performing data operations
com.yash.view	Classes for asking input or giving output to end-user
com.yash.exception	User-defined exception handling classes

com.yashhelper	Classes performing low-level services
com.yash.validation	Classes to perform validation of user-inputs
com.yash.test	Test cases
com.yash.ui	User interface Classes
com.yash.model	Model Classes

Create an enum in com.yash.entity package with name Enum to specify constants as level of Intern.

Notice the enum keyword which is used in place of class or interface. The Java enum keyword signals to the Java compiler that this type definition is an enum.

You can refer to the constants in the enum above like this:

```
Level level = Level.BEGINNNER;
```

Notice how the level variable is of the type Level which is the Java enum type defined below. The level variable can take one of the Level enum constants as value (BEGINNER, INTERMEDIATE or ADVANCED). In this case level is set to BEGINNER.

```
package com.yash.entity;
public enum Level {
    BEGINNER(0),INTERMEDIATE(1),ADVANCED(2);
    private int level;
    private Level(int level) {
        this.level=level;
    }
    public int getLevel() {
        return level;
    }
}
```

Java enums extend the java.lang.Enum class implicitly, so your enum types cannot extend another class.

If a Java enum contains fields and methods, the definition of fields and methods must always come *after* the list of constants in the enum.

Additionally, the list of enum constants must be terminated by a semicolon;

You should use enumerated types any time you need to represent a fixed set of constants. That includes natural enumerated types such as the planets in our solar system, the days of the week, and the suits in a deck of cards as well as sets where you know all possible values at compile time, for example the choices on a menu, rounding modes, command line flags, and so on.

Java programming language enumerated types are much more powerful than their counterparts in other languages, which are just glorified integers. The enum declaration defines a *class* (called an *enum type*). These are the most important properties of enum types:

- Printed values are informative.
- They are typesafe.
- They exist in their own namespace.
- The set of constants is not required to stay fixed for all time.
- You can switch on an enumeration constant.
- They have a static values method that returns an array containing all of the values of the enum type in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enumerated type.
- You can provide methods and fields, implement interfaces, and more.
- They provide implementations of all the Object methods. They are Comparable and Serializable, and the serial form is designed to withstand changes in the enum type.

Create a domain or business entity class Interns in com.yash.entity package.

```
package com.yash.entity;
public class Interns {
     public Interns() {}
     private int id;
     private String internFirstName;
     private String internLastName;
     private int internAge;
    private Level level;
     public int getId() {
          return id;
     }
     public void setId(int id) {
          this.id = id;
     }
     public String getInternFirstName() {
          return internFirstName;
     }
     public void setInternFirstName(String
internFirstName) {
          this.internFirstName = internFirstName;
     }
     public String getInternLastName() {
          return internLastName;
     }
     public void setInternLastName(String internLastName)
{
          this.internLastName = internLastName;
     }
     public int getInternAge() {
          return internAge;
     }
     public void setInternAge(int internAge) {
          this.internAge = internAge;
     }
     public Level getLevel() {
          return level;
     }
     public void setLevel(Level level) {
          this.level = level;
```

```
    }

}
```

In **Java**, **getter and setter** are two conventional methods that are used for retrieving and updating value of a variable. So, a **setter** is a **method** that updates value of a variable. And a **getter** is a **method** that reads value of a variable. **Getter and setter** are also known as accessor and mutator in **Java**.

The class declares a private variable,id. Since id is private, code from outside this class cannot access the variable directly, like this:

```
1  Interns interns = new Interns();
2  interns.id = 10;    // compile error, since number is private
3  int id_no =interns.id; // same as above
```

Instead, the outside code have to invoke the getter, getId() and the setter, setId() in order to read or update the variable, for example:

```
1  Interns interns = new Interns();
2  interns.setId(1001);  // OK
4  int id_no = interns.getId();  // fine
```

So, a setter is a method that updates value of a variable. And a getter is a method that reads value of a variable.

Why getter and setter?

By using getter and setter, the programmer can control how his important variables are accessed and updated in a correct manner, such as changing value of a variable within a specified range.

Consider the following code of a setter method:

```
1  public void setId(int id) {
2      if (id < 0 || id > 100000) {
3          throw new IllegalArgumentException();
3      }
4      this.id = id;
}
```

That ensures the value of id is always set between 1 and 100000. Suppose the variable **id** can be updated directly, the caller can set any arbitrary value to it:

```
1    intern.id =-3;
```

And that violates the constraint for values ranging from 1 to 100000 for that variable. Of course we don't expect that happens. Thus hiding the variable id as private and using a setter comes to rescue.

On the other hand, a getter method is the only way for the outside world reads the variable's value:

```
1    public int getId() {
2        return this.id;
3    }
```

So far, setter and getter methods protect a variable's value from unexpected changes by outside world - the caller code.

When a variable is hidden by *private* modifier and can be accessed only through getter and setter, it is *encapsulated*.

Encapsulation is one of the fundamental principles in object-oriented programming (OOP), thus implementing getter and setter is one of ways to enforce encapsulation in program's code.

Naming convention for getter and setter

The naming scheme of setter and getter should follow *Java bean naming convention* as follows:

getXXX() and setXXX()

where XXX is name of the variable. For example with the following variable name:

```
1    private String internFirstName;
```

then the appropriate setter and getter will be:

```
1    public void setInternFirstName(String internFirstName) { }
2
```

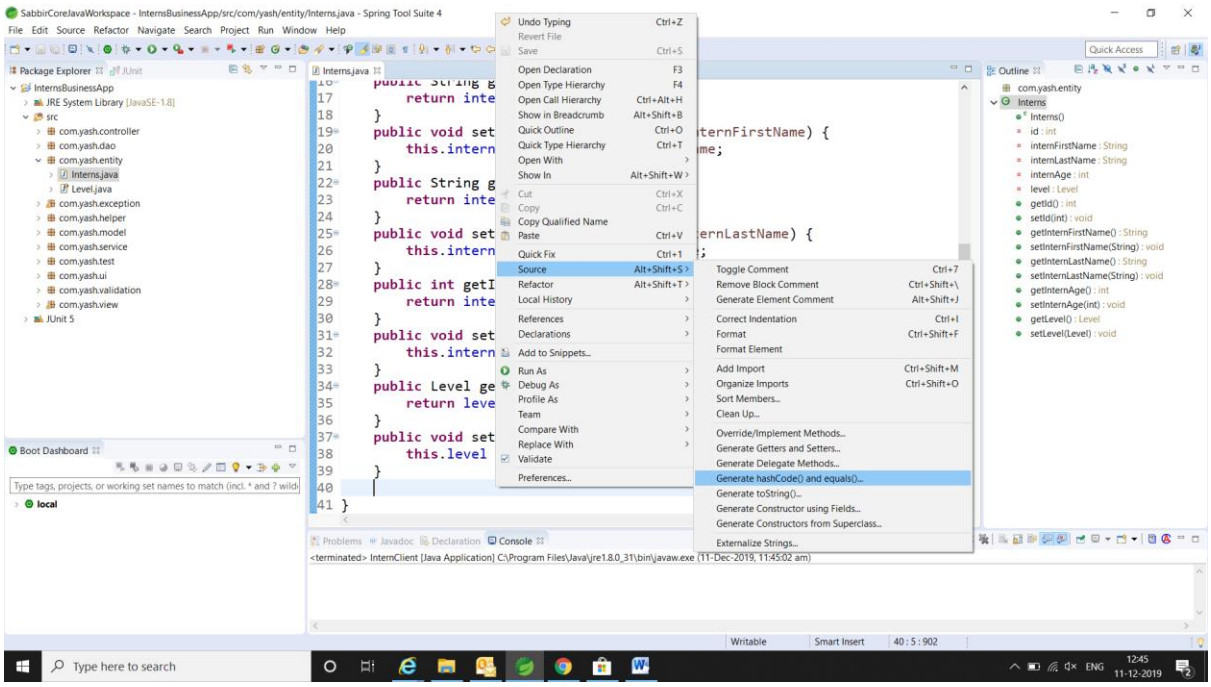
```
3    public String getinternFirstName(){}
```

If the variable is of type boolean, then the getter's name can be either isXXX() or getXXX(), but the former naming is preferred.

For example:

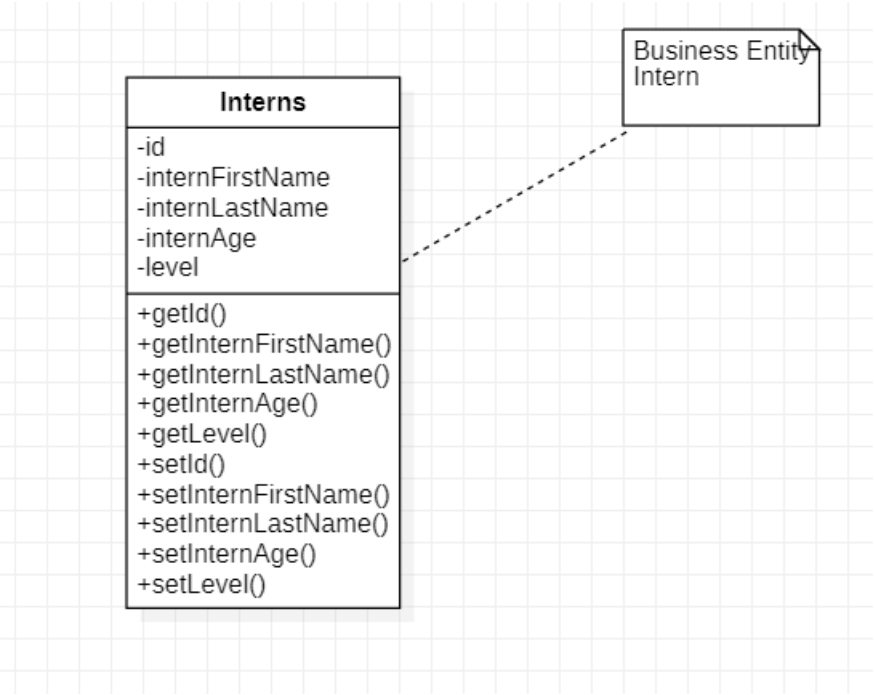
```
1    private boolean eligible;
2
3    public String isEligible() { }
```

Right click in java editor for source file Interns.java and Override toString(),equals() and hashCode().



You must **override hashCode** in every class that **overrides equals**. Failure **to do** so will result in a violation of the general contract for Object.**hashCode**, which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.

If you print any object, java compiler internally invokes the **toString()** method on the object. So **overriding** the **toString()** method, returns the desired output, it can be the state of an object etc. depends on your implementation.



Complete listing of **Interns.java** is,

```
package com.yash.entity;
public class Interns {
     public Interns(){}
     private int id;
     private String internFirstName;
     private String internLastName;
     private int internAge;
    private Level level;

     public int getId() {
          return id;
     }
     public void setId(int id) {
          this.id = id;
     }
     public String getInternFirstName() {
          return internFirstName;
     }
     public void setInternFirstName(String
internFirstName) {
          this.internFirstName = internFirstName;
     }
```

```
    public String getInternLastName() {
        return internLastName;
    }
    public void setInternLastName(String internLastName)
{
        this.internLastName = internLastName;
    }
    public int getInternAge() {
        return internAge;
    }
    public void setInternAge(int internAge) {
        this.internAge = internAge;
    }
    public Level getLevel() {
        return level;
    }
    public void setLevel(Level level) {
        this.level = level;
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + internAge;
        result = prime * result + ((internFirstName ==
null) ? 0 : internFirstName.hashCode());
        result = prime * result + ((internLastName ==
null) ? 0 : internLastName.hashCode());
        result = prime * result + ((level == null) ? 0 :
level.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Interns other = (Interns) obj;
```

```
            if (id != other.id)
                return false;
            if (internAge != other.internAge)
                return false;
            if (internFirstName == null) {
                if (other.internFirstName != null)
                    return false;
            } else if
(!internFirstName.equals(other.internFirstName))
                return false;
            if (internLastName == null) {
                if (other.internLastName != null)
                    return false;
            } else if
(!internLastName.equals(other.internLastName))
                return false;
            if (level != other.level)
                return false;
            return true;
        }
        @Override
        public String toString() {
            return "Interns [id=" + id + ", internFirstName="
+ internFirstName + ", internLastName=" + internLastName
                    + ", internAge=" + internAge + ",
level=" + level + "]";
        }

}
```

What Is an Interface?

In general, an interface is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people, and the protocol of behaviour enforced in the military is the interface between people of different ranks.

Within the Java programming language, an ***interface***.is a type, just as a class is a type. Like a class, an interface defines methods. Unlike a

class, an interface never implements methods; instead, classes that implement the interface implement the methods defined by the interface. A class can implement multiple interfaces.

Three main utilisation of interfaces,

- To hide implementation details
- To relate unrelated classes
- To achieve multiple inheritance

An **abstract class** allows you to create functionality that subclasses can implement or override. An **interface** only allows you to define functionality, not implement it. And whereas a **class** can extend only one **abstract class**, it can take advantage of multiple **interfaces.**

When to choose interfaces over abstract class ?

Abstract class and interface are nothing but set of rules.

When a class need to implement only one set of rule go for abstract class whereas if a class need to implement more than one set of rules go for interfaces.

Simple example to understand above statements.

First scenario:

In Yash technologies, there is one team which is responsible for charting out all business policies of Yash. Each employee in Yash technologies must adhere to these business policies. All business policies are mentioned in one set.

```
abstract class YashRules{

    abstract void hr_rule();
    abstract void account_rule();
    abstract void travel_rule();
}
```

```
class Employee extends YashRules{
    @Override
    void hr_rule() {
        // TODO Auto-generated method stub
    }
    @Override
    void account_rule() {
        // TODO Auto-generated method stub
    }
    @Override
    void travel_rule() {
        // TODO Auto-generated method stub
    }
}
```

Second scenario:

In Yash technologies, there are different departments HR, Finance, Logistics, Travel etc

Each department has a team which chart out business policies relevant to that department. Employee must adhere to policies of each department.

```
interface HR_Rules{

    abstract void rule_1();
    abstract void rule_2();
    abstract void rule_n();
}
```

```
interface Finance_Rules{

    abstract void rule_1();
    abstract void rule_2();
    abstract void rule_n();
}
```

```
class Employee implements HR_Rules,Finance_Rules{
    @Override
    public void rule_1() {
        // TODO Auto-generated method stub
    }
    @Override
    public void rule_2() {
        // TODO Auto-generated method stub
    }

    @Override
    public void rule_n() {
        // TODO Auto-generated method stub
    }
}
```

Another example,

Say we want to create a multithreaded applet. As per Applet specification a class must extend super class java.Applet to be considered as an applet by runtime environment. To provide implementation when thread is running we should override run() from java.lang.Thread class.

Now in this scenario either we can extend Applet or Thread but not both as java do not support multiple inheritance. So we can extend Applet to conform to specification and implement interface Runnable which has life cycle method of thread.

Java 8 extended interface types to also support **default** and **static** method declarations. Consequently, you can declare the `main()` entry-point method as an interface member, which leads to creating interface-based application types.

```
package com.yash.demo;
public interface HelloInterface {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

On running above interface, output on console **“Hello World!”**

Don't expect an interface-based application type to be fully equivalent to a class-based application type. For example, unlike a class, you cannot instantiate an interface. As a result, you cannot declare non-`static` fields in an interface. Also regarding fields, you can declare only `static` constants in an interface: interfaces don't support `static` fields with mutable state.

The **default methods** were introduced to provide backward compatibility so that existing intefaces can **use** the lambda expressions without implementing the **methods** in the implementation class. **Default methods** are also known as defender **methods** or virtual extension **methods**

```
package com.yash.demo;
public interface YashInterface {
    public default void yashMethod() {
        System.out.println("--yashMethod default
implementation--");
    }
}
```

Since one interface can be implemented by more than one class, if you want to provide implementation common to all classes put it in default method instead of duplicating in all classes.

Good example to understand this is List interface. List interface is implemented by ArrayList as well as LinkedList.

Iteration of ArrayList and LinkedList is same. Start from 0 index position till the size.

So a default method is introduced in List interface of Java 8, forEach().

class implementing interface may choose to provide different implementation then default method.

```
package com.yash.demo;
public class YashInterfaceImpl implements YashInterface{
    @Override
    public void yashMethod() {
        System.out.println("--YashInterfaceImpl version
of implementation--");
    }
}
```

To invoke explicitly default method from implementation class,

```
package com.yash.demo;
public class YashInterfaceImpl implements YashInterface{
    @Override
    public void yashMethod() {
   YashInterface.super.yashMethod();
    }
}
```

Notice the super keyword is used on reference of interface.

Logical reasoning is one class may implement one than one interface.
So reference of interface and then super and method.
Whereas in case of classes, Since Java has Single inheritance Strategy, one class must have one super class

So `super.method();`

In Java 8, a new feature introduced is Functional interfaces.

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. **Runnable**, **ActionListener**, **Comparable** are some of the examples of functional interfaces.

Functional interfaces are annotated by **@FunctionalInterface** annotation.

```java
package com.yash.demo;
@FunctionalInterface
public interface YashFunctionalInterface {
    public void yashMethod();
}
```

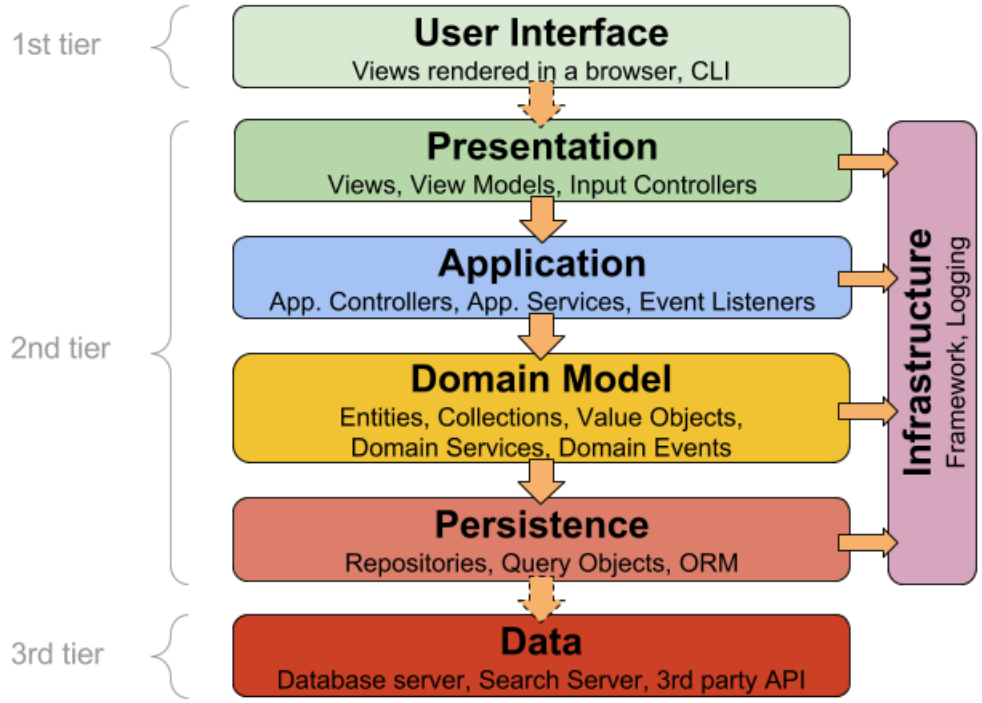
To provide implementation of method declared in functional interface we can use lambda expression

