**JAVA Interview Questions**

**OOP**

Object

Any entity that has state and behaviour is known as an object.

## Class

Collection of objects is called class. It is a logical entity.

### **Encapsulation**

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

The practice of keeping fields within a class private, then providing access to those fields via public methods. Encapsulation is a protective barrier that keeps the data and code safe within the class itself. We can then reuse objects like code components or variables without allowing open access to the data system-wide.

How is encapsulation implemented in a program?

Data Encapsulation is implemented by **using access specifiers (Access Modifiers)** and it defines the scope and visibility of a class member.

It provides the mechanism which is known as **data hiding**

### **Association**

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

* One to One
* One to Many
* Many to One, and
* Many to Many

**class** Mobile {

**private** String mobile\_no;

**public** String getMobileNo() {

**return** mobile\_no;

    }

**public** **void** setMobileNo(String mobile\_no) {

**this**.mobile\_no = mobile\_no;

    }

}

**class** Person {

**private** String name;

    List<Mobile> numbers;

**public** String getName() {

**return** name;

    }

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

**this**.name = name;

    }

**public** List<Mobile> getNumbers() {

**return** numbers;

    }

**public** **void** setNumbers(List<Mobile> numbers) {

**this**.numbers = numbers;

    }

}

## Types of Association

In Java, two types of **Association** are possible:

1. IS-A Association
2. HAS-A Association
   1. Aggregation
   2. Composition



### **1) IS-A Association**

The IS-A Association is also referred to as [Inheritance](https://www.javatpoint.com/inheritance-in-java). We all know about Inheritance in Java .

### **2) HAS-A Association**

The **HAS-A Association** is further classified into two parts, i.e., Aggregation and Composition. Let's understand the difference between both of them one by one.

### **Aggregation**

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a has-a relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.

In Java, the [**Aggregation**](https://www.javatpoint.com/aggregation-in-java) association defines the **HAS-A** relationship. Aggregation follows the one-to-one or one-way relationship. If two entities are in the aggregation composition, and one entity fails due to some error, it will not affect the other entity.

**class** Employee{

**int** id;

String name;

Address address;//Address is a class

...

}

### **Composition**

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

**class** College {

**public** String name;

**public** String address;

    College(String name, String address)

    {

**this**.name = name;

**this**.address = address;

    }

}

// University has more than one college.

**class** University {

    // reference to refer to list of college.

**private** **final** List<College> colleges;

    University(List<College> colleges)

    {

**this**.colleges = colleges;

    }

    // Getting total number of colleges

**public** List<College> getTotalCollegesInUniversity()

    {

**return** colleges;

    }

}

#### **Abstraction**

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, abstraction means simple things like **objects**, **classes** and **variables** represent more complex underlying code and data. This is important because it lets you avoid repeating the same work multiple times.

### **Polymorphism**

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

### **Inheritance**

When one object acquires all the properties and behaviours of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

**Difference between Abstraction and Encapsulation**

Abstraction hides complexity by giving you a more abstract picture, a sort of 10,000 feet view, while Encapsulation hides internal working so that you can change it later. In other words, Abstraction hides details at the **design**level, while Encapsulation hides details at the **implementation** level.

For example, when you first describe an object, you talk in more abstract terms e.g. a Vehicle that can move, you don't tell how the Vehicle will move, whether it will move by using tires or it will fly or it will sell. It just moves. This is called [Abstraction](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html). We are talking about the most essential thing, which is moving, rather than focusing on details like moving in-plane, sky, or water.

On the other hand, Encapsulation is all about implementation. Its sole purpose is to hide the internal working of objects from the outside world so that you can change it later without impacting outside clients.

For example, we have a HashMap which allows you to store the object using the put() method and retrieve the object using the get() method. How HashMap implements this method is an internal detail of HashMap, the client only cares that put stores the object and get return it back, they are not concerned whether HashMap is using an array, how it is resolving the collision, whether it is using [linked list](http://javarevisited.blogspot.com/2016/07/how-to-find-3rd-element-from-end-in-linked-list-java.html) or [binary tree](http://javarevisited.blogspot.sg/2016/10/post-order-binary-tree-traversal-in-java-iteration-recursion.html) to store object landing on the same bucket, etc.  
  
Because of **Encapsulation**, you can change the internal implementation of HashMap with ease without impacting clients who are using HashMap. For example, in Java 8, the java.util.HashMap changes its implementation to use a binary tree instead of LinkedList to store objects in the same bucket after a certain threshold.

