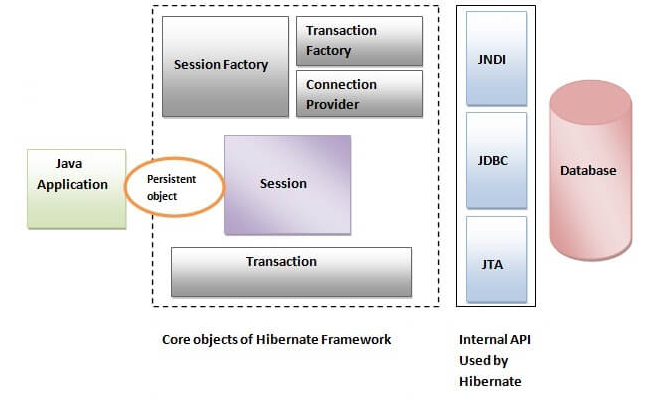
1. **Architecture of Hibernate**

The Hibernate architecture includes many objects such as persistent object, session factory, transaction factory, connection factory, session, transaction etc.

The Hibernate architecture is categorized in four layers.

* Java application layer
* Hibernate framework layer
* Backhand api layer
* Database layer



Elements of Hibernate Architecture

|  |
| --- |
| For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows: |

SessionFactory

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

Session

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

Transaction

The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

ConnectionProvider

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

TransactionFactory

It is a factory of Transaction. It is optional.

## What are the core interfaces of Hibernate framework?

1. **Session Interface:** The basic interface for all hibernate applications. The instances are light weighted and can be created and destroyed without expensive process.  
     
   **2. SessionFactory interface:** The delivery of session objects to hibernate applications is done by this interface. For the whole application, there will be generally one SessionFactory and can be shared by all the application threads.  
     
   **3. Configuration Interface:** Hibernate bootstrap action is configured by this interface. The location specification is specified by specific mapping documents, is done by the instance of this interface.  
     
   **4. Transaction Interface:** This is an optional interface. This interface is used to abstract the code from a transaction that is implemented such as a JDBC / JTA transaction.  
     
   **5. Query and Criteria interface:** The queries from the user are allowed by this interface apart from controlling the flow of the query execution.

## 3. What is ORM?

ORM stands for **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.

**An ORM system has the following advantages over plain JDBC −**

|  |  |
| --- | --- |
| **Sr.No.** | **Advantages** |
| 1 | Let’s business code access objects rather than DB tables. |
| 2 | Hides details of SQL queries from OO logic. |
| 3 | Based on JDBC 'under the hood.' |
| 4 | No need to deal with the database implementation. |
| 5 | Entities based on business concepts rather than database structure. |
| 6 | Transaction management and automatic key generation. |
| 7 | Fast development of application. |

An ORM solution consists of the following four entities −

|  |  |
| --- | --- |
| **Sr.No.** | **Solutions** |
| 1 | An API to perform basic CRUD operations on objects of persistent classes. |
| 2 | A language or API to specify queries that refer to classes and properties of classes. |
| 3 | A configurable facility for specifying mapping metadata. |
| 4 | A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions. |

**5. SQL query syntax in hibernate.**

For Hibernate Native SQL Query, we use Session.createSQLQuery(String query) to create the SQLQuery object and execute it.

SQLQuery query = session.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee");

@Query(value=”SELECT \* FROM……………………………” nativeQuery=true)

6. What is design Pattern?

<https://www.digitalocean.com/community/tutorials/spring-bean-scopes>

<https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns>

Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

|  |  |
| --- | --- |
| **S.N.** | **Pattern & Description** |
| 1 | **Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case. |
| 2 | **Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities. |
| 3 | **Behavioural Patterns** These design patterns are specifically concerned with communication between objects. |
| 4 | **J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Centre. |

**Java Singleton Pattern**

**Java Singleton Pattern** is one of the **Gangs of Four Design patterns** and comes in the **Creational Design Pattern** category. From the definition, it seems to be a very simple design pattern but when it comes to implementation, it comes with a lot of implementation concerns. The implementation of Java Singleton pattern has always been a controversial topic among developers. Here we will learn about Singleton design pattern principles, different ways to implement the Singleton design pattern and some of the best practices for its usage.