```java
YashFunctionalInterface yash=()->{
              System.out.println("Welcome to Yash!");
              };
```

We will utilise in-built functional interfaces in java.util.function in our application. In built functional interfaces are Consumer, Supplier, Function, Predicate etc.

We will create application following layered architecture.

Layered architecture allows to swap and reuse components at will.
Layered architecture enables teams to work on different parts of the **application** parallelly with minimal dependencies on other teams.
Layered architecture enables develop loosely coupled systems.



We will segregate application in three layers- UI(CLI), Presentation layer(Controller, View, Model, Validator)), Business Layer(Business Services classes) and Persistence Layer(DAO's).

Benefit of this approach will be say for example in our application user interface changes from Console based to web based; we can reuse business layer and persistence layer.

Each layer will be created as separate projects and will be in different deployable formats and files. If one of layer changes it do not impact other layers.

For simulation we will consider packages as layer. For example com.yash.dao as data layer, com.yash.service as business layer and com.yash.view and com.yash.controller belongs to UI layer.

Persistence Layer

The reason for you to build a persistence layer or any other kind of intermediate layer between database engine and Business / Application logic, is that by adding this layer in the between you isolate the rest / upper layers of your application from the specific database engine / technology you are using right now.

This has several advantages, like easier migration to other storage engines, better encapsulation of database logic in a single layer (easier to replace or modify later depending on how well you have designed your cross-layer interfaces etc...)

In initial release we will provide memory implementation. We will utilise collection framework for performing CRUD operation for business entity Interns.

We will use Map interface from Collection framework.

The java.util.Map interface represents a mapping between a key and a value and implementation class is java.util.HashMap.

Working of HashMap

Internal storage

The JAVA HashMap class implements the interface Map<K,V>. The main methods of this interface are:

- V put(K key, V value)
- V get(Object key)
- V remove(Object key)
- Boolean containsKey(Object key)

HashMaps use an inner class to store data: the Entry<K, V>. This entry is a simple key-value pair with two extra data:

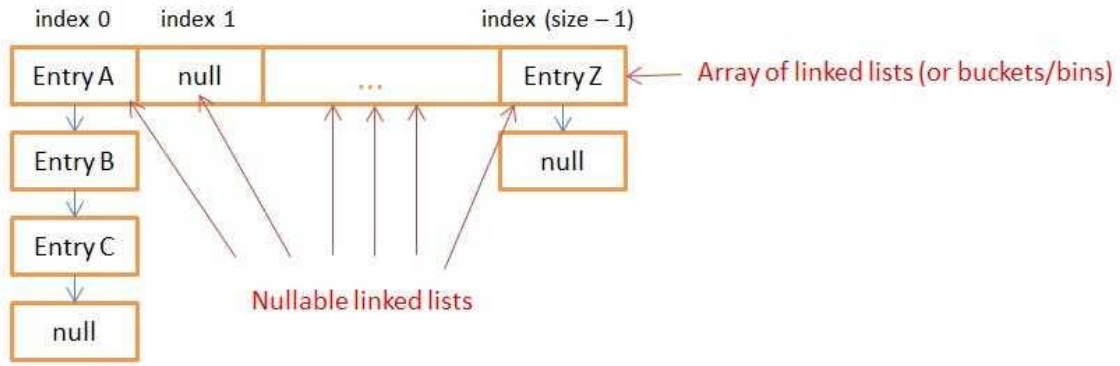
- a reference to another Entry so that a HashMap can store entries like singly linked lists

- a hash value that represents the hash value of the key. This hash value is stored to avoid the computation of the hash every time the HashMap needs it.

Here is a part of the Entry implementation in JAVA 7

```
static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        Entry<K,V> next;
        int hash;
        //remaining code
}
```

A HashMap stores data into multiple singly linked lists of entries (also called **buckets** or **bins**). All the lists are registered in an array of Entry (Entry<K,V>[] array) and the default capacity of this inner array is 16.



The following picture shows the inner storage of a HashMap instance with an array of nullable entries. Each Entry can link to another Entry to form a linked list.

All the keys with the same hash value are put in the same linked list (bucket). Keys with different hash values can end-up in the same bucket.

When a user calls put(K key, V value) or get(Object key), the function computes the index of the bucket in which the Entry should be. Then, the function iterates through the list to look for the Entry that has the same key (using the equals () function of the key).

In the case of the get(), the function returns the value associated with the entry (if the entry exists).

In the case of the put (K key, V value), if the entry exists the function replaces it with the new value otherwise it creates a new entry (from the key and value in arguments) at the head of the singly linked list.

This index of the bucket (linked list) is generated in 3 steps by the map:

- It first gets the **hashcode** of the key.
- It **rehashes** the hashcode to prevent against a bad hashing function from the key that would put all data in the same index (bucket) of the inner array
- It takes the rehashed hash hashcode and **bit-masks** it with the length (minus 1) of the array. This operation assures that the index can't be greater than the size of the array.

JAVA 8 improvements

The inner representation of the HashMap has changed a lot in JAVA 8. In JAVA 8, you still have an array but it now stores Nodes that contains the exact same information as Entries and therefore are also linked lists:

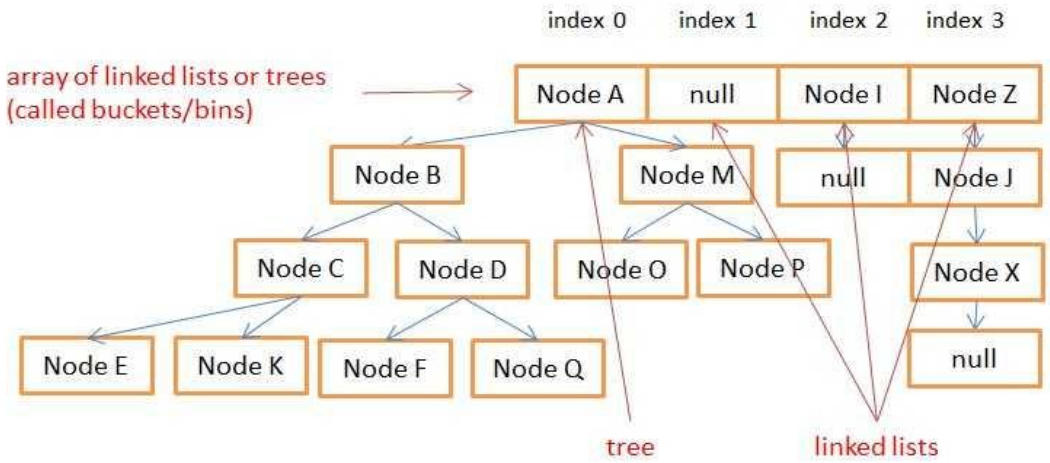
Here is a part of the Node implementation in JAVA 8:

```
static class Node<K,V> implements Map.Entry<K,V> {
          final int hash;
          final K key;
          V value;
          Node<K,V> next;

     }
```

Nodes can be extended to TreeNodes. A TreeNode is a red-black tree structure that stores really more information so that it can add, delete or get an element in O (log(n)).

```
    static final class TreeNode<K,V> extends
LinkedHashMap.Entry<K,V> {
        final int hash; // inherited from Node<K,V>
        final K key; // inherited from Node<K,V>
        V value; // inherited from Node<K,V>
        Node<K,V> next; // inherited from Node<K,V>
        Entry<K,V> before, after;// inherited from
LinkedHashMap.Entry<K,V>
        TreeNode<K,V> parent;
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;
        boolean red;
    }
```

Red black trees are self-balancing binary search trees. Their inner mechanisms ensure that their length is always in log(n) despite new adds or removes of nodes. The main advantage to use those trees is in a case where many data are in the same index (bucket) of the inner table, the search in a tree will cost **O(log(n))** whereas it would have cost **O(n)** with a linked list.



In com.yash.dao package create an interface InternsDAO

```
package com.yash.dao;
import java.util.Map;
import java.util.Optional;
import com.yash.entity.Interns;
public interface InternsDAO {
    Map<Integer,Interns> getAllInterns();
    Optional<Interns> getInternById(int internId);
    boolean storeInternData(Interns intern);
    boolean updateIntern(Interns intern);
    boolean updateInternLevel(Interns intern);
    boolean removeIntern(int internId);
}
```

We are following principle “Design to interface and not implementation”. In this approach first we chart out all possible data operations for business entity Interns and not worrying about implementation.

One of important principle of Object oriented programming is Abstraction.

Expose only what is relevant and hide what is not relevant.

Say for example you are driving to your destination by a car. To you as a driver of car it is been exposed how to start engine but not internal working of engine as it is not relevant.

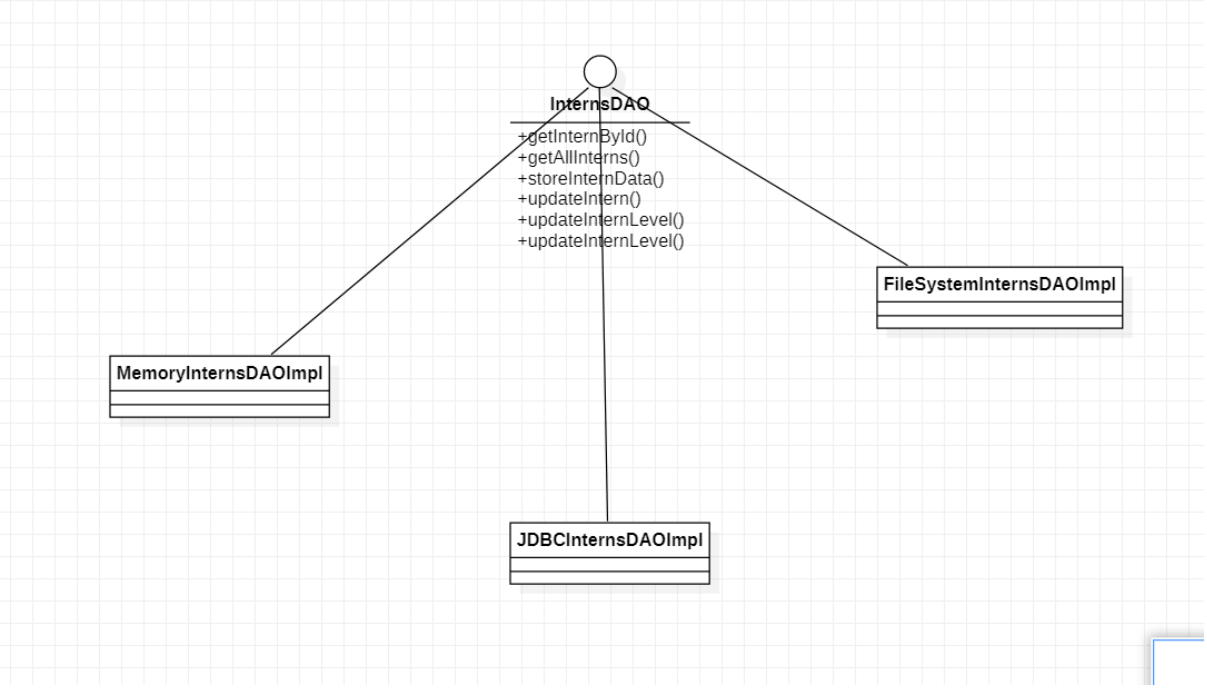
There could be different implementation of above interface like memory implementation, file system implementation or database implementation.

Using interfaces we are achieving abstraction wherein we will expose interface to business layer and not implementation details as they are of no relevance to business layer. Business layer will do computation on data received by data layer and not bothered about source or destination of data. Similarly we will expose business services through interface to UI layer.

To give reference of interface, we will create factory class in com.yash.helper package.

In initial release of application we will provide memory implementation of data operation for business entity Interns. In future releases, we may provide file system implementation or database implementation.

If implementation changes, business layer will not be affected as we will provide only reference of InternDAO interface through factory class.



We will utilise Collection Framework and new feature Stream introduced in Java 8.

Lambda Expressions

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button.

Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

- **Old**

```
button.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    doSomethingWith(e);
  }
});
```

- **New**

```
button.addActionListener(e -> doSomethingWith(e));
```

Another example, say we have array of String and we want to sort elements based on length.

We can drop interface name and method name.

- **Java 7 example**

```
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.length() - s2.length());
  }
});
```

- **Java 8 alternative**

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

An interface which has only one abstract method is called Functional Interface annotated by @FunctionalInterface. Implementation of method can be provided using lambda expression instead of anonymous class.

Built-in Functional Interfaces in Java

Java contains a set of functional interfaces designed for commonly occuring use cases, so you don't have to create your own functional interfaces for every little use case.

The Java Function interface (java.util.function.Function) interface is one of the most central functional interfaces in Java. The Function interface represents a function (method) that takes a single parameter and returns a single value.

Here is how the Function interface definition looks:

```
public interface Function<T,R> {

    public <R> apply(T parameter);
}
```

```
        Function<String,Integer> func=(s)->{
                return Integer.parseInt(s);
            };

            func.apply("10");
```

The Function interface actually contains a few extra methods in addition to the methods listed above, but since they all come with a default implementation, you do not have to implement these extra methods.

Predicate

The Java Predicate interface, java.util.function.Predicate, represents a simple function that takes a single value as parameter, and returns true or false. Here is how the Predicate functional interface definition looks:

```
public interface Predicate {
    boolean test(T t);
}
```

```
            Predicate<Integer> p=(i)->i%2==0;
            boolean result=p.test(10);
```

Predicate is widely used along with stream for filtering objects.

UnaryOperator

The Java UnaryOperator interface is a functional interface that represents an operation which takes a single parameter and returns a parameter of the same type.

```
UnaryOperator<Interns> unaryOperator =
        (intern) -> { intern.internFirstName = "New Name"; return intern;
};
```

The `UnaryOperator` interface can be used to represent an operation that takes a specific object as parameter, modifies that object, and returns it again - possibly as part of a functional stream processing chain.

BinaryOperator

The Java BinaryOperator interface is a functional interface that represents an operation which takes two parameters and returns a single value. Both parameters and the return type must be of the same type.

The Java BinaryOperator interface is useful when implementing functions that sum, subtract, divide, multiply etc. two elements of the same type, and returns a third element of the same type.

```
BinaryOperator<SemesterMarks> binaryOperator =
        (total, semester2Marks) -> { total.add(semester2Marks); return
total; };
```

Supplier

The Java Supplier interface is a functional interface that represents an function that supplies a value of some sorts. The Supplier interface can also be thought of as a factory interface.

```
Supplier<Integer> supplier = () -> new Integer((int) (Math.random() *
1000D));
```

This Java `Supplier` implementation returns a new `Integer` instance with a random value between 0 and 1000.

Consumer

The Java Consumer interface is a functional interface that represents an function that consumes a value without returning any value. A Java Consumer implementation could be printing out a value, or writing it to a file, or over the network etc. Here is an example implementation of the Java Consumer interface:

```
Consumer<InternsModel> consumer = (internsModel) ->
System.out.println(internsModel.internsFirstName);
```

This Java Consumer implementation prints the value passed as parameter to it out to System.out.

Streams

Streams are wrappers around data sources such as arrays or lists. Support many convenient and high-performance operations expressed succinctly with lambdas, executed sequentially or in parallel.

Characteristics of Stream

- Not data structures – Streams have no storage; they carry values from a source through a pipeline of operations.

- They also never modify the underlying data structure (e.g., the List or array that the Stream wraps) • Designed for lambdas – All Stream operations take lambdas as arguments

- Do not support indexed access – You can ask for the first element, but not the second or third or last element – But, see next bullet

- Can easily be output as Lists or arrays – Simple syntax to build a List or array from a Stream

- Lazy – Most Stream operations are postponed until it is known how much data is eventually needed E.g., if you do a 10-second-per-item operation on a 100-element stream, then select the first entry, it takes 10 seconds, not 1000 seconds

- Parallelizable – If you designate a Stream as parallel, then operations on it will automatically be done in parallel, without having to write explicit fork/join or threading code

- Can be unbounded – Unlike with collections, you can designate a generator function, and clients can consume entries as long as they want, with values being generated on the fly

```
Set<Map.Entry<Integer,Interns>>
entries=internsMap.entrySet();
         Map.Entry<Integer,Interns>
internEntry=entries.stream()
               .filter((entry)->entry.getKey()==internId)
               .collect(toSingleton());
```

forEach vs. for Loops

```
boolean[] internUpdated= {false};
         internsMap.forEach((k,v)->{

               if(k.equals(intern.getId())) {
                    internsMap.put(intern.getId(),intern);
                    internUpdated[0]=true;
```

```
            }
        });
```

Advantages of forEach

- designed for lambdas
- Marginally more succinct
- Reusable .You can save the lambda and use it again
- Can be made parallel with minimal effort • someStream.parallel().forEach(someLambda);

Method reference

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

Reference to an static Method of a Particular class

```
List<String> names = Arrays.asList("sabbir", "amit", "rohan");
```

We can achieve this by leveraging a simple lambda expression calling the *StringUtils.capitalize()* method directly:

```
1    messages.forEach(word -> StringUtils.capitalize(word));
```

Or using method reference

```
1    messages.forEach(StringUtils::capitalize);
```

Reference to an Instance Method of a Particular Object

```
public class Interns {

    private int id;
    private String internFirstName;
    // standard constructor, getters and setters
}

public class InternComparator implements Comparator {

    @Override
    public int compare(Intern a, Intern b) {
        return a.getInternFirstName().compareTo(b.getInternFirstName());
    }

}
```

```
internsList.stream()
  .sorted((a, b) -> InternComparator.compare(a, b));


internList().stream()
  .sorted(InternComparator::compare);
```

Reference to an Instance Method of an Arbitrary Object of a Particular Type

```
List<Integer> numbers = Arrays.asList(5, 3, 50, 24, 40, 2, 9, 18);

numbers.stream()
  .sorted((a, b) -> a.compareTo(b));
numbers.stream()
  .sorted(Integer::compareTo);
```

Reference to a Constructor

```
List<String> names = Arrays.asList("sabbir", "amit", "ketan", "rohit");
```

```
public Interns(String internFirstName) {
    this.internFirstName = internFirstName;
```

```
}
```

```
1    names.stream()
2      .map(Interns::new)
3      .toArray(Interns[]::new);
```

Create an implementation class MemoryInternsDAOImpl.

We have utilized new features of Java 8 in MemoryInternsDAOImpl.

Complete listing of MemoryInternsDAOImpl is as follows,

```
package com.yash.dao;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;
import java.util.Set;
import java.util.stream.Collector;
import java.util.stream.Collectors;
import com.yash.entity.Interns;
public class MemoryInternsDAOImpl implements InternsDAO{
    private static Map<Integer,Interns> internsMap=new
HashMap<>();
    public MemoryInternsDAOImpl() {
    }
    public static <T> Collector<T, ?, T> toSingleton() {
        return Collectors.collectingAndThen(
                Collectors.toList(),
                list -> {
                    if (list.size() != 1) {
                        throw new
IllegalStateException();
                    }
                    return list.get(0);
                }
        );
    }
    @Override
    public Map<Integer,Interns> getAllInterns() {
```

```
        // TODO Auto-generated method stub
        return internsMap;
    }
    @Override
    public Optional<Interns> getInternById(int internId)
{
        // TODO Auto-generated method stub
        Set<Map.Entry<Integer,Interns>>
entries=internsMap.entrySet();
       Map.Entry<Integer,Interns>
internEntry=entries.stream()
             .filter((entry)->entry.getKey()==internId)
             .collect(toSingleton());
      return
Optional.ofNullable(internEntry.getValue());
    }
    @Override
    public boolean storeInternData(Interns intern) {
        // TODO Auto-generated method stub
        internsMap.put(intern.getId(), intern);
        boolean[] internStored= {false};
        internsMap.forEach((k,v)->{
            if(k==intern.getId()) {
                internStored[0]=true;
            }
        });
        return internStored[0];
    }
    @Override
    public boolean updateIntern(Interns intern) {
        // TODO Auto-generated method stub

        boolean[] internUpdated= {false};
        internsMap.forEach((k,v)->{

            if(k.equals(intern.getId())) {
                internsMap.put(intern.getId(),intern);
                internUpdated[0]=true;
            }
        });

        if(internUpdated[0])
```

```
            return true;
            else
            return false;
        }
        @Override
        public boolean updateInternLevel(Interns intern) {
            boolean[] internUpdated= {false};
            internsMap.forEach((k,v)->{
                if(k.equals(intern.getId())) {
                    Interns
interns=internsMap.get(intern.getId());
                    interns.setLevel(intern.getLevel());
                    internsMap.put(k, interns);
                    internUpdated[0]=true;
                }
            });
            if(internUpdated[0])
            return true;
            else
            return false;
        }
        @Override
        public boolean removeIntern(int internId) {
            // TODO Auto-generated method stub
            Set<Map.Entry<Integer,Interns>>
entries=internsMap.entrySet();
               Map.Entry<Integer,Interns>
internEntry=entries.stream()
                    .filter((entry)-
>entry.getKey()==internId)
                    .collect(toSingleton());
            internsMap.remove(internEntry.getKey());
             boolean ifDeleted[]= {false};
            internsMap.forEach((k,v)->{
                if(k!=internId) {
                    ifDeleted[0]=true;
                }
            });
            if(internsMap.isEmpty()) {
                ifDeleted[0]=true;
            }
            return ifDeleted[0];
```