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information. | Encapsulation is also a feature of OOPs. It hides the code and data into a **single** entity or unit so that the data can be protected from the outside world. |
| It solves an issue at the **design** level. | Encapsulation solves an issue at **implementation** level. |
| It focuses on the **external** lookout. | It focuses on **internal** working. |
| It can be implemented using **abstract classes** and **interfaces**. | It can be implemented by using the [**access modifiers**](https://www.javatpoint.com/access-modifiers) (private, public, protected). |
| It is the process of **gaining** information. | It is the process of **containing** the information. |
| In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities. | We use the **getters** and **setters** methods to hide the data. |
| The objects are **encapsulated** that helps to perform abstraction. | The object need not to **abstract** that result in encapsulation. |
| Abstraction | Encapsulation |
| Abstraction is the process or method of gaining the information. | While encapsulation is the process or method to contain the information. |
| In abstraction, problems are solved at the design or interface level. | While in encapsulation, problems are solved at the implementation level. |
| Abstraction is the method of hiding the unwanted information. | Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside. |
| We can implement abstraction using abstract class and interfaces. | Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public. |
| In abstraction, implementation complexities are hidden using abstract classes and interfaces. | While in encapsulation, the data is hidden using methods of getters and setters. |
| The objects that help to perform abstraction are encapsulated. | Whereas the objects that result in encapsulation need not be abstracted. |
| Abstraction provides access to specific part of data. | Encapsulation hides data and the user can not access same directly (data hiding. |
| Abstraction focus is on “what” should be done. | Encapsulation focus is on “How” it should be done. |

Why constructor cannot be final?

Whenever you make a method final, you cannot **override** it. i.e. you cannot provide implementation to the superclass's final method from the subclass.

i.e. The purpose of making a method final is to prevent modification of a method from outside (child class).

In inheritance whenever you extend a class. The child class inherits all the members of the superclass except the constructors.

In other words, constructors cannot be inherited in Java therefore you cannot **override** constructors.

So, writing final before constructors makes no sense. Therefore, java does not allow final keyword before a constructor.

### **Methods of Object class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

# Overloading in Java

## Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism

**What if the exact prototype does not match with arguments.**  
Ans.  
Priority wise, compiler take these steps:

1. Type Conversion but to higher type(in terms of range) in same family.
2. Type conversion to next higher family(suppose if there is no long data type available for an int data type, then it will search for the float data type).

## **Can we overload methods on return type?** We **cannot** overload by return type. However, Overloading methods on return type are possible in cases where the data type of the function being called is explicitly specified.

class A {

    public int foo(int a) { return 10; }

    public char foo(int a, int b) { return 'a'; }

}

## **Can we overload static methods?** The answer is ‘**Yes**’. We can have two ore more static methods with same name, but differences in input parameters.

**Can we overload methods that differ only by static keyword?**  
We **cannot** overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same).

**Can we overload main() in Java?**  
Like other static methods, we **can** [overload main() in Java](https://www.geeksforgeeks.org/gfact-48-overloading-main-in-java/).

# Overriding in Java

## In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

**Rules for method overriding:**

**Overriding and Access-Modifiers :** The [access modifier](https://www.geeksforgeeks.org/access-modifiers-java/) for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

## **Final methods can not be overridden :** If we don’t want a method to be overridden, we declare it as [final](https://www.geeksforgeeks.org/final-keyword-java/).

## **Static methods can not be overridden(Method Overriding vs Method Hiding) :**When you define a static method with same signature as a static method in base class, it is known as [method hiding](https://www.geeksforgeeks.org/can-we-overload-or-override-static-methods-in-java/).

## **Private methods can not be overridden :** [Private methods](https://www.geeksforgeeks.org/can-override-private-methods-java/) cannot be overridden as they are bonded during compile time. Therefore we can’t even override private methods in a subclass.

**The overriding method must have same return type (or subtype) :** From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child’s return type should be sub-type of parent’s return type. This phenomena is known as [**covariant return type**](https://www.geeksforgeeks.org/covariant-return-types-java/).

## **Invoking overridden method from sub-class :** We can call parent class method in overriding method using [super keyword](https://www.geeksforgeeks.org/super-keyword/).

**Overriding and Exception-Handling :**Below are two rules to note when overriding methods related to exception-handling.

* **Rule#1 :** If the super-class overridden method does not throw an exception, subclass overriding method can only throws the [unchecked exception](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/), throwing checked exception will lead to compile-time error.