## Singleton Pattern

**Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine**.

The singleton class must provide **a global access point** to get the **instance of the class**.

Singleton pattern is used **for**[**logging**](https://www.digitalocean.com/community/tutorials/logger-in-java-logging-example)**, driver’s objects, caching and**[**thread pool**](https://www.digitalocean.com/community/tutorials/threadpoolexecutor-java-thread-pool-example-executorservice).

Singleton design pattern is also used in other design patterns like [Abstract Factory](https://www.digitalocean.com/community/tutorials/abstract-factory-design-pattern-in-java), [Builder](https://www.digitalocean.com/community/tutorials/builder-design-pattern-in-java), [Prototype](https://www.digitalocean.com/community/tutorials/prototype-design-pattern-in-java), [Facade](https://www.digitalocean.com/community/tutorials/facade-design-pattern-in-java) etc.

Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

When should we use Singleton design pattern?

A singleton should be used **when managing access to a resource which is shared by the entire application**, and it would be destructive to potentially have multiple instances of the same class. Making sure that access to shared resources thread safe is one very good example of where this kind of pattern can be vital

Some approach for getting a singleton is to make the constructor private and write a own static method, that creates a new element on the first call, saves it in a static variable and always returns this object when called again.

***Spring Bean Scopes***

There are five types of spring bean scopes:

singleton - only one instance of the spring bean will be created for the spring container. This is the default spring bean scope. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.

prototype – A new instance will be created every time the bean is requested from the spring container.

request – This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.

session – A new bean will be created for each HTTP session by the container.

global-session – This is used to create global session beans for Portlet applications.

Spring Bean Singleton and Prototype Scope

Spring bean singleton and prototype scopes can be used in standalone spring apps. Let’s see how we can easily configure these scopes using @Scope annotation. Let’s say we have a java bean class.

package com.journaldev.spring;

public class MyBean {

public MyBean() {

System.out.println("MyBean instance created");

}

}

Let’s define the spring configuration class where we will define the method to get MyBean instance from spring container.

package com.journaldev.spring;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Scope;

@Configuration

public class MyConfiguration {

@Bean

@Scope(value="singleton")

public MyBean myBean() {

return new MyBean();

}

}

Note that singleton is default scope, so we can remove @Scope(value="singleton") from above bean definition. Now let’s create a main method and test the singleton scope.

package com.journaldev.spring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MySpringApp {

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(MyConfiguration.class);

ctx.refresh();

MyBean mb1 = ctx.getBean(MyBean.class);

System.out.println(mb1.hashCode());

MyBean mb2 = ctx.getBean(MyBean.class);

System.out.println(mb2.hashCode());

ctx.close();

}

}

When above program is executed, we will get output like below.

MyBean instance created

867988177

867988177

Notice that both MyBean instances have same hashcode and the constructor is called once once, it means that spring container is returning the same instance of MyBean always. Now let’s change the scope to prototype.

@Bean

@Scope(value="prototype")

public MyBean myBean() {

return new MyBean();

}

This time we will get following output when main method is executed.

MyBean instance created

867988177

MyBean instance created

443934570

## Java Singleton Pattern Implementation

To implement a Singleton pattern, we have different approaches but all of them have the following common concepts.

**Private constructor** to **restrict** **instantiation** of the class from other classes.

**Private static variable** of the same class that is the **only instance of the class.**

**Public static method** that returns the **instance of the class**, this is the global access point for outer world to get the instance of the singleton class.

Can singleton have multiple instances?