```
    }
}
```

Stream.collect() is one of the Java 8's *Stream API*'s terminal methods. It allows to perform mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in a *Stream* instance.

The strategy for this operation is provided via *Collector* interface implementation.

ToList collector can be used for collecting all *Stream* elements into a *List* instance

toSingleTon() is our method to retrieve only single object from collection. If list will be empty it will throw IllegalStateException.

The **collectingAndThen(Collector downstream, Function finisher)** method of **class collectors in Java**, which adopts **Collector** so that we can perform an additional finishing transformation.

Syntax :

```
public static <T, A, R, RR>
       Collector <T, A, RR>
       collectingAndThen(Collector <T, A, R> downstream,
                         Function <R, RR> finisher)
```

Where,

- **T**: The type of the input elements
- **A**: Intermediate accumulation type of the downstream collector
- **R**: Result type of the downstream collector
- **RR**: Result type of the resulting collector

Optional Class

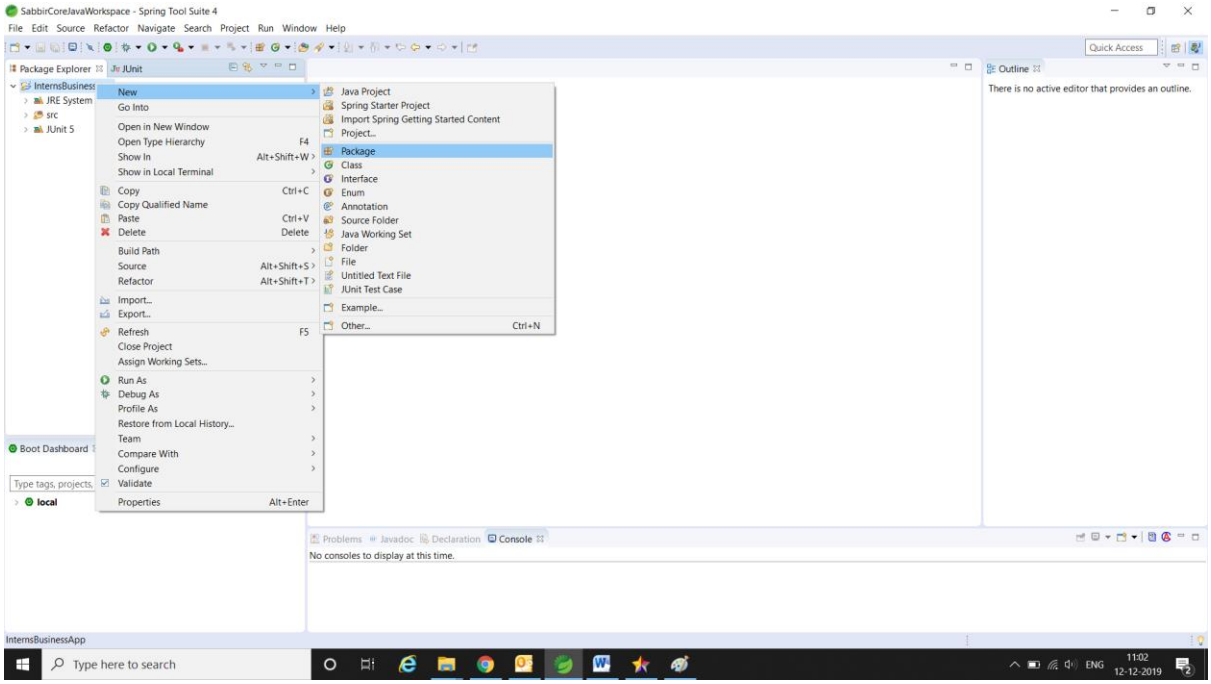
Optional is a container object used to contain not-null objects. **Optional** object is used to represent null with absent value. This **class** has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values.

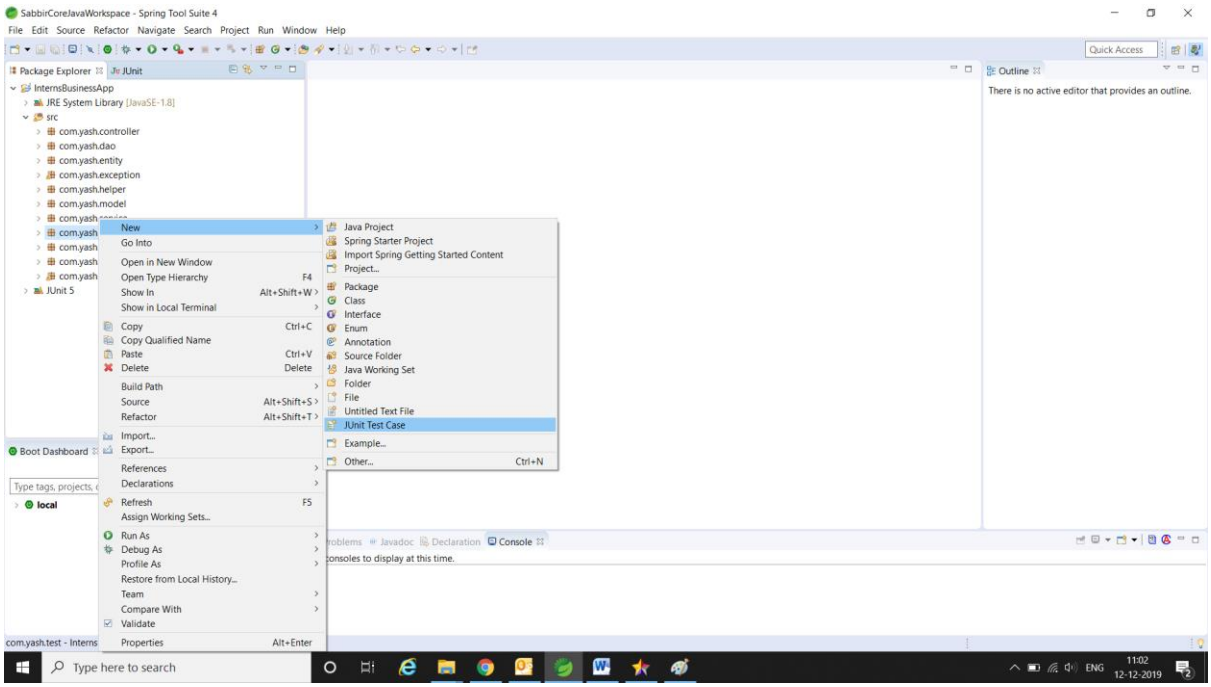
TDD can be defined as a programming practice that instructs developers to write new code only if an automated test has failed. This avoids duplication of code. **TDD** means "**Test Driven Development**". ... This helps to avoid duplication of code as we write a small amount of code at a time in order to pass tests.

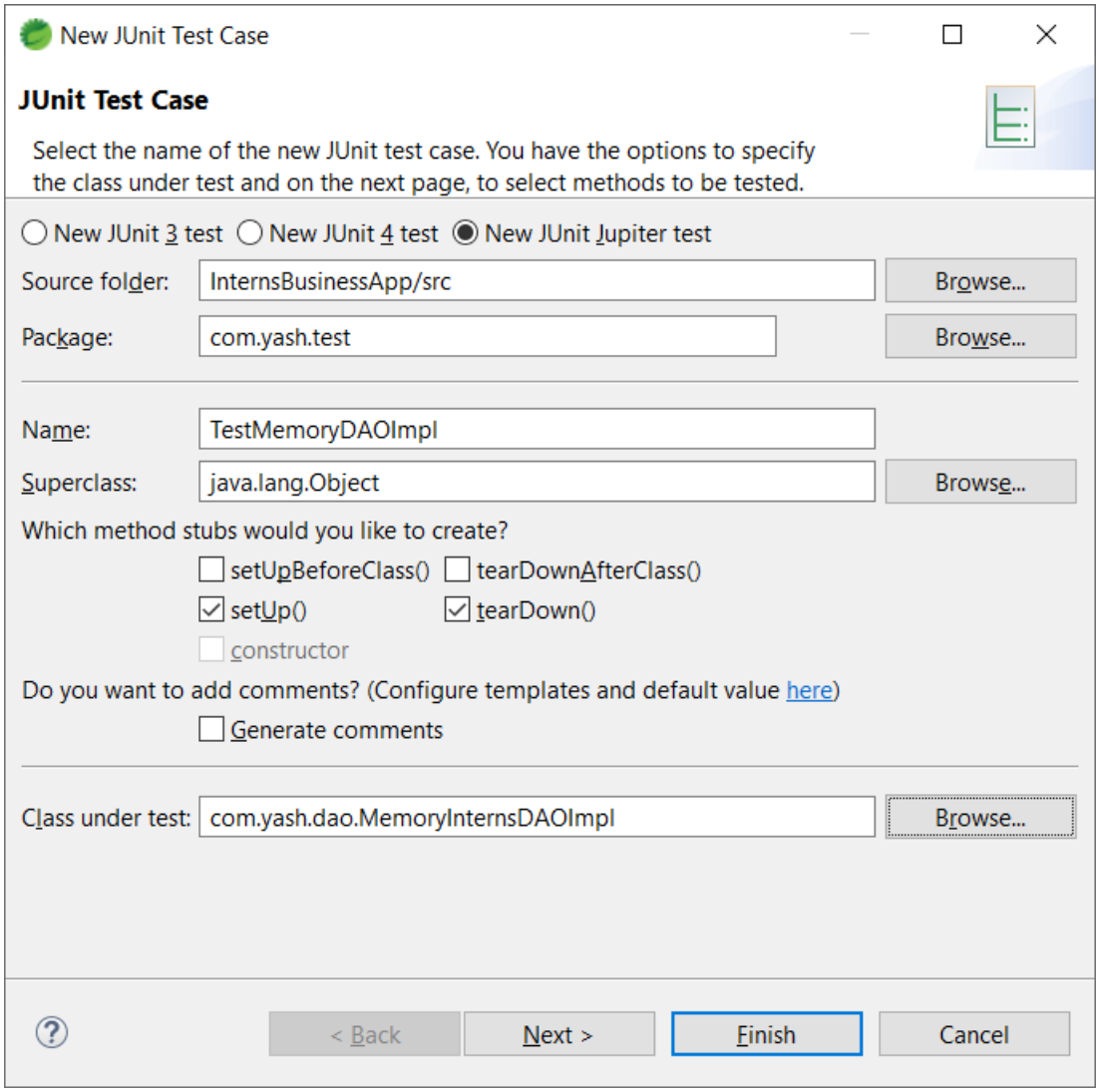
The ideal approach should be write one implementation in MemoryInternsDAOImpl and write test condition in TestMemoryInternsDAOImpl. Run test conditions related to this method positive,negative and exception. If all test conditions pass, then only move to next method.

Before we move to next layer we will perform unit testing of **MemoryInternsDAOImpl**

Create a new package com.yash.test and JUnit Test case TestMemoryInternsDAOImpl,


SabbirCoreJavaWorkspace - Spring Tool Suite 4
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
JUnit
InternsBusiness
JRE System
src
JUnit 5
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Show in Local Terminal
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Assign Working Sets...
Run As
Debug As
Profile As
Restore from Local History...
Team
Compare With
Configure
Validate
Properties Alt+Enter
Java Project
Spring Starter Project
Import Spring Getting Started Content
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
JUnit Test Case
Example...
Other... Ctrl+N
Outline
There is no active editor that provides an outline.
Boot Dashboard
local
Problems Javadoc Declaration Console
No consoles to display at this time.
InternsBusinessApp
Type here to search
ENG
11:02
12-12-2019


SabbirCoreJavaWorkspace - Spring Tool Suite 4
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
JUnit
InternsBusinessApp
JRE System Library [JavaSE-1.8]
src
com.yash.controller
com.yash.dao
com.yash.entity
com.yash.exception
com.yash.helper
com.yash.model
JUnit 5
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Show in Local Terminal
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
References
Declarations
Refresh F5
Assign Working Sets...
Run As
Debug As
Profile As
Restore from Local History...
Team
Compare With
Validate
Properties Alt+Enter
Java Project
Spring Starter Project
Import Spring Getting Started Content
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
JUnit Test Case
Example...
Other... Ctrl+N
Outline
There is no active editor that provides an outline.
Boot Dashboard
local
Problems Javadoc Declaration Console
consoles to display at this time.
Type here to search
ENG
11:02
12-12-2019



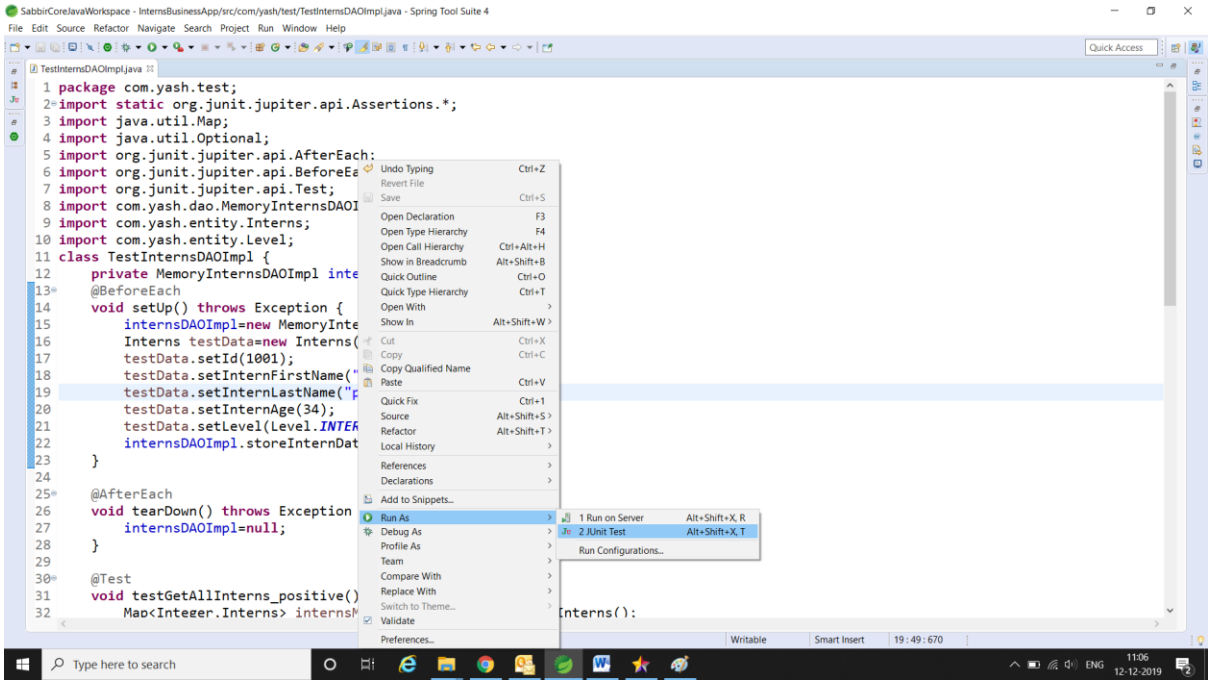
We will perform positive testing for all methods in MemoryInternsDAOimpl.

```
package com.yash.test;
import static org.junit.jupiter.api.Assertions.*;
import java.util.Map;
import java.util.Optional;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import com.yash.dao.MemoryInternsDAOImpl;
import com.yash.entity.Interns;
import com.yash.entity.Level;
class TestInternsDAOImpl {
    private MemoryInternsDAOImpl internsDAOImpl=null;
```

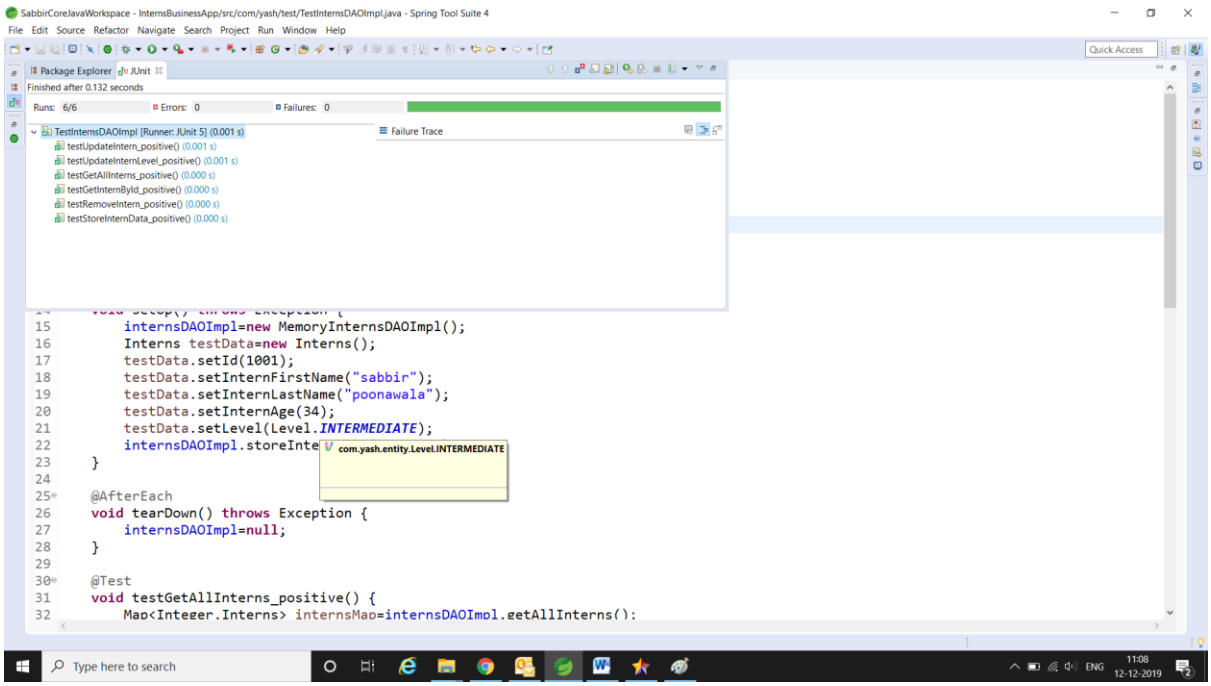
```
    @BeforeEach
    void setUp() throws Exception {
        internsDAOImpl=new MemoryInternsDAOImpl();
        Interns testData=new Interns();
        testData.setId(1001);
        testData.setInternFirstName("sabbir");
        testData.setInternLastName("poonawala");
        testData.setInternAge(34);
        testData.setLevel(Level.INTERMEDIATE);
        internsDAOImpl.storeInternData(testData);
    }

    @AfterEach
    void tearDown() throws Exception {
        internsDAOImpl=null;
    }

    @Test
    void testGetAllInterns_positive() {
        Map<Integer,Interns>
internsMap=internsDAOImpl.getAllInterns();
        assertEquals(internsMap.size()>0,true);
    }
    @Test
    void testGetInternById_positive() {
     Optional<Interns>
internsOptional=internsDAOImpl.getInternById(1001);
     assertEquals(1001,internsOptional.get().getId());
    }
    @Test
    void testStoreInternData_positive() {
        Interns intern=new Interns();
        intern.setId(1002);
        intern.setInternFirstName("amit");
        intern.setInternLastName("kumar");
        intern.setInternAge(21);
        intern.setLevel(Level.BEGINNER);
        boolean
internStored=internsDAOImpl.storeInternData(intern);
        assertEquals(true,internStored);
    }
    @Test
```

```
    void testUpdateIntern_positive() {
        Interns internUpdate=new Interns();
        internUpdate.setId(1001);
        internUpdate.setInternFirstName("amit");
        internUpdate.setInternLastName("kumar");
        internUpdate.setInternAge(21);
        internUpdate.setLevel(Level.INTERMEDIATE);
        boolean
internUpdated=internsDAOImpl.updateIntern(internUpdate);
        assertEquals(true,internUpdated);
    }
    @Test
    void testUpdateInternLevel_positive() {
        Interns internUpdateLevel=new Interns();
        internUpdateLevel.setId(1001);
        internUpdateLevel.setLevel(Level.ADVANCED);
        boolean
internUpdatedLevel=internsDAOImpl.updateInternLevel(intern
UpdateLevel);
        assertEquals(true,internUpdatedLevel);
    }
    @Test
    void testRemoveIntern_positive() {
        boolean
internDeleted=internsDAOImpl.removeIntern(1001);
        assertEquals(true,internDeleted);
    }
}
```

Run JUnit test,



Ensure all test condition passes. Write negative and exception test conditions on your own.



Business Service Layer

The **business logic layer** is the **business** components that provide OAGIS services to return data or start **business** processes. The presentation **layer** uses these OAGIS services to display data, or to invoke a **business** process. The **business logic** provides data required by the presentation **layer.**

Chart out business operations that have to be exposed to presentation layer in business interface.

Create an interface InternsService in com.yash.service with below methods,

```
package com.yash.service;
import java.util.List;
import java.util.Optional;
import com.yash.model.InternsModel;
public interface InternsService {
    List<InternsModel> retrieveInternsService();
    List<InternsModel> retrieveInternsService(String
sortBy);
    Optional<InternsModel> retrieveInternsByIdService(int
internId);
    String registerInternService(InternsModel
internsModel);
    String updateInternService(InternsModel
internsModel);
    String updateInternLevelService(InternsModel
internsModel);
    String removeInternService(int internId);
}
```

To return reference of InternsDAO interface to InternsServiceImpl, and InternsService to InternsController, we will create factory class

FactoryInterns in com.yash.helper package,

```
package com.yash.helper;
import com.yash.dao.InternsDAO;
import com.yash.dao.MemoryInternsDAOImpl;
import com.yash.service.InternsService;
import com.yash.service.InternsServiceImpl;
```

```
public class FactoryInterns {
     public static InternsDAO createInternsDAO() {
          InternsDAO internsDAO=new MemoryInternsDAOImpl();
          return internsDAO;
     }
     public static InternsService createInternsService() {
          InternsService internsService=new
InternsServiceImpl();
          return internsService;
     }

}
```

If we utilise frameworks (like Spring), they act as factory to inject object where dependency arises.

Create a custom exception handling class InternsException in com.yash.exception package.

If business rules are not followed, we will throw explicitly an object of InternsException in service implementation class.