## **Rule#2 :** If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in [Exception hierarchy](https://www.geeksforgeeks.org/exceptions-in-java/) will lead to compile time error.Also there is no issue if subclass overridden method is not throwing any exception.

## Access Modifiers

 access modifiers in Java helps to restrict the scope of a class, constructor, variable, method, or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

## access-modifiers-in-java

## What is the use of making class private when anyone can access using get and set method in encapsulation?

If you are wondering what is the point of declaring variables private when they can be accessed using public getters and setters, let me try to explain

1- By declaring variables no one can access then directly outside of the class.

2- About public getters and setters, it depends on you and the use case. You may choose not to create any getters and setters. But then what is the point of declaring a variable which cant be used? You may want to use it inside the class only for some reason.

3- The idea behind declaring the variables private and creating public getters and setters is to keep track/validation on how your variable should be used. For example :

1. class Dog {
2. public int age;
3. {

If you see the above class anyone can set any value in the age field. It could be 0 or any negative or maximum value of **int.** But if you think of a real-life scenario you wouldn't want to do that. Age can never be a negative number. second, you want to put validation on the max value as well. For example, max age of any dog cant be more than 20 years. Or maybe if there is no age set for a Dog you may want to throw an exception that invalid age found or something. You can achieve that only using getters and setters. See the below example.

1. class Dog{
2. private int age;
3. public void setAge(int age){
4. if(age !> 0 || age >20){
5. // throw some invalid age error
6. }
7. else {
8. this.age = age;
9. }
10. }
11. public int getAge(){
12. if(this.age <=0 ){
13. // throw invalid/no age found for this dig
14. }
15. else {
16. return this.age;
17. }
18. }
19. }

### **Programming to the interface**

Coding to interfaces is a technique to write classes based on an interface; interface that defines what the behavior of the object should be. It involves creating an interface first, defining its methods and then creating the actual class with the implementation. At first, it looks like over coding, but, there are two main reasons that why I decided to go for it: Test Driven Development and Flexibility.

On the testing side, is absolutely needed. TDD is about creating the tests for the functionality to achieve, to later develop the code required to pass the tests. As there is no implementation of the code yet, using Interfaces to Mock objects is necessary to accomplish the test cases; when the actual implementation is ready, is just a matter of substituting the mocks and injecting the real object. If the object complies with the interface, your tests will behave the same way as the mocks. It also works when creating tests that depend on code done by another person or team and you do not have the real object yet.

On the other hand, there is flexibility and maintainability. To explain why, I’ll first tell a story about our project. A few days ago (curiously the same day that my friend asked about interfaces), due to a change of requirements, we had to change the two API’s that we were using on our project; These APIs were, nevertheless, part of our core functionality.

When looking at the new APIs, we realized that refactoring the code was going to be a lot more work than just refactoring the code. Why? Because we had coded against interfaces, therefore, as soon as we finish the new code, all other objects dependent on the implementations did not even notice that something changed. And guess what? All our tests passed on the first run, like if nothing had changed.

# **Why wait ,notify and notifyAll Methods are in Object Class?**

In the Java language, you wait() on a particular instance of an Object -- a monitor assigned to that object to be precise. If you want to send a signal to a single thread that is waiting on that specific object instance then you call notify() on that object. If you want to send a signal to all threads that are waiting on that object instance, you use notifyAll() on that object.  
  
If wait() and notify() were on the Thread instead then each thread would have to know the status of every other thread. How would thread1 know that thread2 was waiting for access to a particular resource? If thread1 needed to call thread2.notify() it would have to somehow find out that thread2 was waiting. There would need to be some mechanism for threads to register the resources or actions that they need so others could signal them when stuff was ready or available.  
  
In Java, the object itself is the entity that is shared between threads which allows them to communicate with each other. The threads have no specific knowledge of each other and they can run asynchronously. They run and they lock, wait, and notify on the object that they want to get access to. They have no knowledge of other threads and don't need to know their status. They don't need to know that it is thread2 which is waiting for the resource -- they just notify on the resource and whomever it is that is waiting (if anyone) will be notified.  
  
In Java, we then use lock objects as synchronization, mutex, and communication points between threads. We synchronize on a lock object to get mutex access to an important code block and to synchronize memory. We wait on a lock object if we are waiting for some condition to change -- some resource to become available. We notify on an object if we want to awaken sleeping threads.