No………

This is the simple example for Singleton class in java. By calling Singleton.getInstance() you can get the instance of this Singleton class. **Here instance is private static and constructor is private so only one object is available per JVM.**

public class Singleton {

private static Singleton instance = new Singleton ();

private Singleton () {

}

public static Singleton getInstance() {

return instance;

}

}

**Lazy instantiation to create the singleton also known as classic singleton** and it is not a thread safe version

public class Singleton {

private static Singleton instance = null;

private Singleton () {

}

public static Singleton getInstance() {

If (instance==null) {

instance=new Singleton();

}

return instance;

}

}

A thread safe lazy initialization singleton class can be implemented in a simple way as follows

public class Singleton {

private Singleton () {}

private static class LazyHolder {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance () {

return LazyHolder.INSTANCE;

}

}

What is clone? Do singleton classes get cloned? If so, how to prevent cloning.

**What is Object Cloning?**

The object cloning is a way to create exact copy of an object. So if somebody will clone our singleton instance, it will create another copy of the Singleton instance which violates principle of Singleton Design Pattern.

**How does Java support Cloning?**

In java, clone() method of Object class is used for cloning.  clone() is protected method and as every class in java by default extends Object class, object of any class including our Singleton class can call clone() method.

**If somebody will call instance.clone() method, will it create copy of our Singleton class?**

No. The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don’t implement Cloneable interface, clone() method generates CloneNotSupportedException.

***So, if we are not implementing Cloneable interface in our Singleton class, then we do not require to prevent cloning. Java will do it for us.***

**Example:**

class Main implements Cloneable {

// declare variables

String name;

int version;

public static void main(String[] args) {

// create an object of Main class

Main obj1 = new Main();

// initialize name and version using obj1

obj1.name = "Java";

obj1.version = 14;

// print variable

System.out.println(obj1.name); // Java

System.out.println(obj1.version); // 14

try {

// create clone of obj1

Main obj2 = (Main)obj1.clone();

// print the variables using obj2

System.out.println(obj2.name); // Java

System.out.println(obj2.version); // 14

}

catch (Exception e) {

System.out.println(e);

}

}

}

**What if our Singleton class is inherriting properties of some class which has implemented cloneable interface?**

**For example,**

class CloneableParent implements Cloneable {

}

class SingletonChild extends CloneableParent {

    private static SingletonChild instance = new SingletonChild();

    private SingletonChild() {

    }

    public static SingletonChild getInstance() {

        return instance;

    }

    // clone() method code here

}

Well, in this case we need to override clone() method inside SingletonChild and throw CloneNotSupportedException explicitly as shown below:

@Override

protected Object clone() throws CloneNotSupportedException {

    throw new CloneNotSupportedException();

}

**How to create Immutable class in Java?**

Immutable class in java means that once an object is created, we cannot change its content. In Java, all the [**wrapper classes**](https://www.geeksforgeeks.org/wrapper-classes-java/) (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well. Prior to going ahead do go through characteristics of immutability in order to have a good understanding while implementing the same. Following are the requirements:

* The class must be declared as **final** so **that child classes can’t be created**.
* **Data members** in the class must be declared **private** so that direct access is not allowed.
* Data members in the class must be declared as **final** so that we can’t change the value of it after object creation.
* A **parameterized constructor** should **initialize** all the **fields** performing a **deep copy** so that data members **can’t be modified with an object reference**.
* **Deep Copy of objects** should be performed in **the getter methods** to return a copy rather than returning the **actual object reference**)

***Note:****There should* ***be no setters*** *or in simpler terms, there should* ***be no option to change the value of the instance variable.***

# **Difference between intermediate and terminal operations in Java 8**

Stream is introduced in Java 8, it is only used for **processing group of data** not for the storting elements.. It does not modify the actual collection, they only **provide the result** as per the pipelined methods.