```
package com.yash.exception;
public class InternsException extends RuntimeException {
     private String message;
     public InternsException(String message) {
          super(message);
     }
     public String getMessage() {
          return message;
     }
}
```

And the implementation class InternsServiceImpl,

```
package com.yash.service;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Collectors;
import com.yash.dao.InternsDAO;
import com.yash.entity.Interns;
import com.yash.exception.InternsException;
import com.yash.helper.FactoryInterns;
import com.yash.model.InternsModel;
import com.yash.entity.Level;
public class InternsServiceImpl implements InternsService
{
     public InternsServiceImpl() {

     this.internsDAO=FactoryInterns.createInternsDAO();
     }
     private InternsDAO internsDAO;
     public List<InternsModel> retrieveInternsService() {
          // TODO Auto-generated method stub
          Map<Integer,Interns> interns=
internsDAO.getAllInterns();
          List<Interns>
internsList=interns.values().stream().collect(Collectors.t
oList());
          List<InternsModel> internsModelList=new
ArrayList<>();
          internsList.forEach((intern)->{
               InternsModel model=new InternsModel();
               model.setId(intern.getId());

     model.setInternFirstName(intern.getInternFirstName())
;

     model.setInternLastName(intern.getInternLastName());
               model.setInternAge(intern.getInternAge());
```

```
            model.setLevel(intern.getLevel().toString());
                internsModelList.add(model);
            });
            if(interns.isEmpty()){
                throw new InternsException("No interns
records found");
            }else{
                return internsModelList;
            }
        }
        @Override
        public Optional<InternsModel>
retrieveInternsByIdService(int internId) {
            // TODO Auto-generated method stub
            Optional<Interns>
internOptional=internsDAO.getInternById(internId);
            InternsModel model=new InternsModel();
            if(internOptional.isPresent()) {
                Interns intern=internOptional.get();
            model.setId(intern.getId());

        model.setInternFirstName(intern.getInternFirstName())
;

        model.setInternLastName(intern.getInternLastName());
            model.setInternAge(intern.getInternAge());
            model.setLevel(intern.getLevel().toString());
           }else {
                throw new InternsException("Intern not
found");
            }
            return Optional.of(model);
        }
        public Level
determineLevelBySemesterMarks(InternsModel internsModel){
            int sem1Marks=internsModel.getSemester1Marks();
            int sem2Marks=internsModel.getSemester2Marks();
            int sem3Marks=internsModel.getSemester3Marks();
            int semAverage=(sem1Marks+sem2Marks+sem3Marks)/3;
            if(semAverage<=50){
```

```
                throw new InternsException("Intern did not
match eligibility");
            }
            if(semAverage>50 && semAverage<=60){
                return Level.BEGINNER;
            }else if(semAverage>60 && semAverage<70){
                return Level.INTERMEDIATE;
            }else{
                return Level.ADVANCED;
            }
        }
        @Override
        public String registerInternService(InternsModel
internsModel) {
            // TODO Auto-generated method stub
            Level
level=determineLevelBySemesterMarks(internsModel);

            Interns interns=new Interns();
            interns.setId(internsModel.getId());

        interns.setInternFirstName(internsModel.getInternFirs
tName());

        interns.setInternLastName(internsModel.getInternLastN
ame());

        interns.setInternAge(internsModel.getInternAge());
            interns.setLevel(level);
            boolean
isInternRegistered=internsDAO.storeInternData(interns);
            if(isInternRegistered)
                return "success";
            else
                return "fail";
        }
        @Override
        public String updateInternService(InternsModel
internsModel) {
            // TODO Auto-generated method stub
            Level
level=determineLevelBySemesterMarks(internsModel);
```

```
        Interns interns=new Interns();
        interns.setId(internsModel.getId());
    interns.setInternFirstName(internsModel.getInternFirs
tName());
    interns.setInternLastName(internsModel.getInternLastN
ame());
    interns.setInternAge(internsModel.getInternAge());
        interns.setLevel(level);
        boolean
checkUpdation=internsDAO.updateIntern(interns);
        if(checkUpdation==false)
            throw new InternsException("Updation
failed.....");
       if(checkUpdation)
        return "success";
       else
           return "fail";
    }
    @Override
    public String updateInternLevelService(InternsModel
internsModel) {
        // TODO Auto-generated method stub
        Level
level=determineLevelBySemesterMarks(internsModel);
        Interns interns=new Interns();
        interns.setId(internsModel.getId());
        interns.setLevel(level);
        boolean
checkUpdation=internsDAO.updateInternLevel(interns);
        if(checkUpdation==false)
            throw new InternsException("Updation
failed.....");
         if(checkUpdation)
            return "success";
           else
               return "fail";
    }
    @Override
    public String removeInternService(int internId) {
```

```
        // TODO Auto-generated method stub
        boolean
checkRemoval=internsDAO.removeIntern(internId);
        if(checkRemoval==false)
            throw new InternsException("Deletion
failed....");
        if(checkRemoval)
            return "success";
        else
            return "fail";
    }
    @Override
    public List<InternsModel>
retrieveInternsService(String sortBy) {
        // TODO Auto-generated method stub
        List<InternsModel>
internsModelList=retrieveInternsService();
        switch(sortBy) {
        case "InternFirstName":
                                    Comparator<InternsModel>
sortByFirstName=(model1,model2)->{
                                    return
model1.getInternFirstName().compareTo(model2.getInternFirs
tName());
                                    };

Collections.sort(internsModelList,sortByFirstName);
                                    break;
        case "InternLastName":
                                    Comparator<InternsModel>
sortByLastName=(model1,model2)->{
                                return
model1.getInternFirstName().compareTo(model2.getInternFirs
tName());
                                    };

Collections.sort(internsModelList,sortByLastName);
                                break;
        case "InternAge":
                                    Comparator<InternsModel>
sortByAge=(model1,model2)->{
```

```
if(model1.getInternAge()>model2.getInternAge()) {
                                        return 1;
                                    }else
if(model1.getInternAge()<model2.getInternAge()) {
                                        return -1;
                                    }else {
                                        return 0;
                                    }
                                };

Collections.sort(internsModelList,sortByAge);
                                    break;
            default:
                                Comparator<InternsModel>
sortById=(model1,model2)->{

if(model1.getId()>model2.getId()) {
                                       return 1;
                                    }else
if(model1.getId()<model2.getId()) {
                                       return -1;
                                    }else {
                                       return 0;
                                    }
                                    };

Collections.sort(internsModelList,sortById);
                                  break;
            }
            return internsModelList;
        }
}
```

Service class will perform complex business logic and apply business rules on data flowing from presentation layer or persistence layer.

Simple computation we are performing in our application is to determine level of intern based on semester marks and check eligibility. In real scenario applications it will be complex business logic not only arithmetic computations.

Intern's data will be sorted based on intern name, age or id. We have utilised lambda expression to provide different sorting options using Comparator interface.

Comparator interface in Java8 is marked as Functional Interface. Comparator interface has one method compare() which takes two parameters as objects which are to be compared.

To provide implementation of compare method, in Java 8 we can utilise lambda expression instead of creating outer classes, inner classes or anonymous classes.

Before Java 8,

```
Collections.sort(internsModelIst,new Comparator<InternsModel>(){

@Override

public int compare(InternsModel m1,InternsModel m2){

return m1.getInternsFirstname().compareTo(m2.getInternsFirstName();
}

});
```

Cleaner approach in Java 8 using lambda expression,

```
Comparator<InternsModel> sortByFirstName=(model1,model2)->{
                                    return model1.getInternFirstName().compareTo(model2.getInternFirstName());
                                    };

Collections.sort(internsModelList,sortByFirstName);
```

Test cases to be written for InternsServiceImpl same as MemoryInternsDAOImpl.

Presentation layer

MVC architecture separates the application into three components which consists of Model, View and Controller. In **MVC** architecture, user interacts with the controller with the help of view. ... Typically 3-**layer** and **MVC** are used together and **MVC** acts as the **Presentation layer.**

Create a model class InternsModel in com.yash.model package.

You should never expose business entity or domain class to presentation layer. Business entity is typically mapped with physical resources (for e.g database) and will have business rules applied.

Model class is a storage medium of data flowing from UI or business layer which will be presented to UI by a view.

```
package com.yash.model;
public class InternsModel  {
     private int id;
     private String internFirstName;
     private String internLastName;
     private int internAge;
     private String level;
     private int semester1Marks;
     private int semester2Marks;
     private int semester3Marks;
     public InternsModel() {}
     public int getId() {
          return id;
     }
     public void setId(int id) {
          this.id = id;
     }
     public String getInternFirstName() {
          return internFirstName;
     }
     public void setInternFirstName(String
internFirstName) {
```

```
        this.internFirstName = internFirstName;
    }
    public String getInternLastName() {
        return internLastName;
    }
    public void setInternLastName(String internLastName)
{
        this.internLastName = internLastName;
    }

    public int getInternAge() {
        return internAge;
    }

    public void setInternAge(int internAge) {
        this.internAge = internAge;
    }

    public String getLevel() {
        return level;
    }

    public void setLevel(String level) {
        this.level = level;
    }

    public int getSemester1Marks() {
        return semester1Marks;
    }

    public void setSemester1Marks(int semester1Marks) {
        this.semester1Marks = semester1Marks;
    }

    public int getSemester2Marks() {
        return semester2Marks;
    }

    public void setSemester2Marks(int semester2Marks) {
        this.semester2Marks = semester2Marks;
    }
```

```
    public int getSemester3Marks() {
        return semester3Marks;
    }

    public void setSemester3Marks(int semester3Marks) {
        this.semester3Marks = semester3Marks;
    }

    public String fullName() {
        return internFirstName+internLastName;
    }
}
```

```
package com.yash.model;
public class InternsModel  {
    private int id;
    private String internFirstName;
    private String internLastName;
    private int internAge;
    private String level;
    private int semester1Marks;
    private int semester2Marks;
    private int semester3Marks;
    public InternsModel() {}
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getInternFirstName() {
        return internFirstName;
    }
    public void setInternFirstName(String
internFirstName) {
        this.internFirstName = internFirstName;
    }
    public String getInternLastName() {
        return internLastName;
    }
```

```
    public void setInternLastName(String internLastName)
{
        this.internLastName = internLastName;
    }

    public int getInternAge() {
        return internAge;
    }

    public void setInternAge(int internAge) {
        this.internAge = internAge;
    }

    public String getLevel() {
        return level;
    }

    public void setLevel(String level) {
        this.level = level;
    }

    public int getSemester1Marks() {
        return semester1Marks;
    }

    public void setSemester1Marks(int semester1Marks) {
        this.semester1Marks = semester1Marks;
    }

    public int getSemester2Marks() {
        return semester2Marks;
    }

    public void setSemester2Marks(int semester2Marks) {
        this.semester2Marks = semester2Marks;
    }

    public int getSemester3Marks() {
        return semester3Marks;
    }

    public void setSemester3Marks(int semester3Marks) {
```

```
        this.semester3Marks = semester3Marks;
    }

    public String fullName() {
        return internFirstName+internLastName;
    }
}
```

Input given by user have to be thoroughly validated. If input is not valid we will throw custom exception object of below class,

Create a class ValidationException in com.yash.exception package,

```
package com.yash.exception;
public class ValidationException extends RuntimeException{
    private String message;
    public ValidationException(String message) {
        super(message);
    }
    public String getMessage() {
        return message;
    }
}
```

For validating user input, create a InternsValidator class in com.yash.validation package.

```
package com.yash.validation;
import java.util.ArrayList;
import java.util.List;
import com.yash.model.InternsModel;
public class InternsModelValidator {
public boolean validate(InternsModel model) {
        boolean result=false;
        if(validString(model.getInternFirstName()) &&
validString(model.getInternLastName()) &&
validNumber(model.getId()) &&
validAge(model.getInternAge())){
            result=true;
```

```
        }
        return result;
    }
    public boolean validString(String val) {
        boolean result=false;
        char chars[]=val.toCharArray();
        List<Character> alphabets=new ArrayList<>();
        for(int i=97;i<=122;i++) {
            alphabets.add((char)i);
        }
        for(char ch:chars) {
            if(alphabets.contains(ch)) {
                result=true;
            }else {
                return false;
            }
        }
        return result;
    }
    public boolean validNumber(int number) {
        boolean result=false;
        String data=String.valueOf(number);
        if(data.matches(".*[0-9]")) {
            result=true;
        }
        return result;
    }
    public boolean validAge(int internAge) {
        boolean result=false;
        if(internAge>18) {
            result=true;
        }
        return result;
    }
}
```

Create following enum's in com.yash.controller package,

RequestType will determine request for intern data coming from UI class.

```
package com.yash.controller;
public enum RequestType {
     NAME("name"),LEVEL("level"),SORT("sort");
     private String val;
     private RequestType(String val) {
          this.val=val;
     }
     public String getVal() {
          return val;
     }
}
```

And enum SortType to determine sorting options selected by end-user

```
package com.yash.controller;
public enum SortType {
     ID("ID"),INTERNFIRSTNAME("INTERNFIRSTNAME"),INTERNLAS
TNAME("INTERNLASTNAME"),INTERNAGE("INTERNAGE");
     private String val;
     private SortType(String val) {
          this.val=val;
     }
     public String getVal() {
          return val;
     }
}
```

MVC Controllers are responsible for controlling the flow of the application execution. When you make a request (means request a view) to **MVC** applications, a **controller** is responsible for returning the response to that request.

A **controller** can have one or more actions.

```
package com.yash.controller;
import java.util.List;
import com.yash.helper.FactoryInterns;
import com.yash.model.InternsModel;
import com.yash.service.InternsService;
import com.yash.validation.InternsModelValidator;
```

```
import com.yash.view.InternsView;
import com.yash.view.MainView;
public class InternsController {
    private InternsService internsService;
    InternsView internView=new InternsView();
    public InternsController() {

    this.internsService=FactoryInterns.createInternsServi
ce();
    }
   public void handleRetrieveInterns(RequestType request)
{
         List<InternsModel>
models=internsService.retrieveInternsService();
         MainView mainView=new MainView();
         switch(request) {

         case NAME: internView.showInternName(models);
                    mainView.viewInternMenu();
                     break;
         case LEVEL:internView.showInternLevel(models);
                    mainView.viewInternMenu();
                     break;
         default:   mainView.invalidOption();
                     break;

         }
    }

   public void handleRetrieveSortedInterns(SortType
sortType) {
         MainView mainView=new MainView();
         switch(sortType) {

         case ID  : List<InternsModel>
internsModelListSortedById=internsService.retrieveInternsS
ervice("ID");

internView.showSortedIntern(internsModelListSortedById);
                    mainView.viewSortedInternMenu();
                     break;
         case INTERNFIRSTNAME:
```

```
                    List<InternsModel>
internsModelListSortedByFirstName=internsService.retrieveI
nternsService("InternFirstName");

internView.showSortedIntern(internsModelListSortedByFirstN
ame);
                    mainView.viewSortedInternMenu();
                    break;

          case INTERNLASTNAME:
                    List<InternsModel>
internsModelListSortedByLastName=internsService.retrieveIn
ternsService("InternLastName");

internView.showSortedIntern(internsModelListSortedByLastNa
me);
                    mainView.viewSortedInternMenu();
                    break;

          case INTERNAGE:
                    List<InternsModel>
internsModelListSortedByAge=internsService.retrieveInterns
Service("InternAge");

internView.showSortedIntern(internsModelListSortedByAge);
                    mainView.viewSortedInternMenu();
                    break;

          default:  mainView.invalidOption();
                    break;
          }
    }
     public void handleRegisterIntern(InternsModel model)
{
          InternsModelValidator validator=new
InternsModelValidator();
          if(validator.validate(model)) {
          String
outcome=internsService.registerInternService(model);
          if(outcome.contentEquals("success")) {
               internView.showRegistrationSuccess(model);
          }else {
```

```
                internView.showRegistrationFailure(model);
            }
        }else {
            internView.validationFailedError();
        }
        }
        public void handleUpdateIntern(InternsModel model) {
            String
outcome=internsService.updateInternService(model);
            if(outcome.contentEquals("success")) {
                internView.showInternUpdateSuccess(model);
            }else {
                internView.showInternUpdateFailure(model);
            }

        }
        public void handleUpdateInternLevel(InternsModel
model) {
            String
outcome=internsService.updateInternLevelService(model);
            if(outcome.contentEquals("success")) {

        internView.showInternLevelUpdateSuccess(model);
            }else {

        internView.showInternLevelUpdateFailure(model);
            }
        }

        public void handleDeleteIntern(InternsModel model) {
            String
outcome=internsService.removeInternService(model.getId());
            if(outcome.contentEquals("success")) {
                internView.showDeleteSuccess(model);
            }else {
                internView.showDeleteFailure(model);
            }
        }
}
```

Test cases to be written for InternsController same as MemoryInternsDAOImpl.

View is a user interface. **View** displays data from the model to the user and also enables them to modify the data.

Create below view classes, InternsView

```
package com.yash.view;
import java.util.List;
import java.util.function.Consumer;
import com.yash.model.InternsModel;
public class InternsView {
 private MainView mainView=new MainView();
 Consumer<InternsModel> consumerName= intern->
System.out.println("Full Name:"+intern.fullName()+"\n");
 Consumer<InternsModel> consumerLevel= intern->
System.out.println("Level:"+intern.getLevel()+"\n");
 Consumer<InternsModel> consumerSortedInterns= intern->

 {
      System.out.println("Id:"+intern.getId()+"\n");
      System.out.println("Intern First
Name:"+intern.getInternFirstName()+"\n");
      System.out.println("Intern Last
Name:"+intern.getInternLastName()+"\n");
      System.out.println("Intern
Age:"+intern.getInternAge()+"\n");
      System.out.println("Intern
Level:"+intern.getLevel()+"\n");
 };

  public static void print(List<InternsModel> models,
Consumer<InternsModel> consumer){
  for(InternsModel model:models){
    consumer.accept(model);
   }
 }
    public void showInternName(List<InternsModel> models)
{
         print(models,consumerName);
```

```
    }
    public void showInternLevel(List<InternsModel>
models) {
        print(models,consumerLevel);
    }
    public void showSortedIntern(List<InternsModel>
models) {
        print(models,consumerSortedInterns);
    }
    public void showRegistrationSuccess(InternsModel
model) {

        System.out.println("\n Registration successful
for Intern id=>"+model.getId());
        mainView.mainMenu();
    }

    public void showRegistrationFailure(InternsModel
model) {
        System.out.println("\n Registration unsuccessful
for Intern id=>"+model.getId());
        mainView.mainMenu();
    }

  public void showInternUpdateSuccess(InternsModel model)
{
        System.out.println("\n successful updated for
Intern id=>"+model.getId());
        mainView.mainMenu();
    }

    public void showInternUpdateFailure(InternsModel
model) {
        System.out.println("\n Level updated failed for
Intern id=>"+model.getId());
        mainView.mainMenu();
    }

    public void showInternLevelUpdateSuccess(InternsModel
model) {
        System.out.println("\n Level successfully updated
for Intern id=>"+model.getId());
```

```
            mainView.mainMenu();
        }

        public void showInternLevelUpdateFailure(InternsModel
model) {
            System.out.println("\n Level updated failed for
Intern id=>"+model.getId());
            mainView.mainMenu();
        }

        public void showDeleteSuccess(InternsModel model){
            System.out.println("\n Intern record deleted for
Intern id=>"+model.getId());

        }
          public void showDeleteFailure(InternsModel model){
                System.out.println("\n Intern record
deletion failed for Intern id=>"+model.getId());
        }

          public void validationFailedError() {
              System.out.println("Data validation failed!!");
          }
}
```

Notice the utilisation of in-built functional interface Consumer in Java 8.

Only one method print() can be used to print different information about intern using implementation provided by Consumer.