## ****Stack Memory in Java****

**Stack Memory in Java is used for static memory allocation and the execution of a thread.** It contains primitive values that are specific to a method and references to objects referred from the method that are in a heap.

Access to this memory is in Last-In-First-Out (LIFO) order. Whenever we call a new method, a new block is created on top of the stack which contains values specific to that method, like primitive variables and references to objects.

When the method finishes execution, its corresponding stack frame is flushed, the flow goes back to the calling method, and space becomes available for the next method.

### **2.1. Key Features of Stack Memory**

Some other features of stack memory include:

* It grows and shrinks as new methods are called and returned, respectively.
* Variables inside the stack exist only as long as the method that created them is running.
* It's automatically allocated and deallocated when the method finishes execution.
* If this memory is full, Java throws java.lang.StackOverFlowError.
* Access to this memory is fast when compared to heap memory.
* This memory is threadsafe, as each thread operates in its own stack.

## ****3. Heap Space in Java****

**Heap space is used for the dynamic memory allocation of Java objects and JRE classes at runtime**. New objects are always created in heap space, and the references to these objects are stored in stack memory.

These objects have global access and we can access them from anywhere in the application.

We can break this memory model down into smaller parts, called generations, which are:

1. **Young Generation –**this is where all new objects are allocated and aged. A minor Garbage collection occurs when this fills up.
2. **Old or Tenured Generation –**this is where long surviving objects are stored. When objects are stored in the Young Generation, a threshold for the object's age is set, and when that threshold is reached, the object is moved to the old generation.
3. **Permanent Generation –**this consists of JVM metadata for the runtime classes and application methods.

### **3.1. Key Features of Java Heap Memory**

Some other features of heap space include:

* It's accessed via complex memory management techniques that include the Young Generation, Old or Tenured Generation, and Permanent Generation.
* If heap space is full, Java throws java.lang.OutOfMemoryError.
* Access to this memory is comparatively slower than stack memory
* This memory, in contrast to stack, isn't automatically deallocated. It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage.
* Unlike stack, a heap isn't threadsafe and needs to be guarded by properly synchronizing the code.

## ****4. Example****

Based on what we've learned so far, let's analyze a simple Java code to assess how to manage memory here:

**class** **Person** {

**int** id;

String name;

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

this.id = id;

this.name = name;

}

}

**public** **class** **PersonBuilder** {

**private** **static** Person **buildPerson**(**int** id, String name) {

**return** **new** **Person**(id, name);

}

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

**int** id = 23;

**String** name = "John";

**Person** person = null;

person = buildPerson(id, name);

}

}

Let's analyze this step-by-step:

1. When we enter the main() method, a space in stack memory is created to store primitives and references of this method.
   * Stack memory directly stores the primitive value of integer id.
   * The reference variable person of type Person will also be created in stack memory, which will point to the actual object in the heap.
2. The call to the parameterized constructor Person(int, String) from main() will allocate further memory on top of the previous stack. This will store:
   * The this object reference of the calling object in stack memory
   * The primitive value id in the stack memory
   * The reference variable of String argument name, which will point to the actual string from string pool in heap memory
3. The main method is further calling the buildPerson() static method, for which further allocation will take place in stack memory on top of the previous one. This will again store variables in the manner described above.

| **Parameter** | **Stack Memory** | **Heap Space** |
| --- | --- | --- |
| Application | Stack is used in parts, one at a time during execution of a thread | The entire application uses Heap space during runtime |
| Size | Stack has size limits depending upon OS, and is usually smaller than Heap | There is no size limit on Heap |
| Storage | Stores only primitive variables and references to objects that are created in Heap Space | All the newly created objects are stored here |
| Order | It's accessed using Last-in First-out (LIFO) memory allocation system | This memory is accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation. |
| Life | Stack memory only exists as long as the current method is running | Heap space exists as long as the application runs |
| Efficiency | Much faster to allocate when compared to heap | Slower to allocate when compared to stack |
| Allocation/Deallocation | This Memory is automatically allocated and deallocated when a method is called and returned, respectively | Heap space is allocated when new objects are created and deallocated by Gargabe Collector when they're no longer referenced |

1. However, heap memory will store all instance variables for the newly created object person of type Person.

## Spring Boot Starter Parent

The spring-boot-starter-parent is a project starter. It provides default configurations for our applications. It is used internally by all dependencies. All Spring Boot projects use spring-boot-starter-parent as a parent in pom.xml file.