Stream api supports multiple operations and operations are divided into two parts −

* **Intermediate Operation**- These operations are used to pipeline other methods and to transform into the other streams. They don’t produce results because these operation does not invoke until the terminal operation gets executed. Below are the examples −
* **sorted(Comparator<T>)**
* **peek(Consumer<T>)**
* **distinct()**
* **Terminal operations -** These operations are used to produce results. They can’t be used for chaining the other methods. Below are the examples −
* **forEach**
* **count**
* **toArray**

| **Sr. No.** | **Key** | **Intermediate Operations** | **Terminal Operations** |
| --- | --- | --- | --- |
| 1 | Basic | These operations are used to pipeline other methods and to transform into the other streams | A terminal operation in Java is a method applied to a stream as the final step. |
| 2 | Return Type | They only return another stream. | They return final result. |
| 3 | Method | sorted(Comparator<T>)peek(Consumer<T>)distinct() | forEachcounttoArray |
| 4. | Use Case | These operations should be used to transform stream into another stream | They can be used to produce results. |

**How to remove duplicates from List in java?**

**Using LinkedHashSet**

A better way (both time complexity and ease of implementation wise) is to remove duplicates from an ArrayList is to convert it into a Set that does not allow duplicates. Hence LinkedHashSet is the best option available as this do not allows duplicates as well it preserves the insertion order.

// Java program to remove duplicates from ArrayList

import java.util.\*;

public class GFG {

// Function to remove duplicates from an ArrayList

public static <T> ArrayList<T> removeDuplicates(ArrayList<T> list)

{

// Create a new LinkedHashSet

Set<T> set = new LinkedHashSet<>();

// Add the elements to set

set.addAll(list);

// Clear the list

list.clear();

// add the elements of set

// with no duplicates to the list

list.addAll(set);

// return the list

return list;

}

// Driver code

public static void main(String args[])

{

// Get the ArrayList with duplicate values

ArrayList<Integer>

list = new ArrayList<>(

Arrays.asList(1, 10, 1, 2, 2, 3, 10, 3, 3, 4, 5, 5));

// Print the Arraylist

System.out.println("ArrayList with duplicates: "+ list);

// Remove duplicates

ArrayList<Integer>

newList = removeDuplicates(list);

// Print the ArrayList with duplicates removed

System.out.println("ArrayList with duplicates removed: "+ newList);

}

}

**Using Java 8 Stream.distinct()**

You can use the distinct() method from the Stream API. The distinct() method return a new Stream without duplicates elements based on the result returned by equals() method, which can be used for further processing. The actual processing of Stream pipeline starts only after calling terminal methods like forEach() or collect().

// Java program to remove duplicates from ArrayList

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

// Program to remove duplicates from a List in Java 8

class GFG

{

public static void main(String[] args)

{

// input list with duplicates

List<Integer> list = new ArrayList<>(

Arrays.asList(1, 10, 1, 2, 2, 3, 10, 3, 3, 4, 5, 5));

// Print the Arraylist

System.out.println("ArrayList with duplicates: "+ list);

// Construct a new list from the set constucted from elements

// of the original list

List<Integer> newList = list.stream().distinct().collect(Collectors.toList());

// Print the ArrayList with duplicates removed

System.out.println("ArrayList with duplicates removed: "+ newList);

}

}

**String Buffer**

**StringBuffer**is a peer class of **String**that provides much of the functionality of strings. The **string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.** StringBuffermay have characters and substrings inserted in the middle or appended to the end. Itwill automatically grow to make room for such additions and often has more characters pre allocated than are actually needed, to allow room for growth.

**Some Interesting Facts about the StringBuffer class**

Do keep the following points in the back of your mind: 

* java.lang.StringBuffer extends (or inherits from) [Object class](https://www.geeksforgeeks.org/object-class-in-java/).
* All Implemented Interfaces of StringBuffer class: **Serializable**, **Appendable**, **CharSequence**.
* public final class StringBuffer extends Object implements Serializable, CharSequence, Appendable.
* **String buffers** **are safe for use by multiple threads**. The methods can be **synchronized** wherever necessary so that all the operations on any particular instance behave as if they occur in some serial order.
* Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence) this class synchronizes only on the string buffer performing the operation, not on the source.
* It inherits some of the methods from the Object class which such as ***clone****(),* ***equals****(),* ***finalize****(),* ***getClass****(),* ***hashCode****(),***notifies**()*,* ***notifyAll****().*

## String vs StringBuffer vs StringBuilder

String is immutable whereas StringBuffer and StringBuilder are mutable classes.