```
Consumer<InternsModel> consumerName= intern-> {
        System.out.println("Intern Id:"+intern.getId());
        System.out.println("Full
Name:"+intern.fullName()+"\n");


 };
 Consumer<InternsModel> consumerLevel= intern->{
        System.out.println("Intern Id:"+intern.getId());
        System.out.println("Level:"+intern.getLevel()+"\n");
```

```
 };
 Consumer<InternsModel> consumerSortedInterns= intern->
 {
      System.out.println("Id:"+intern.getId()+"\n");
      System.out.println("Intern First
Name:"+intern.getInternFirstName()+"\n");
      System.out.println("Intern Last
Name:"+intern.getInternLastName()+"\n");
      System.out.println("Intern
Age:"+intern.getInternAge()+"\n");
      System.out.println("Intern
Level:"+intern.getLevel()+"\n");
 };
```

```
public static void print(List<InternsModel> models,
Consumer<InternsModel> consumer){
  for(InternsModel model:models){
    consumer.accept(model);
   }
 }
```

Create another view class MainView in com.yash.view package,

```
package com.yash.view;
import java.time.LocalDate;
import java.time.Period;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.StringTokenizer;
import com.yash.controller.InternsController;
import com.yash.controller.RequestType;
import com.yash.controller.SortType;
import com.yash.exception.ValidationException;
import com.yash.model.InternsModel;
import com.yash.validation.InternsModelValidator;
public class MainView {
    public void mainMenu() {
        System.out.println("\n");
```

```
        System.out.println("\t \t ========Main
Menu========");
        System.out.println("=>1. View Interns Details");
        System.out.println("=>2  View Sorted Intern
Details");
        System.out.println("=>3. Register Intern");
        System.out.println("=>4. Update Intern");
        System.out.println("=>5. Update Intern Level");
        System.out.println("=>6. Delete Intern");
        System.out.println("=>7. Exit");
        try(Scanner scanner=new Scanner(System.in);){
            System.out.print("\nOption:");
            int option=scanner.nextInt();
            switch(option) {
            case 1:viewInternMenu();
                   break;
            case 2:viewSortedInternMenu();
                   break;
            case 3:registerInternForm();
                    break;
            case 4:updateInternForm();
                   break;
            case 5:updateInternLevelForm();
                   break;
            case 6:deleteEmployeeForm();
                   break;
            case 7:System.exit(0);
                  break;
            default:System.out.println("!ERROR[SELECT
APPROPRIATE OPTION]");
                    mainMenu();
            }

        }catch(Exception e) {

            System.out.println("!ERROR[SELECT
APPROPRIATE OPTION]");
        }
    }
    public void viewInternMenu() {
        try(
                Scanner scanner=new Scanner(System.in);
```

```
          ){
                System.out.println("1. View Intern Name");
                System.out.println("2. View Intern Level");
                System.out.println("3. Main Menu");
                System.out.print("Enter choice:");
                int option=scanner.nextInt();
                InternsController internController=new
InternsController();

                if(option==1)

     internController.handleRetrieveInterns(RequestType.NA
ME);
                if(option==2)

     internController.handleRetrieveInterns(RequestType.LE
VEL);
             if(option==3)
               mainMenu();
          }catch(Exception e) {
                e.printStackTrace();
          }
     }
public void viewSortedInternMenu() {
          try(
                    Scanner scanner=new Scanner(System.in);
          ){
                System.out.println("1. Sort based on  Intern
Id");
                System.out.println("2. Sort based on  Intern
FirstName");
                System.out.println("3. Sort based on  Intern
LastName");
                System.out.println("4. Sort based on  Intern
Age");
                System.out.println("5. Main Menu");
                System.out.print("Enter choice:");
                int option=scanner.nextInt();
                InternsController internController=new
InternsController();
                if(option==1)
```

```
                internController.handleRetrieveSortedInterns(SortType
.ID);
                if(option==2)
                internController.handleRetrieveSortedInterns(SortType
.INTERNFIRSTNAME);
                if(option==3)
                internController.handleRetrieveSortedInterns(SortType
.INTERNLASTNAME);
                if(option==4)
                internController.handleRetrieveSortedInterns(SortType
.INTERNAGE);
                if(option==5)
                  mainMenu();
              }catch(Exception e) {
                  e.printStackTrace();
              }
          }
public void registerInternForm() {
              try(Scanner scanner=new Scanner(System.in);){
                  int id=0;
                  System.out.print("Intern Id:");
                  if(scanner.hasNextInt()) {
                  id=scanner.nextInt();
                  }
                  else {
                       try {
                            throw new
ValidationException("[!ERROR:Invalid Employee Id]");
                            }catch(ValidationException e) {
                                 e.printStackTrace();
                            
          System.out.println(e.getMessage());
                                 mainMenu();
                            }
                  }
                  InternsModelValidator validator=new
InternsModelValidator();
                  System.out.print("First Name:");
```

```
                String internFirstName=scanner.next();
                boolean
validfirstName=validator.validString(internFirstName);
                if(!validfirstName)
                     try {
                     throw new
ValidationException("[!ERROR:Invalid First Name]");
                     }catch(ValidationException e) {
                          e.printStackTrace();
                          System.out.println(e.getMessage());
                          mainMenu();
                     }
                System.out.print("Last Name:");
                String internLastName=scanner.next();

                boolean
validLastName=validator.validString(internLastName);
                if(!validLastName)
                     try {
                     throw new
ValidationException("[!ERROR:Invalid Last Name]");
                     }catch(ValidationException e) {
                          e.printStackTrace();
                          System.out.println(e.getMessage());
                          mainMenu();
                     }
                System.out.print("Date of
Birth(DD/MM/YYYY):");
                String dobDateString=scanner.next();

                StringTokenizer tokens=new
StringTokenizer(dobDateString,"/");
                List<String> tokensList=new ArrayList<>();
                while(tokens.hasMoreTokens()) {
                     tokensList.add(tokens.nextToken());
                }
                int
dayOfMonth=Integer.parseInt(tokensList.get(0));
                int
month=Integer.parseInt(tokensList.get(1));
                int
year=Integer.parseInt(tokensList.get(2));
```

```
                LocalDate dateOfBirth=LocalDate.of(year,
month-1, dayOfMonth);
                LocalDate now=LocalDate.now();
                Period diff = Period.between(dateOfBirth,
now);
                int internAge=diff.getYears();
                boolean
validAge=validator.validAge(internAge);
                if(!validAge)
                     try {
                     throw new
ValidationException("[!ERROR:Invalid Age]");
                     }catch(ValidationException e) {
                          e.printStackTrace();
                          System.out.println(e.getMessage());
                          mainMenu();
                     }
                int semester1Marks=0;
                System.out.print("Semester 1 Marks:");
                if(scanner.hasNextInt()) {
                     semester1Marks=scanner.nextInt();
                }
                else {
                     try {
                          throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                          }catch(ValidationException e) {

     System.out.println(e.getMessage());
                               mainMenu();
                          }
                }
                int semester2Marks=0;
                System.out.print("Semester 2 Marks:");
                if(scanner.hasNextInt()) {
                     semester2Marks=scanner.nextInt();
                }
                else {
                     try {
                          throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                          }catch(ValidationException e) {
```

```
                    System.out.println(e.getMessage());
                                mainMenu();
                            }
                        }
                        int semester3Marks=0;
                        System.out.print("Semester 3 Marks:");
                        if(scanner.hasNextInt()) {
                            semester3Marks=scanner.nextInt();
                        }
                        else {
                            try {
                                throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                            }catch(ValidationException e) {
                    System.out.println(e.getMessage());
                                mainMenu();
                            }
                        }
                        InternsModel model=new InternsModel();
                        model.setId(id);
                        model.setInternFirstName(internFirstName);
                        model.setInternLastName(internLastName);
                        model.setInternAge(internAge);
                        model.setSemester1Marks(semester1Marks);
                        model.setSemester2Marks(semester2Marks);
                        model.setSemester3Marks(semester3Marks);
                        InternsController controller=new
InternsController();
                        controller.handleRegisterIntern(model);
                    mainMenu();
                }catch(Exception e) {
                    e.printStackTrace();
                }
            }

public void updateInternForm() {
    try(Scanner scanner=new Scanner(System.in);){
```

```
          InternsModelValidator validator=new
InternsModelValidator();
          int id=0;
          System.out.print("Id:");
          if(scanner.hasNextInt()) {
          id=scanner.nextInt();
          }
          else {
               try {
                    throw new
ValidationException("[!ERROR:Invalid Id]");
                    }catch(ValidationException e) {
                         System.out.println(e.getMessage());
                         mainMenu();
                    }
          }
          System.out.print("First Name:");
          String internFirstName=scanner.next();

          boolean
validfirstName=validator.validString(internFirstName);
          if(!validfirstName)
               try {
               throw new
ValidationException("[!ERROR:Invalid First Name]");
               }catch(ValidationException e) {
                    System.out.println(e.getMessage());
                    mainMenu();
               }
          System.out.print("Last Name:");
          String internLastName=scanner.next();
          boolean
validLastName=validator.validString(internLastName);
          if(!validLastName)
               try {
               throw new
ValidationException("[!ERROR:Invalid Last Name]");
               }catch(ValidationException e) {
                    System.out.println(e.getMessage());
                    mainMenu();
               }
          System.out.print("Date of Birth(DD/MM/YYYY):");
```

```
            String dobDateString=scanner.next();

            StringTokenizer tokens=new
StringTokenizer(dobDateString,"/");

            List<String> tokensList=new ArrayList<>();
            while(tokens.hasMoreTokens()) {
                tokensList.add(tokens.nextToken());
            }

            int
dayOfMonth=Integer.parseInt(tokensList.get(0));
            int month=Integer.parseInt(tokensList.get(1));
            int year=Integer.parseInt(tokensList.get(2));
            LocalDate dateOfBirth=LocalDate.of(year, month-1,
dayOfMonth);
            LocalDate now=LocalDate.now();
            Period diff = Period.between(dateOfBirth, now);
            int internAge=diff.getYears();

            boolean
validsalary=validator.validAge(internAge);
            if(!validsalary)
                try {
                throw new
ValidationException("[!ERROR:Invalid Age]");
                }catch(ValidationException e) {
                    System.out.println(e.getMessage());
                    mainMenu();
                }

            int semester1Marks=0;
            System.out.print("Semester 1 Marks:");
            if(scanner.hasNextInt()) {
                semester1Marks=scanner.nextInt();
            }
            else {
                try {
                    throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                    }catch(ValidationException e) {
                        System.out.println(e.getMessage());
```

```
                        mainMenu();
                    }
            }

            int semester2Marks=0;
            System.out.print("Semester 2 Marks:");
            if(scanner.hasNextInt()) {
                semester2Marks=scanner.nextInt();
            }
            else {
                try {
                    throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                    }catch(ValidationException e) {
                        System.out.println(e.getMessage());
                        mainMenu();
                    }
            }

            int semester3Marks=0;
            System.out.print("Semester 3 Marks:");
            if(scanner.hasNextInt()) {
                semester3Marks=scanner.nextInt();
            }
            else {
                try {
                    throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                    }catch(ValidationException e) {
                        System.out.println(e.getMessage());
                        mainMenu();
                    }
            }

            InternsModel model=new InternsModel();
            model.setId(id);
            model.setInternFirstName(internFirstName);
            model.setInternLastName(internLastName);
            model.setInternAge(internAge);
            model.setSemester1Marks(semester1Marks);
            model.setSemester2Marks(semester2Marks);
            model.setSemester3Marks(semester3Marks);
```

```
            InternsController controller=new
InternsController();
            controller.handleUpdateIntern(model);

        mainMenu();
    }catch(Exception e) {
            System.out.println("!Error processing request.
Please try again later");
    }
}

public void updateInternLevelForm() {
    try(Scanner scanner=new Scanner(System.in);){
            InternsModelValidator validator=new
InternsModelValidator();
            int id=0;
            System.out.print("Id:");
            if(scanner.hasNextInt()) {
            id=scanner.nextInt();
            }
            else {
                try {
                    throw new
ValidationException("[!ERROR:Invalid Id]");
                    }catch(ValidationException e) {
                        System.out.println(e.getMessage());
                        mainMenu();
                    }
            }
            int semester1Marks=0;
            System.out.print("Semester 1 Marks:");
            if(scanner.hasNextInt()) {
                semester1Marks=scanner.nextInt();
            }
            else {
                try {
                    throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                    }catch(ValidationException e) {
                        System.out.println(e.getMessage());
                        mainMenu();
```

```
                    }
               }

               int semester2Marks=0;
               System.out.print("Semester 2 Marks:");
               if(scanner.hasNextInt()) {
                    semester2Marks=scanner.nextInt();
               }
               else {
                    try {
                         throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                         }catch(ValidationException e) {
                              System.out.println(e.getMessage());
                              mainMenu();
                         }
               }

               int semester3Marks=0;
               System.out.print("Semester 3 Marks:");
               if(scanner.hasNextInt()) {
                    semester3Marks=scanner.nextInt();
               }
               else {
                    try {
                         throw new
ValidationException("[!ERROR:Invalid Semester Marks]");
                         }catch(ValidationException e) {
                              System.out.println(e.getMessage());
                              mainMenu();
                         }
               }
               InternsModel model=new InternsModel();
               model.setId(id);
               model.setSemester1Marks(semester1Marks);
               model.setSemester2Marks(semester2Marks);
               model.setSemester3Marks(semester3Marks);

               InternsController controller=new
InternsController();
               controller.handleUpdateIntern(model);
```

```
            mainMenu();
        }catch(Exception e) {
            System.out.println("!Error processing request.
Please try again later");
        }
}

public void deleteEmployeeForm() {
        try(Scanner scanner=new Scanner(System.in);){

            int id=0;
            System.out.print("Id:");
            if(scanner.hasNextInt()) {
            id=scanner.nextInt();
            }
            else {
                try {
                    throw new
ValidationException("[!ERROR:Invalid Employee Id]");
                    }catch(ValidationException e) {
                        System.out.println(e.getMessage());
                        mainMenu();
                    }
            }
            InternsModel model=new InternsModel();
            model.setId(id);
            InternsController controller=new
InternsController();
            controller.handleDeleteIntern(model);

            mainMenu();
        }catch(Exception e) {
            System.out.println("!Error processing request.
Please try again later");
        }
}
    public void invalidOption() {
        System.out.println("!Error Invalid option");
}
        }
```

New Date-Time API in Java 8

New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API:

1. **Not thread safe:** Unlike old java.util.Date which is not thread safe the new date-time API is *immutable* and doesn't have setter methods.
2. **Less operations :** In old API there are only few date operations but the new API provides us with many date operations.

Java 8 under the package java.time introduced a new date-time API, most important classes among them are :

1. **Local :** Simplified date-time API with no complexity of timezone handling.
2. **Zoned :** Specialized date-time API to deal with various timezones.

LocalDate and LocalTime

```
    // Get the current date and time
        LocalDateTime currentTime = LocalDateTime.now();
        System.out.println("Current DateTime: " + currentTime);

        LocalDate date1 = currentTime.toLocalDate();
        System.out.println("date1: " + date1);

        Month month = currentTime.getMonth();
        int day = currentTime.getDayOfMonth();
        int seconds = currentTime.getSecond();

        System.out.println("Month: " + month +"day: " + day
+"seconds: " + seconds);

        LocalDateTime date2 =
currentTime.withDayOfMonth(10).withYear(2012);
        System.out.println("date2: " + date2);

        //12 december 2014
        LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);
        System.out.println("date3: " + date3);

        //22 hour 15 minutes
        LocalTime date4 = LocalTime.of(22, 15);
        System.out.println("date4: " + date4);
```

```
        //parse a string
        LocalTime date5 = LocalTime.parse("20:15:30");
        System.out.println("date5: " + date5);
```

Chrono Units Enum

```
        LocalDate today = LocalDate.now();
        System.out.println("Current date: " + today);

        //add 1 week to the current date
        LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
        System.out.println("Next week: " + nextWeek);

        //add 1 month to the current date
        LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);
        System.out.println("Next month: " + nextMonth);

        //add 1 year to the current date
        LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
        System.out.println("Next year: " + nextYear);

        //add 10 years to the current date
        LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
        System.out.println("Date after ten year: " + nextDecade);
    }
```

Period and Duration

```
    //Get the current date
        LocalDate date1 = LocalDate.now();
        System.out.println("Current date: " + date1);

        //add 1 month to the current date
        LocalDate date2 = date1.plus(1, ChronoUnit.MONTHS);
        System.out.println("Next month: " + date2);

        Period period = Period.between(date2, date1);
        System.out.println("Period: " + period);
    }

    public void testDuration() {
        LocalTime time1 = LocalTime.now();
```

```
        Duration twoHours = Duration.ofHours(2);

        LocalTime time2 = time1.plus(twoHours);
        Duration duration = Duration.between(time1, time2);

        System.out.println("Duration: " + duration);
```

We are asking end-user to give date of birth as input and calculating age based on date of birth.

To do work around with date we are using LocalDate from java.util.time package introduced in Java 8.

```
String dobDateString=scanner.next();

          StringTokenizer tokens=new
StringTokenizer(dobDateString,"/");

          List<String> tokensList=new ArrayList<>();
          while(tokens.hasMoreTokens()) {
              tokensList.add(tokens.nextToken());
          }

          int
dayOfMonth=Integer.parseInt(tokensList.get(0));
          int month=Integer.parseInt(tokensList.get(1));
          int year=Integer.parseInt(tokensList.get(2));
          LocalDate dateOfBirth=LocalDate.of(year, month-1,
dayOfMonth);
          LocalDate now=LocalDate.now();
          Period diff = Period.between(dateOfBirth, now);
          int internAge=diff.getYears();
```

try with resources was introduced in Java 7.

The Java *try with resources* construct, AKA *Java try-with-resources*, is an exception handling mechanism that can automatically close resources like a Java InputStream or a JDBC Connection when you are done with them.

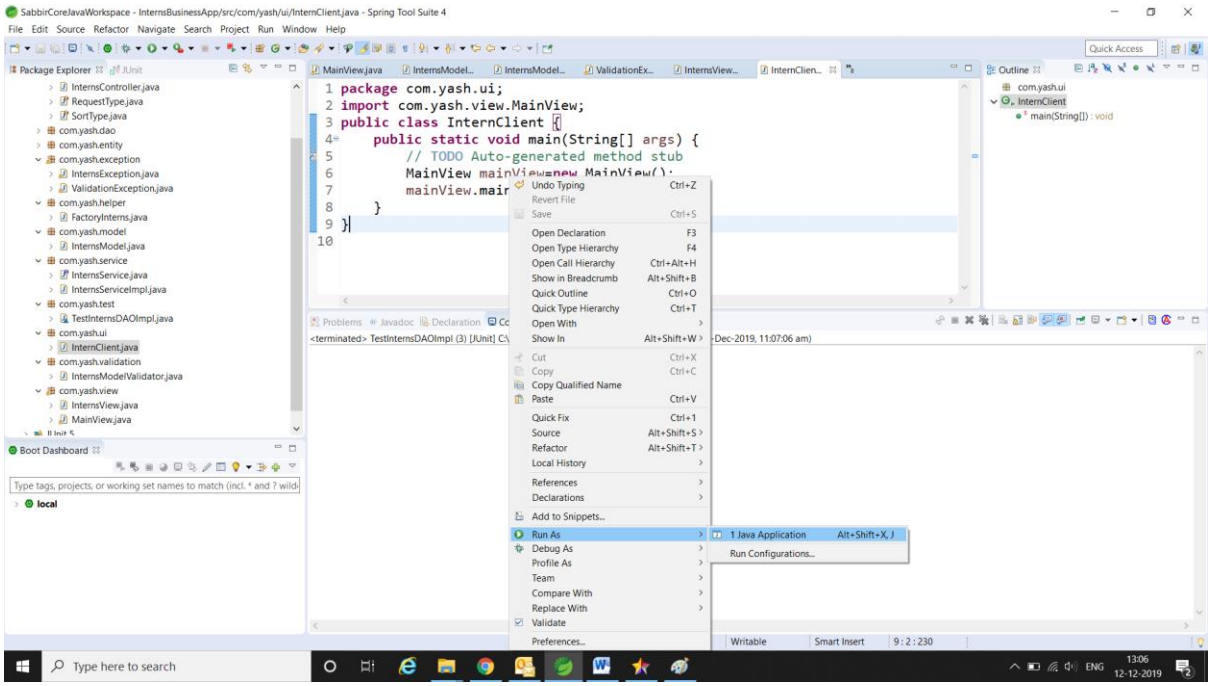
To do so, you must open and use the resource within a Java *try-with-resources* block. When the execution leaves the *try-with-resources* block, any resource opened within the *try-with-resources* block is automatically closed, regardless of whether any exceptions are thrown either from inside the *try-with-resources* block, or when attempting to close the resources.

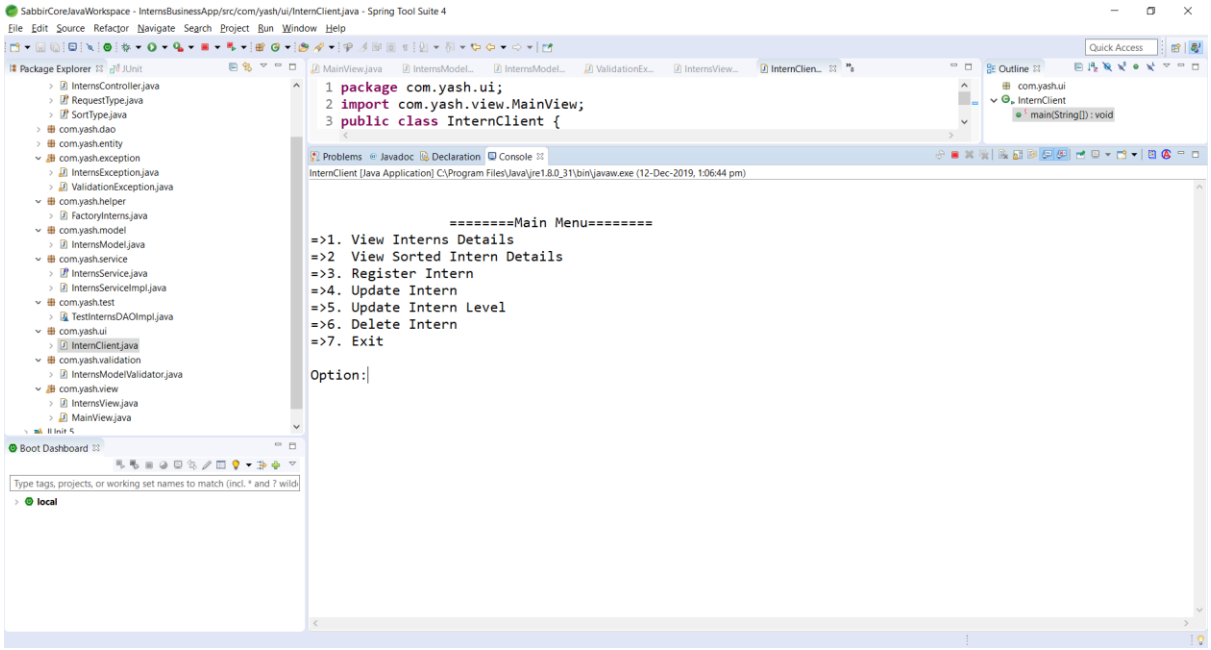
```
try(Scanner scanner=new Scanner(System.in);){
          InternsModelValidator validator=new
InternsModelValidator();
          int id=0;
          System.out.print("Id:");
          if(scanner.hasNextInt()) {
          id=scanner.nextInt();
          }
          else {
               try {
                    throw new
ValidationException("[!ERROR:Invalid Id]");
                    }catch(ValidationException e) {
                         System.out.println(e.getMessage());
                         mainMenu();
                    }
          }
```

Create a user interface class InternsClient in com.yash.ui package