StringBuffer is thread-safe and synchronized whereas StringBuilder is not. That’s why StringBuilder is faster than StringBuffer.

**String concatenation operator (+) internally uses StringBuffer or StringBuilder class**.

For String manipulations in a **non-multi threaded** environment, we should **use StringBuilder** else use StringBuffer class.

**Use of String Buffer**

For simple concatenations like:

String s = "a" + "b" + "c";

It is rather pointless to use StringBuffer - as *jodonnell* pointed out it will be smartly translated into:

String s = new StringBuffer().append("a").append("b").append("c").toString();

**BUT** it is very unperformant to concatenate strings in a loop, like:

String s = "";

for (int i = 0; i < 10; i++) {

s = s + Integer.toString(i);

}

Using string in this loop will generate 10 intermediate string objects in memory: "0", "01", "012" and so on. While writing the same using StringBuffer you simply update some internal buffer of StringBuffer and you do not create those intermediate string objects that you do not need:

StringBuffer sb = new StringBuffer();

for (int i = 0; i < 10; i++) {

sb.append(i);

}

Actually for the example above you should use StringBuilder (introduced in Java 1.5) instead of **StringBuffer** - StringBuffer is little heavier as all its methods are synchronized.

# **Hashtable in Java**

The **Hashtable** class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the **hashCode** method and the **equals** method.

**Features of Hashtable**

* It is similar to HashMap, but is **synchronized**.
* Hashtable stores key/value pair in hash table.
* In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The **key** is then **hashed**, and the resulting hash code is used as the **index** at which the value is stored within the table.
* The initial default capacity of Hashtable class is **11** whereas loadFactor is **0.75**.
* HashMap doesn’t provide any **Enumeration**, while Hashtable provides not fail-fast Enumeration.

# Fail Fast and Fail Safe Iterators in Java

 how those collections behave which doesn’t iterate as fail-fast. First of all, there is no term as fail-safe given in many places as Java SE specifications does not use this term. I am using fail safe to segregate between Fail fast and Non fail-fast iterators.  
**Concurrent Modification:** Concurrent Modification in programming means to modify an object concurrently when another task is already running over it. For example, in Java to modify a collection when another thread is iterating over it. Some Iterator implementations (including those of all the general purpose collection implementations provided by the JRE) may choose to throw *ConcurrentModificationException* if this behavior is detected.

**Fail Fast And Fail Safe Iterators in Java**

[Iterators](https://contribute.geeksforgeeks.org/iterators-in-java/) in java are used to iterate over the Collection objects.Fail-Fast iterators immediately throw *ConcurrentModificationException* if there is **structural modification** of the collection. Structural modification means adding, removing any element from collection while a thread is iterating over that collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.  
Fail-Safe iterators don’t throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that’s why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.

**How Fail Fast Iterator works ?**

To know whether the collection is structurally modified or not, fail-fast iterators use an internal flag called *modCount* which is updated each time a collection is modified.Fail-fast iterators checks the *modCount* flag whenever it gets the next value (i.e. using *next()* method), and if it finds that the *modCount* has been modified after this iterator has been created, it throws *ConcurrentModificationException*.

**The following table explains the difference between the extends and interface:**

|  |  |  |
| --- | --- | --- |
| S.No. | Extends | Implements |
| 1. | By using “extends” keyword a class can inherit another class, or an interface can inherit other interfaces | By using “implements” keyword a class can implement an interface |
| 2. | It is not compulsory that subclass that extends a superclass override all the methods in a superclass. | It is compulsory that class implementing an interface has to implement all the methods of that interface. |
| 3. | Only one superclass can be extended by a class. | A class can implement any number of an interface at a time |
| 4. | Any number of interfaces can be extended by interface. | An interface can never implement any other interface |