```
package com.yash.ui;
import com.yash.view.MainView;
public class InternClient {
     public static void main(String[] args) {
          // TODO Auto-generated method stub
          MainView mainView=new MainView();
          mainView.mainMenu();
     }
}
```

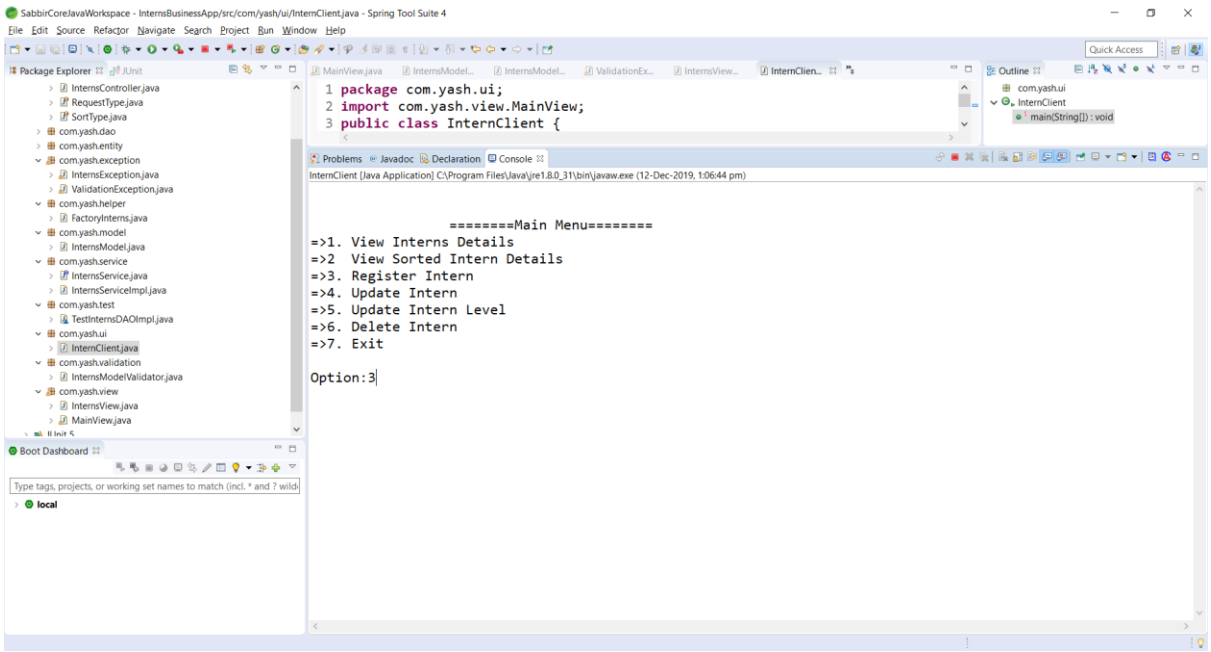
Run the above class to test the application,



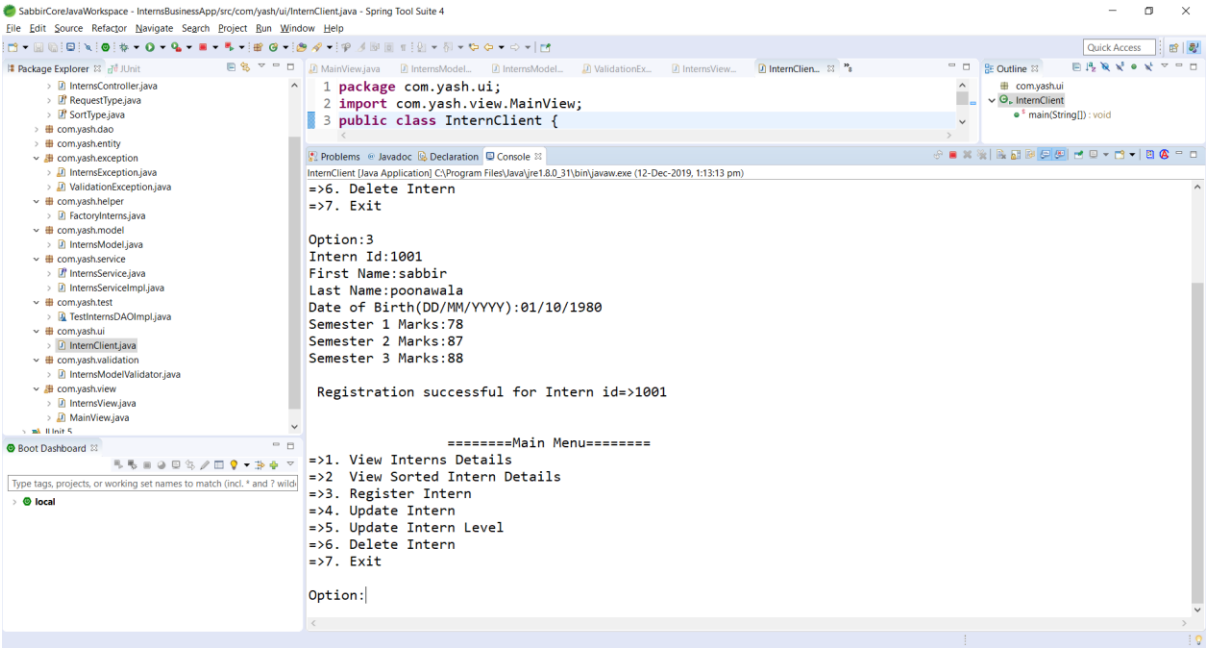


First we will have to register intern since initially map to store intern data is empty.

Enter option 3



Give input for Id, First Name, and Last Name etc.

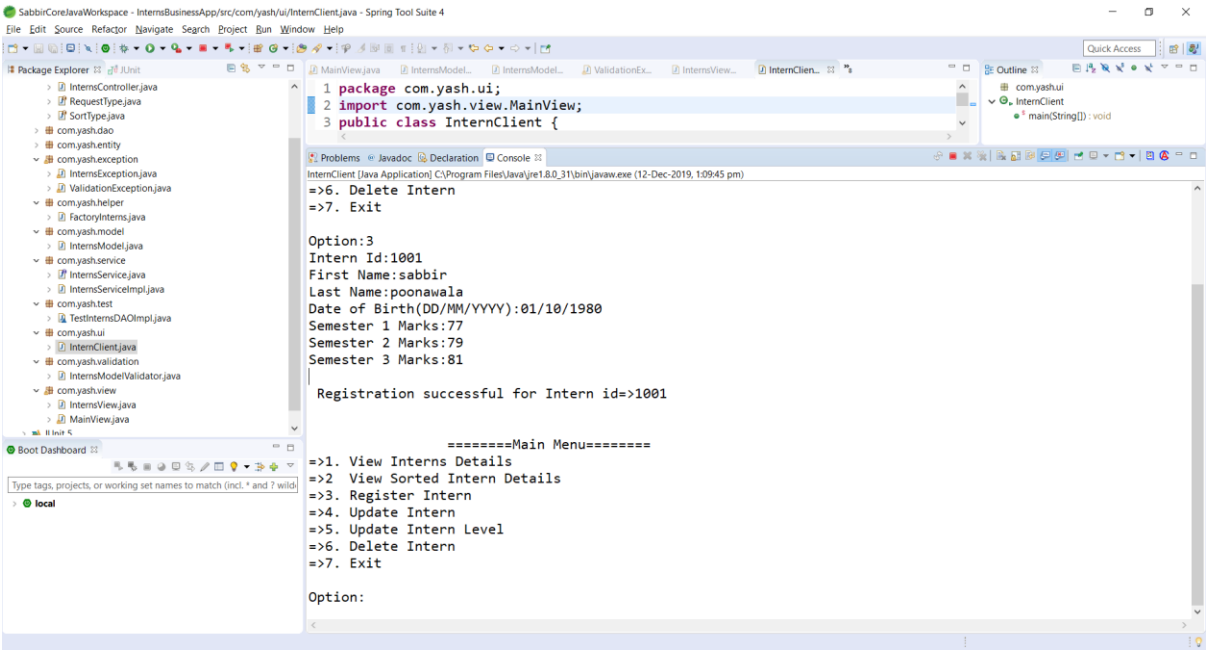


Note: first name and last name should be lower case

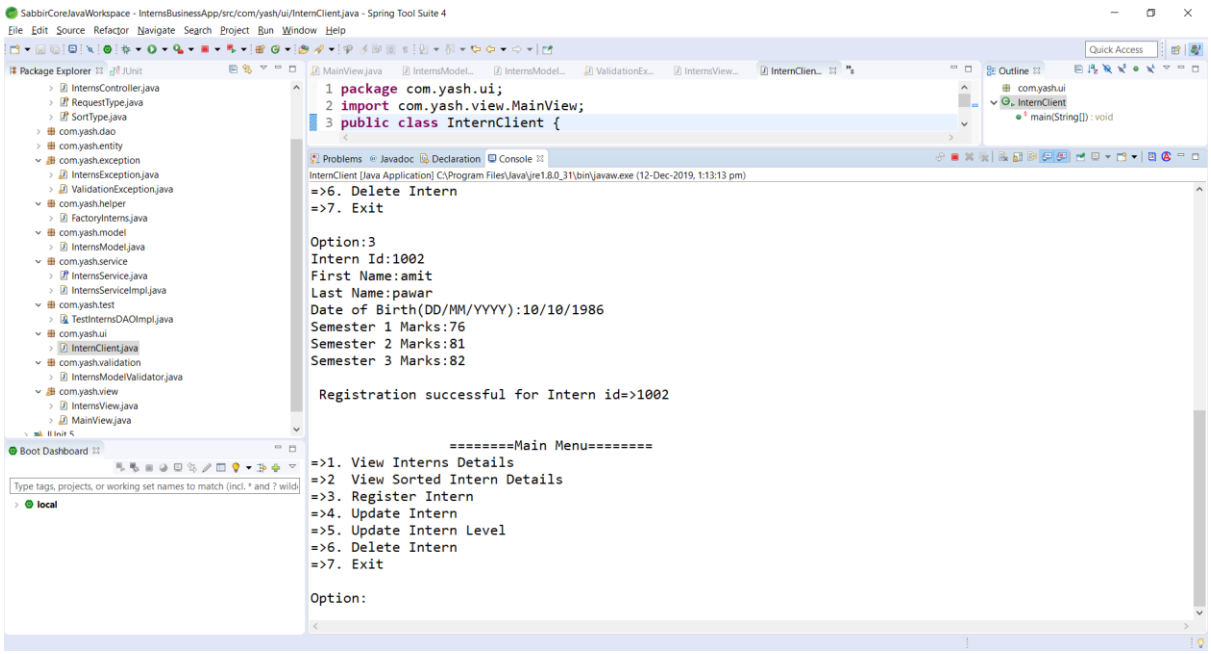
Id will not accept characters and first name and last name will not accept numbers.

Age should be greater than 18

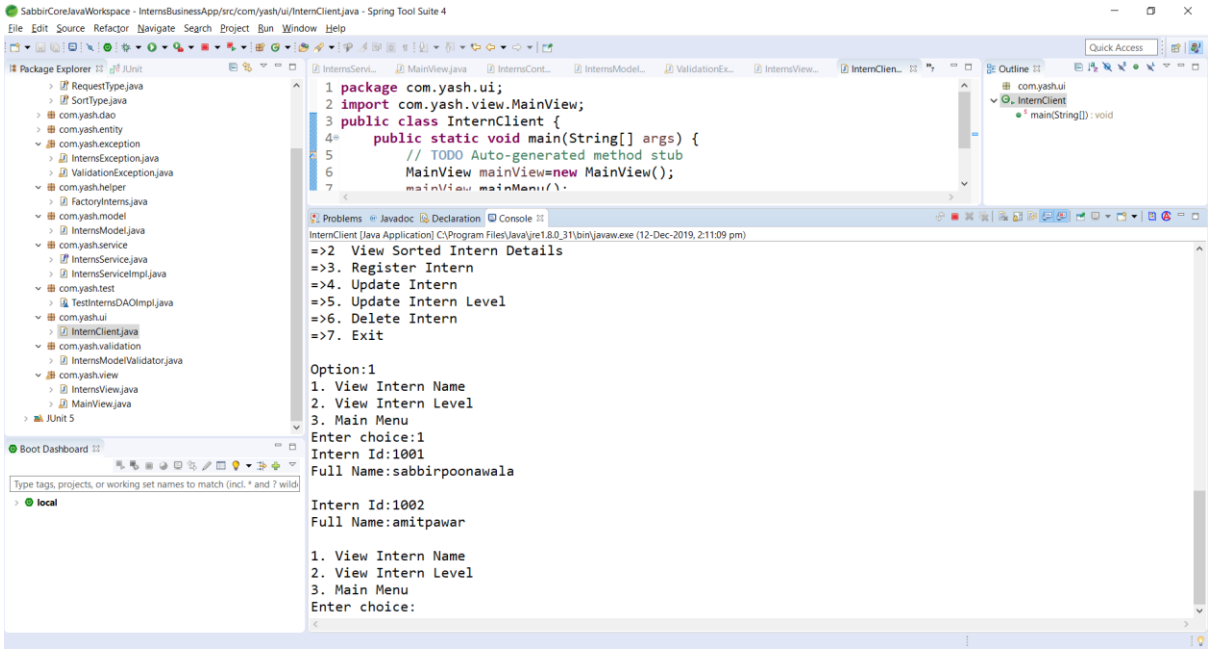
Average of semester should be greater than 50.

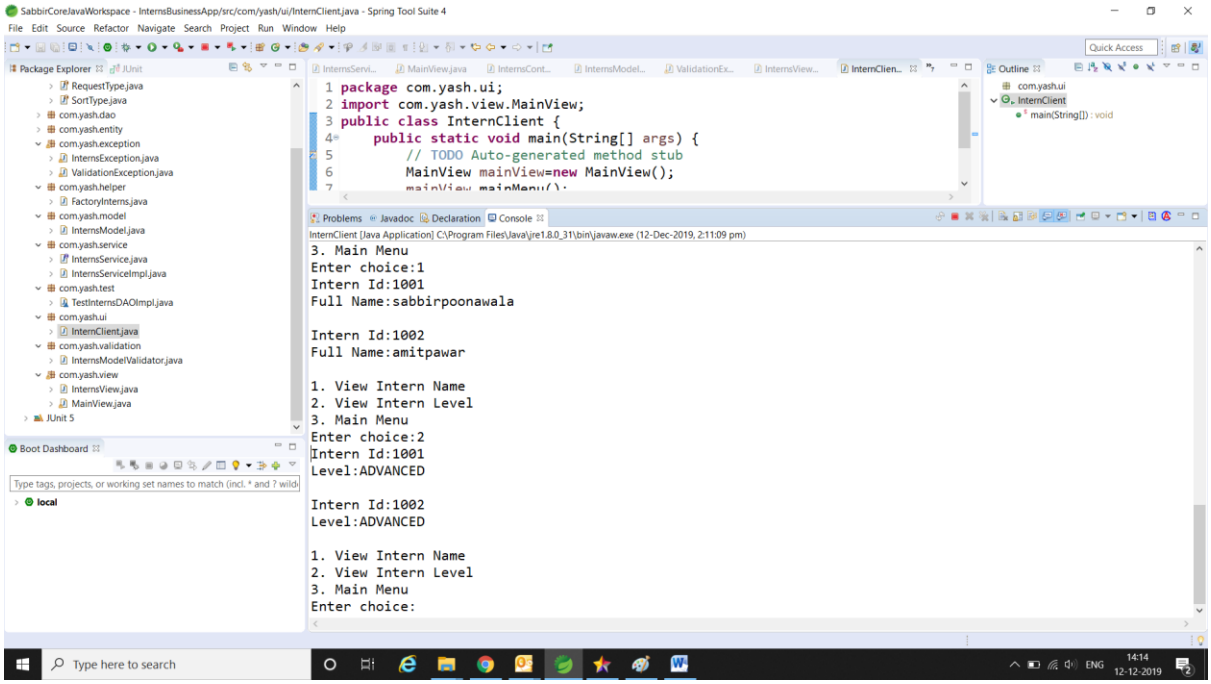


Add one more record,

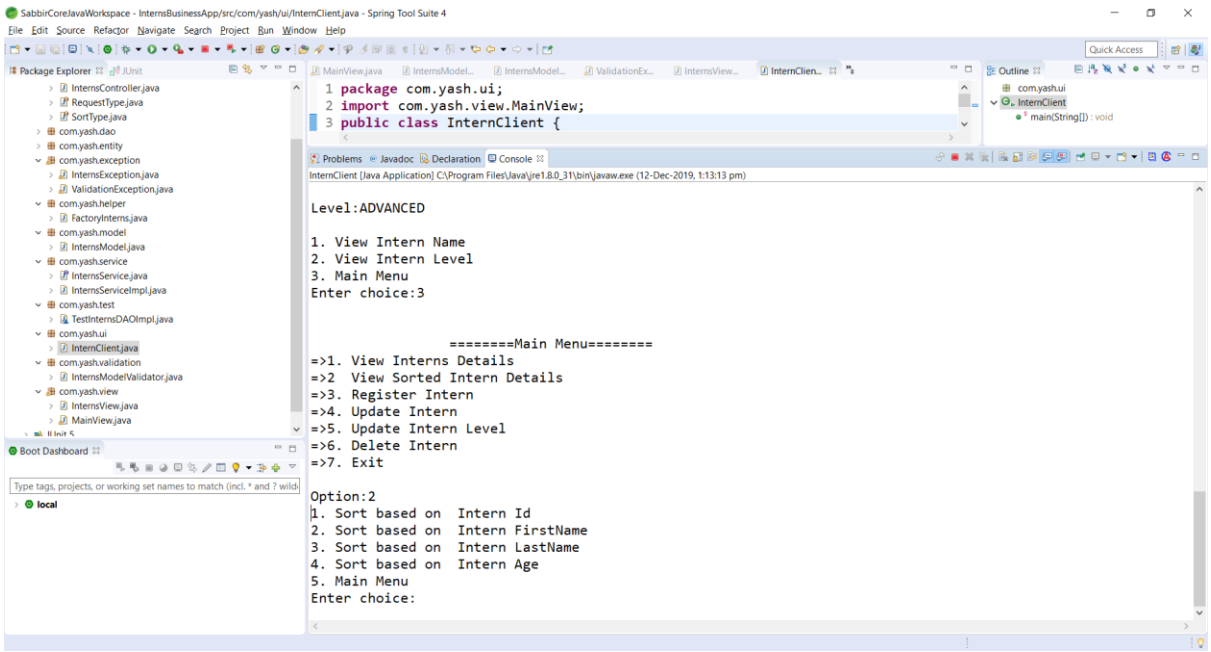


To view intern details,

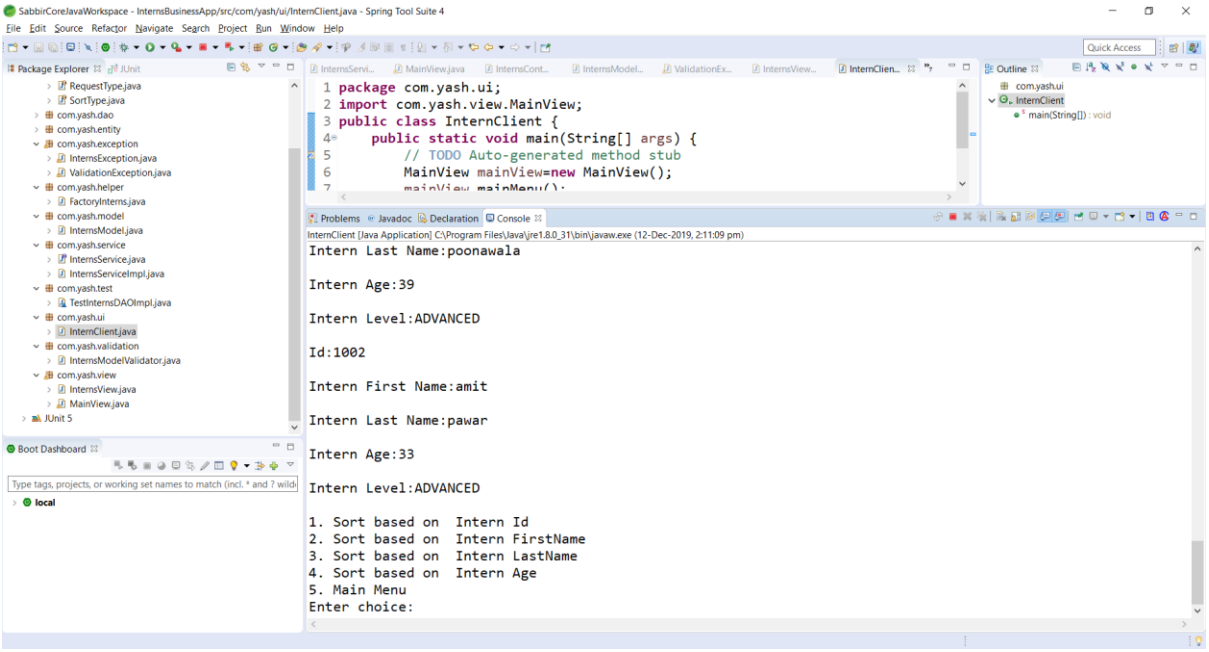




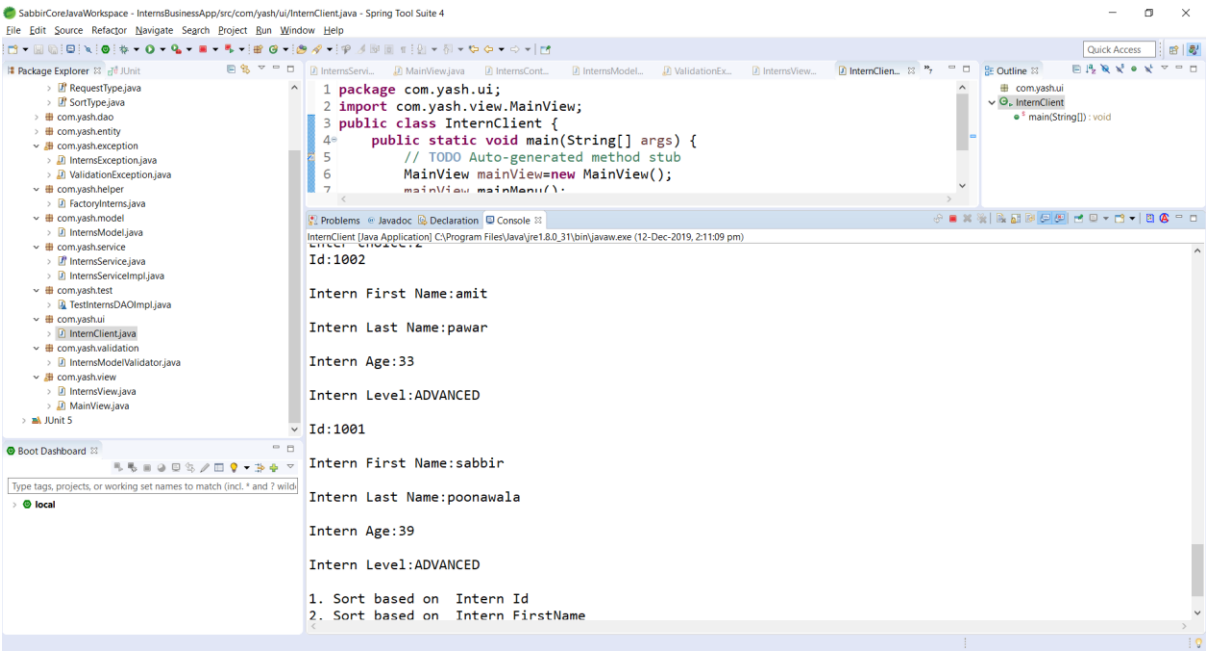
To sort intern details,



By id,

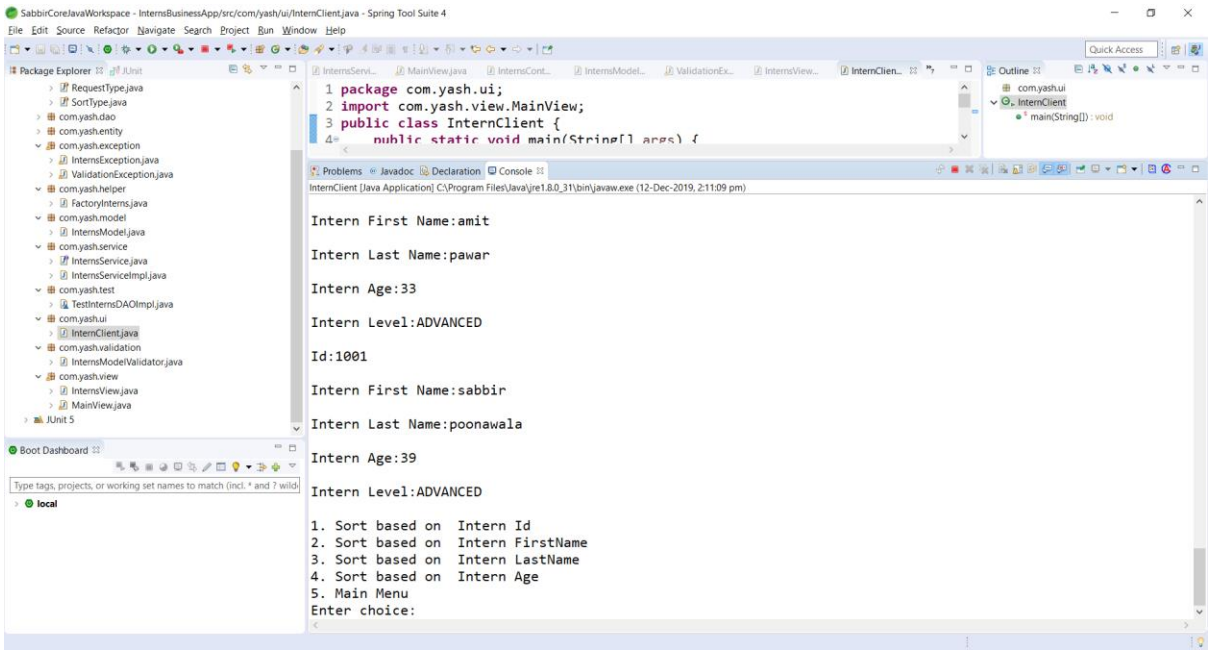


By first name,

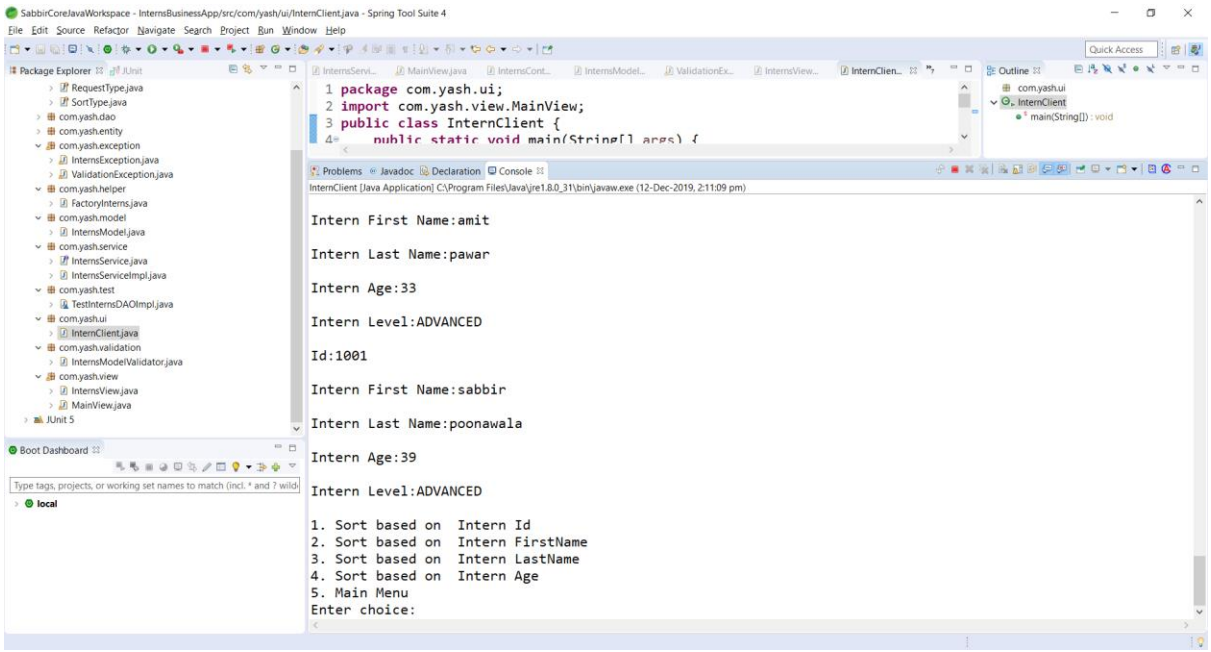


By last name,

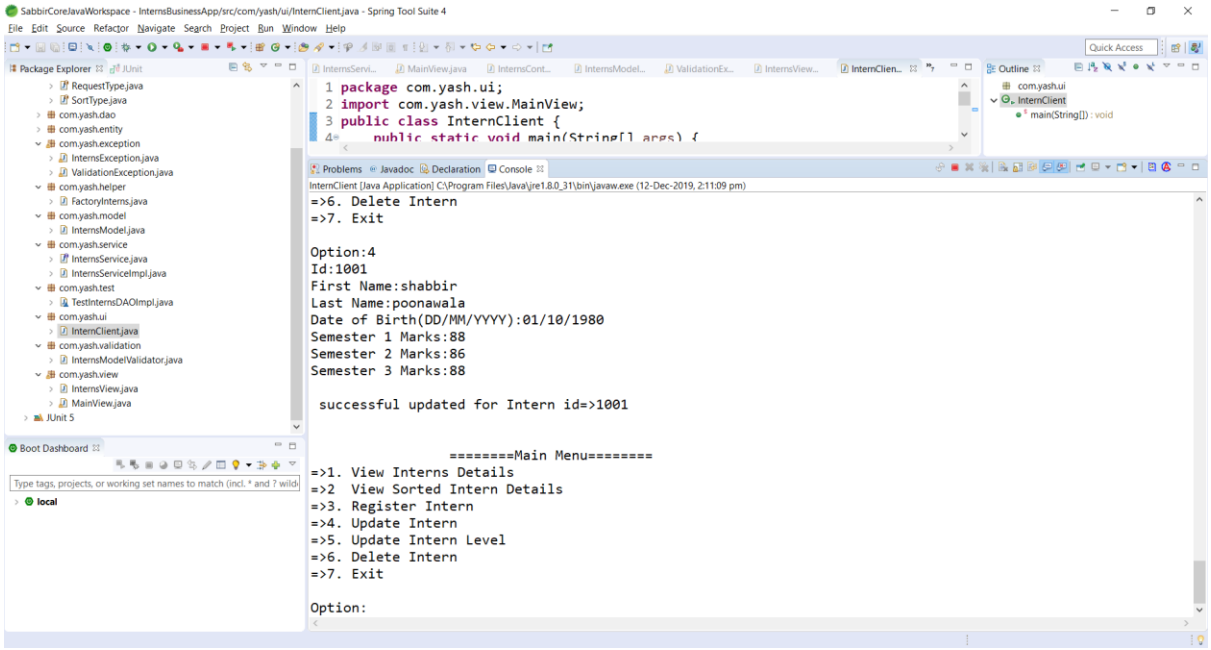
Since both last name starts with ‘p’ it will be sorted lexographically(i.e. sorted based on second character)



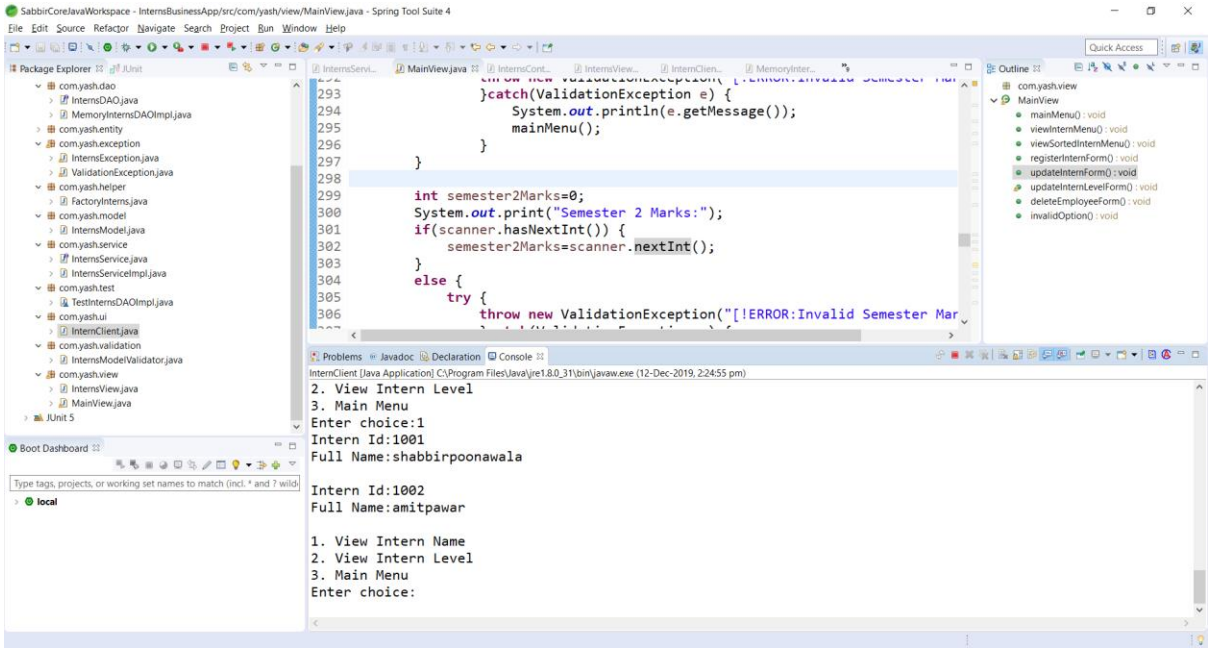
Sort by age,

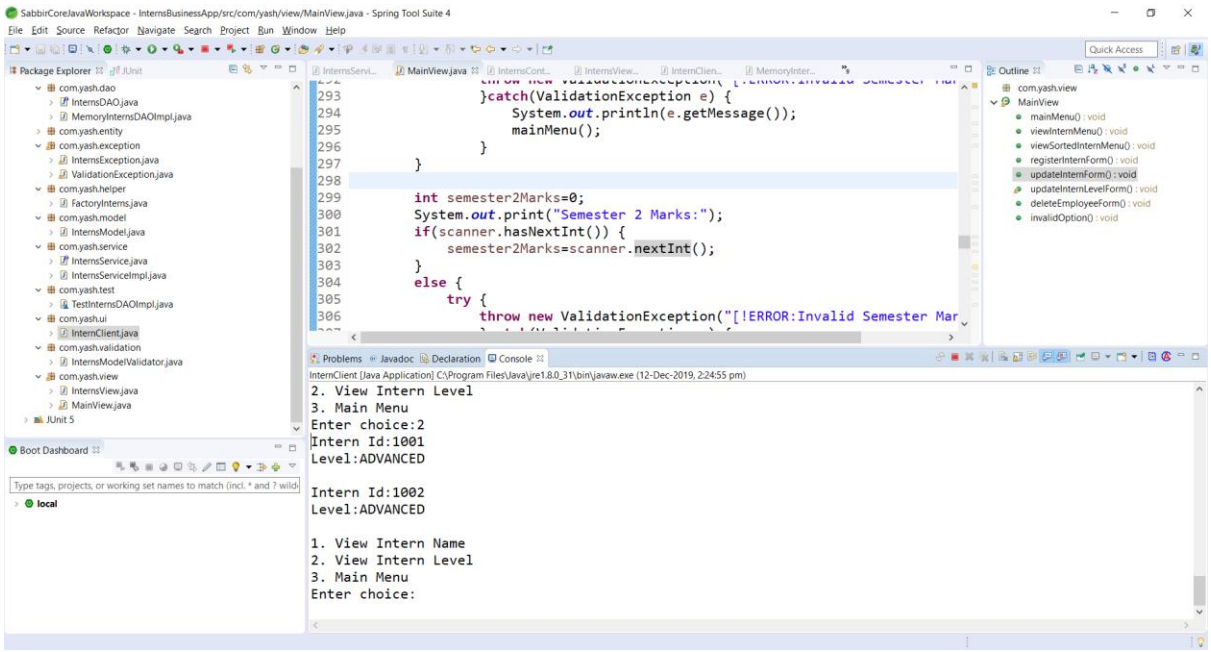


Update intern,

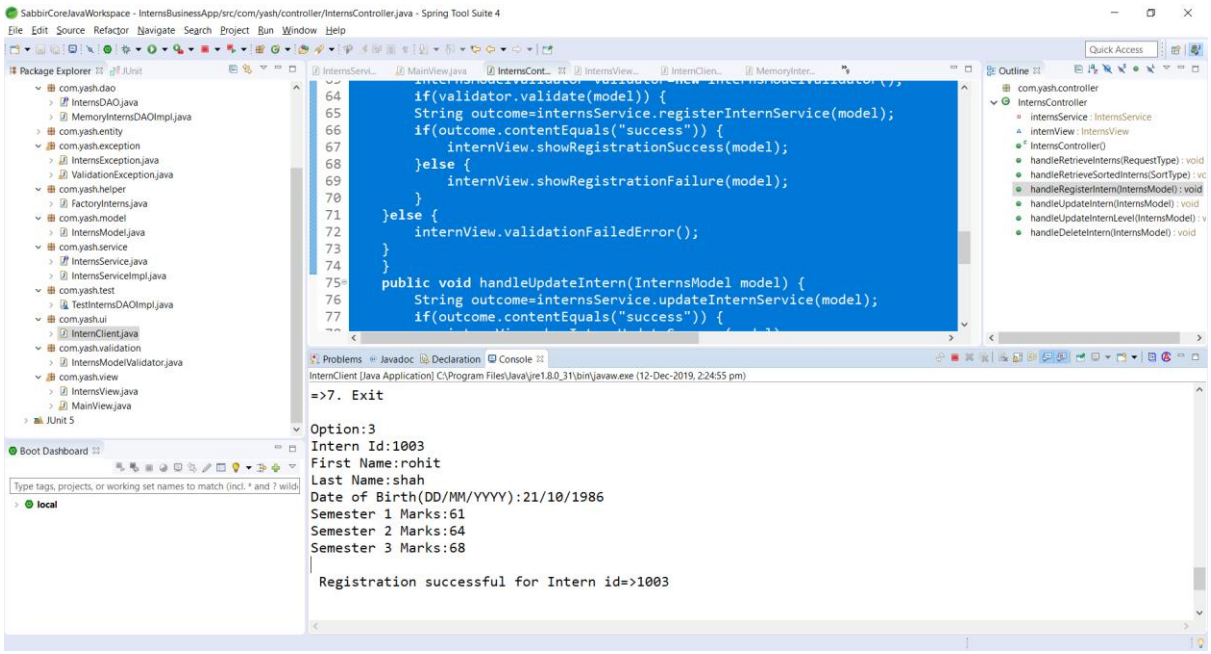


View updated data,

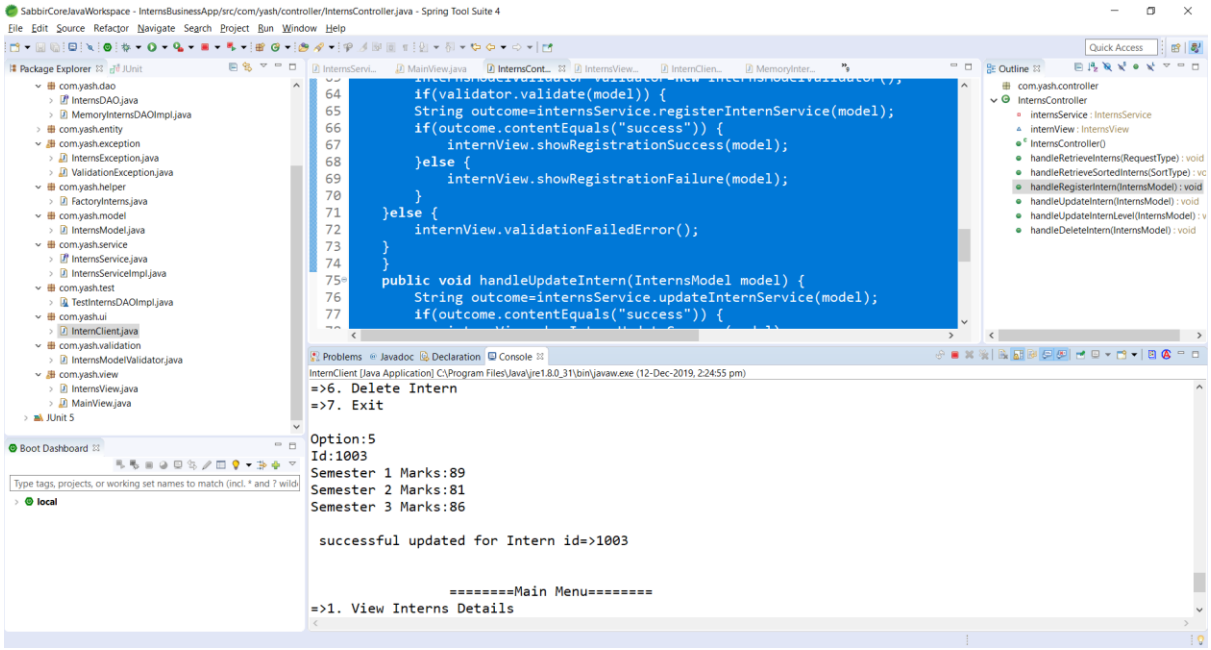




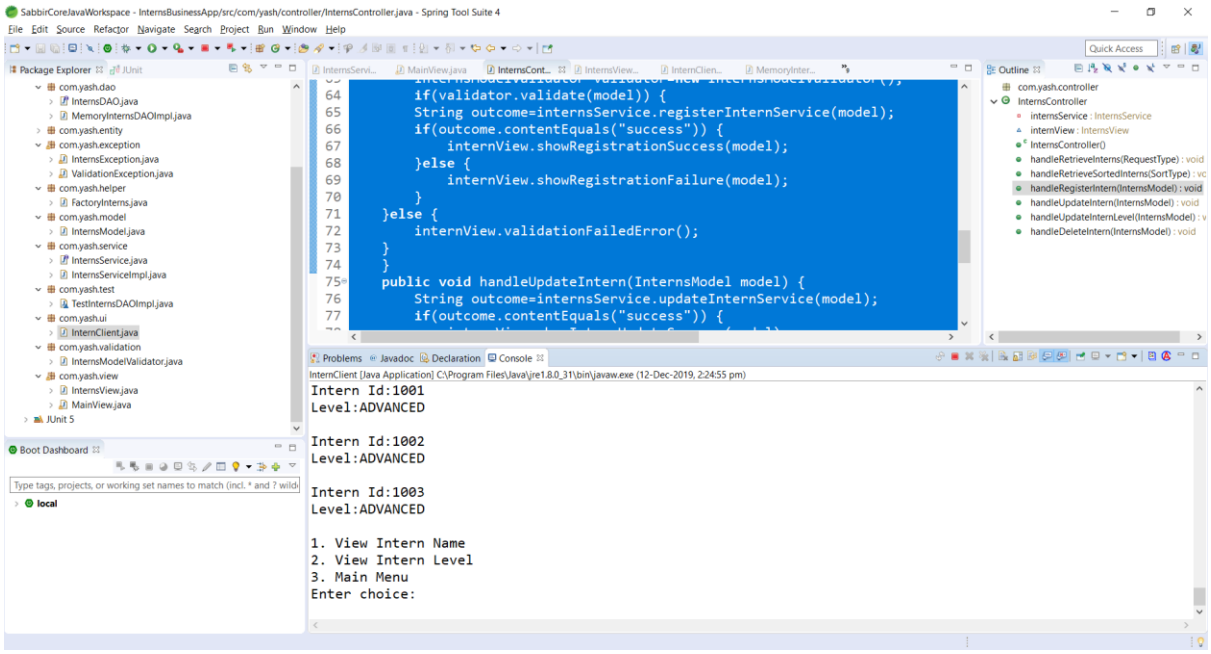
To update level, add new record with low semester marks.



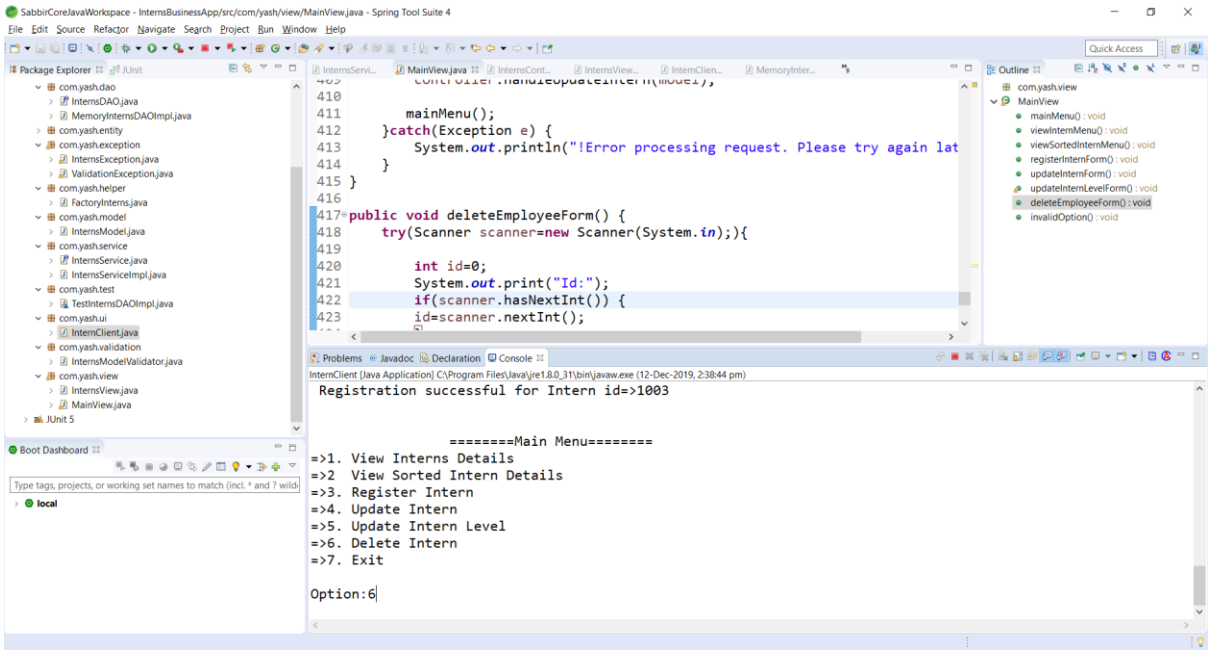
Now update level with higher semester marks.

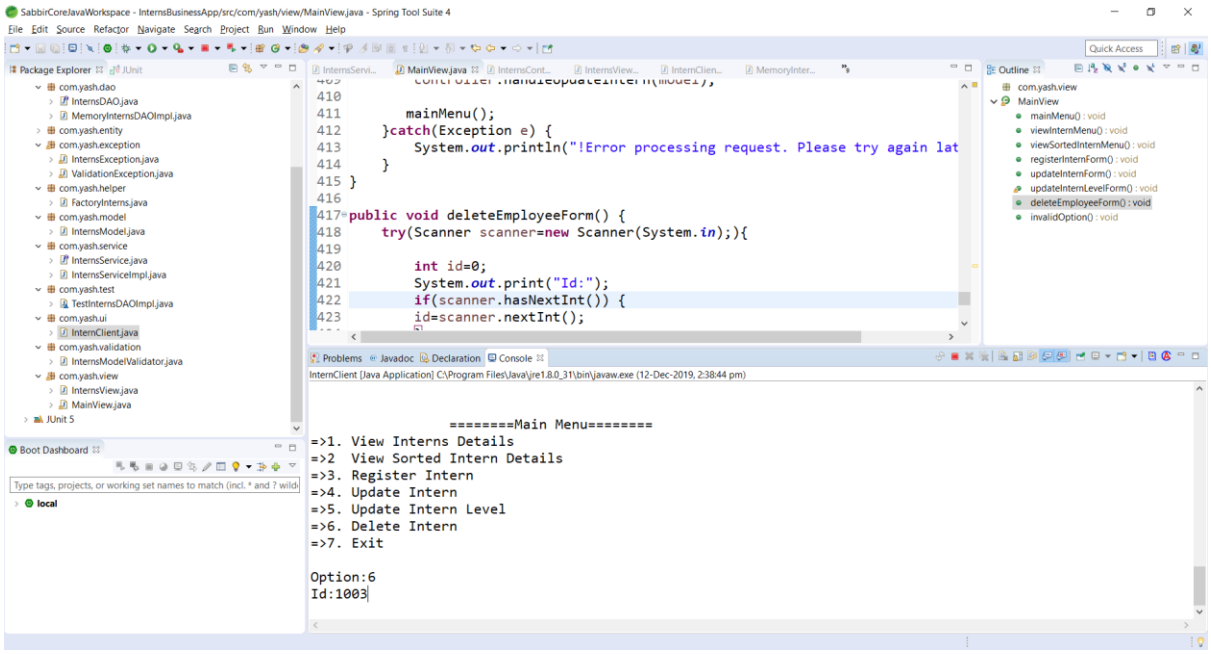


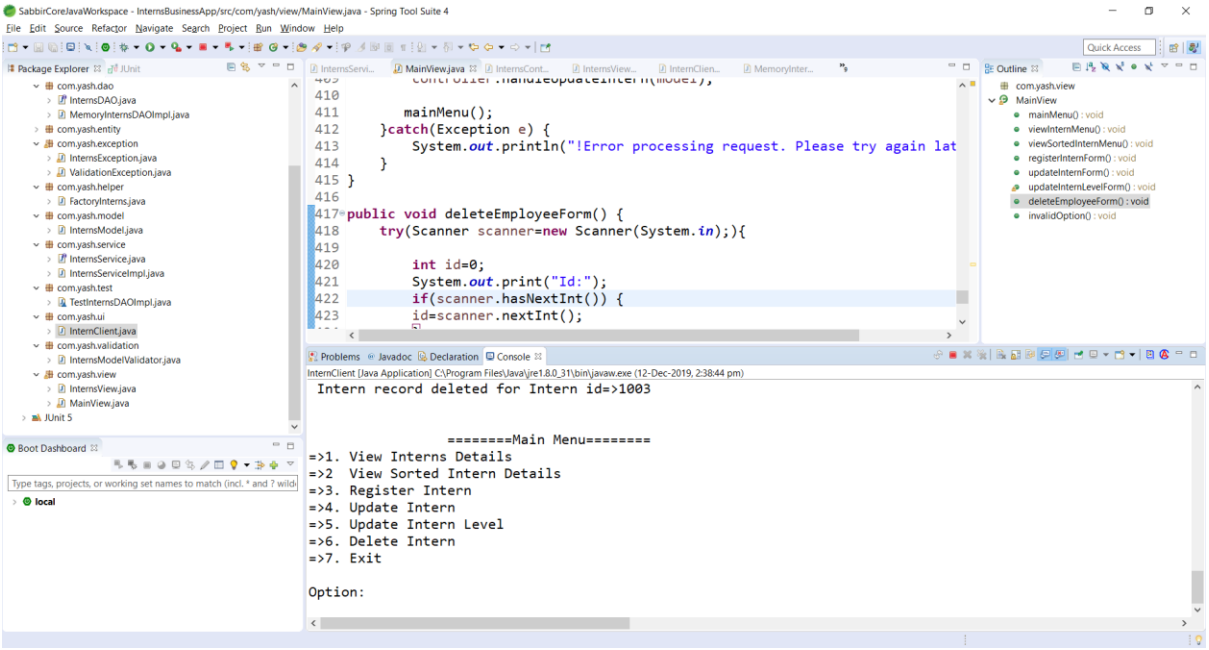
View intern updated level,

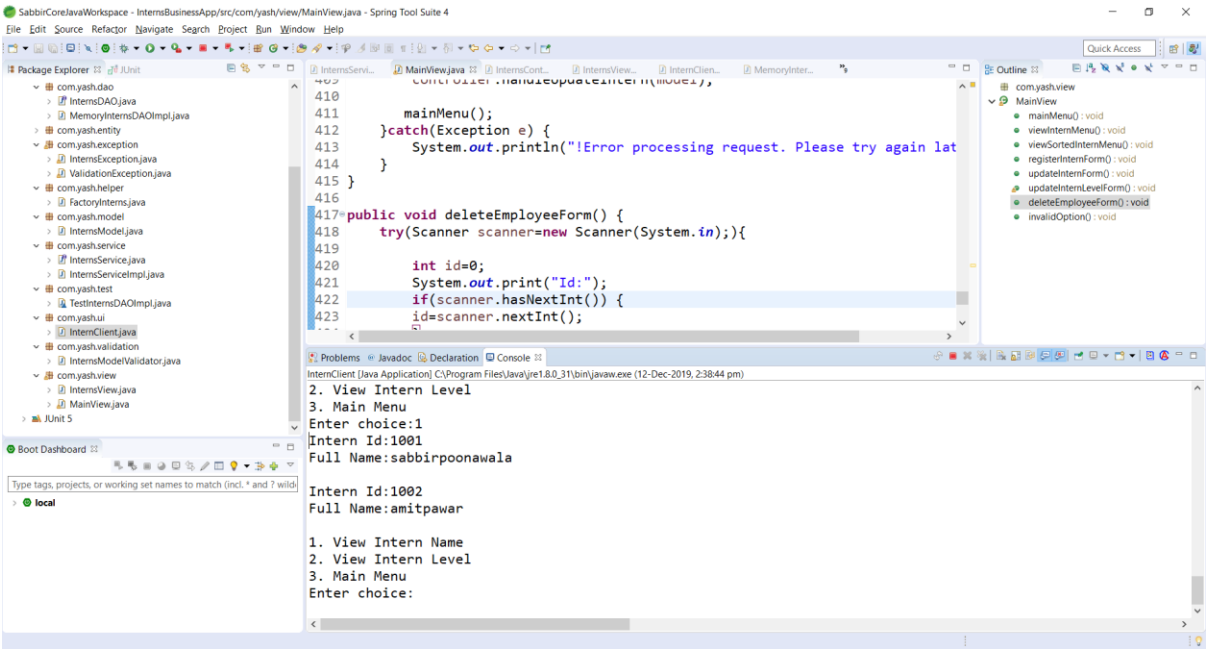


To remove intern,









We conclude in above application we have implemented MVC design pattern, collection framework and some of new features of Java 8.

In Web based MVC application, Controller class which is POJO (plain old java object) will become Servlet, View will be JSP and Model will be Java Bean.

In Spring MVC, Controller class will be annotated by @Controller, business service as @Service and DAO as @Repository, DispatcherServlet as FrontController. Spring IOC acts as Factory.

In Spring REST application, no rendering of data in business application, instead @RestController will give response as XML or JSON.

Base concept remains the same. Instead of we following design pattern and provide low level services we utilise framework which is by product of design patterns.

Happy Coding ☺

For any Queries, write to

sabbir.poonawala@yash.com