### **Difference Between Hashmap and Hashtable**

|  |  |  |
| --- | --- | --- |
| S. No. | Hashmap | Hashtable |
| 1. | **No** method is **synchronized**. | **Every** method is **synchronized**. |
| 2. | Multiple threads can operate **simultaneously** and hence **hashmap’s** object is **not thread-safe.** | **At a time only one thread** is allowed to operate the **Hashtable’s** object. Hence it is thread-safe. |
| 3. | Threads are **not** required to **wait** and hence relatively **performance is high.** | It increases the **waiting** **time** of the thread and hence **performance is low**. |
| 4. | **Null** is **allowed** for both key and value. | Null is **not** **allowed** for both key and value. Otherwise, we will get a null pointer exception. |
| 5. | It is introduced in **the 1.2 version**. | It is introduced in **the 1.0 version.** |
| 6. | It is **non-legacy**. | It is a **legacy**. |

**Synchronised and non-synchronised in Java**

Synchronized basically means that **only one thread** can access methods of that particular class at any given time. **StringBuffer** is an example of a synchronized class. A Synchronized class is a **thread-safe** class.

Non-Synchronized means that **two or more threads** can access the methods of that particular class at any given time. **StringBuilder** is an example of a non-synchronized class. Generally, a non-synchronized class is not thread-safe. (but some non-synchronized classes are thread-safe)

## ****What is an interface?****

An interface is **a behavioural contract** between multiple systems. Any class that implements an interface must also guarantee and provide the implementation for all of its methods. An interface can also be used to define default behaviour for the subclasses. All methods within an interface must be public and abstract.

## ****What is an abstract class?****

An abstract class is a guideline created for the derived concrete classes. There must at least be a **single abstract method**, which provides the implementation for the non-abstract methods as well. The moment of indifference arises if you define an abstract class with implementation, then you might need to consider whether going with the interface would be a better choice.

## ****Differences and Similarities****

All methods that are mentioned in **the interface are public and abstracts implicitly.** Abstract classes can even contain **non-abstract methods**. **They can both have the methods and variables**, and **neither one can** be **instantiated**.

“**All the variables declared within an interface are final, while an abstract class might contain non-final variables.”**

## ****When should you use an abstract class?****

An abstract class is a great choice if you are bringing into account the **inheritance** concept because it provides **the common base class implementation** to the derived classes.

An abstract class is also good if you want to declare **non-public members**. In an **interface**, all methods must be **public**.

If you want to add **new methods in the future**, then it is great to go with the abstract class. Because if you add new methods to the interface, then all of the classes that are already implemented in the interface will have to be changed in order to implement these new methods.

If you want to create **multiple versions of your component**, then go with the abstract class. Abstract classes provide a simple and easy way to version your components. When you go with updating the base class, all of the inheriting classes would be automatically updated with the change. Interfaces, on the other hand, can’t be changed once these are created. If you want a new version of the interface, then you must create a new interface.

Abstract classes always have the advantage of allowing **better** forwarding **compatibility**. Once the clients are onto an interface then you simply can't change it in the end. But if the abstract class is set up, then you can still add the **behaviour without breaking** the existing code.

An abstract class is used if you want to provide a **common, implemented functionality** among all the implementations of the component. Abstract classes will allow you to partially implement your class, whereas interfaces would have no implementation for any members whatsoever.

## ****When Should You Use an Interface?****

If you are creating **functionality that will be useful across** a wide range of objects, then you must use an interface. Abstract classes, at the end of the day, should be used for objects that are closely related. But the interfaces are best suited for providing common functionality to unrelated cases.

Interfaces are a great choice if you think that the **API won’t be changing for a while**.

Interfaces are also a great choice. If you want to have something **similar to the multiple inheritances**, then you can implement various interfaces.

If we are going to design **the small, concise bits of functionality**, then you must use interfaces. But if you are designing the large functional units, then you must use an abstract class.

## What is Servlet?

**Servlet** is a Java technology that is managed by a **container** called a **servlet engine**. It generates **dynamic content** and **interacts** with the **client** through **Request and Response**.

Servlet **extends** the **functionality of a web server.**

Though servlets can respond to many types of requests, they generally **implement web containers** **for hosting** any website on web server. Therefore, it is qualified as a **server-side servlet web API**.

**What is JSP? Java Server Pages**.

**JSP** is a collection of technologies developed by Sun Microsystems.

It is used **to develop web pages** by **inserting Java code** into the **HTML pages** by making **special JSP tags**.

It can consist of either **HTML or XML** (combination of both is also possible) with [JSP Actions](https://www.guru99.com/jsp-action-tags.html) and commands.

The full form of JSP is **Java Server Pages**.

**KEY DIFFERENCES**

* **Servlet** can accept **all protocol requests**, including HTTP, while **JSP** can only accept **HTTP requests**.
* In MVC architecture, servlet works as a **controller** while **JSP** works as a **view** for **displaying output**.
* **Servlet** should be used when there is more **data processing** involved whereas, JSP is generally used when there **is less involvement of data processing**.
* **Servlets run faster than JSP,** on the other hand JSP runs slower than servlet as it takes time to compile the program and convert into servlets.
* You can override the **service**() **method** in servlet but, in JSP, you can’t override the service() method.
* In Servlet, you have to implement both **business logic and presentation logic** in the single file. Whereas in JSP, **business logic is split from presentation logic using JavaBeans.**

### **Dispatcher Servlet**

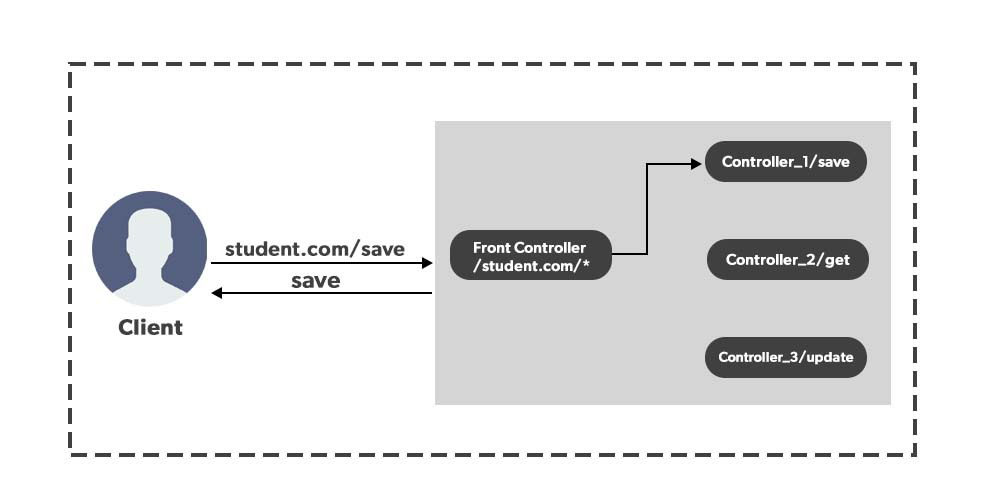
**DispatcherServlet** acts as the **Front Controller** for Spring-based web applications.

what is Front Controller?

Any request is going to come into our website, the front controller is going to stand in front and is going to accept all the requests and once the front controller accepts that request then this is the job of the front controller that it will make a decision that who is the right controller to handle that request.

For example, refer to the below image.

Suppose we have a website called **student.com** and the client is make a request to save student data by hitting the following URL **student.com/save** and its first come to the front controller and once the front controller accepts that request it is going to assign to the Controller\_1 as this controller handle the request for /save operation. Then it is going to return back the response to the Client.



DispatcherServlet **handles an incoming HttpRequest**, **delegates the request**, and **processes that request** according to the configured **HandlerAdapter interfaces** that have been implemented within the Spring application along **with accompanying annotations specifying handlers**, **controller endpoints**, and **response objects**.