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.4.0.RELEASE</version>

</parent>

Parent Poms allow us to manage the following things for multiple child projects and modules:

* **Configuration:** It allows us to maintain consistency of Java Version and other related properties.
* **Dependency Management:** It controls the versions of dependencies to avoid conflict.
* Source encoding
* Default Java Version
* Resource filtering
* It also controls the default plugin configuration.

**Serialization**

Serialization in Java is the concept of representing an object's state as a byte stream. The byte stream has all the information about the object. Usually used in Hibernate, JMS, JPA, and EJB, serialization in Java **helps transport the code from one JVM to another and then de-serialize it there**.

**Difference between map and flatMap?**

Map() can be used where we have to map the elements of a particular collection to a certain function, and then we need to return the stream which contain the updated results.

**Stream map(Function mapper)** returns a stream consisting of the results of applying the given function to the elements of this stream.

Stream map(Function mapper) is an **intermediate operation**. These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

S**tream flatMap(Function mapper)** returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Stream flatMap(Function mapper) is an ***intermediate operation***. These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

**Note :**Each mapped stream is closed after its contents have been placed into this stream. If a mapped stream is null, an empty stream is used, instead.

**flatMap() V/s**[map()](https://www.geeksforgeeks.org/stream-map-java-examples/)**:**  
1) [map()](https://www.geeksforgeeks.org/stream-map-java-examples/) takes a Stream and transform it to another Stream. It applies a function on each element of Stream and store return value into new Stream. It does not flatten the stream. But flatMap() is the combination of a map and a flat operation i.e, it applies a function to elements as well as flatten them.  
2) [map()](https://www.geeksforgeeks.org/stream-map-java-examples/) is used for transformation only, but flatMap() is used for both transformation and flattening.

flatMap() can be used where we have to flatten or transform out the string, as we cannot flatten our string using map().

FlatMap converts or transforms one type to another as like map() method. Used in the context where each element in the stream represent multiple elements like Stream<List> and Stream<Arrays>.

| map() | flatMap() |
| --- | --- |
| The function passed to map() operation returns a single value for a single input. | The function you pass to flatmap() operation returns an arbitrary number of values as the output. |
| One-to-one mapping occurs in map(). | One too many mapping occurs in flatMap(). |
| Only perform the mapping. | Perform mapping as well as flattening. |
| Produce a stream of value. | Produce a stream of stream value. |
| map() is used only for transformation. | flatMap() is used both for transformation and mapping. |

**For-each loop**

It is used to iterate over an array or a Collections class.Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.

ForEach method

We use forEach to iterate over a collection and perform a certain action on each element. **The action to be performed is contained in a class that implements the Consumer interface and is passed to forEach as an argument.**

**Equals and ‘==’**

**Both equals() method** and the == operator are used to compare two objects in Java. == is an operator and equals() is method. But == operator compares reference or memory location of objects in a heap, whether they point to the same location or not. ... equals() evaluates to the comparison of values in the objects.

Default methods use case:

In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement too. Otherwise, the design will break down.

Default interface methods are an efficient way to deal with this issue.

Static Methods

They don’t belong to any particular object. They are not part of the API of the classes implementing the interfaces. Therefore they have to be called by using the interface name preceding the method name.

**Streams**

Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

**Functional Interfaces**

Any interface with a SAM(Single Abstract Method) is a functional interface, and its implementation may be treated as lambda expressions.

**we can use lambda expressions to instantiate them and avoid using bulky anonymous class implementation**.

Java 8 Supplier is a functional interface whose functional method is get(). The Supplier interface **represents an operation that takes no argument and returns a result**.

@FunctionalInterface public interface Consumer<T> Represents an operation that accepts a single input argument and returns no result.]

The major benefit of java 8 functional interfaces is that **we can use lambda expressions to instantiate them and avoid using bulky anonymous class implementation**.

Anonymous Comparator Implementation

Collections.sort(Employee\_list, new Comparator<Employee>(){

public int compare(Employee e1, Employee e2){

if(e1.salary < e2.salary){

return 1;

}

return -1;

}

});

Lambda implementation

Collections.sort(Employee\_list, (Employee e1,Employee e2) ->{

if(e1.salary < e2.salary){

return 1;

}

return -1;

});

Predicate

In Java 8, Predicate is **a functional interface, which accepts an argument and returns a boolean**. Usually, it used to apply in a filter for a collection of objects.

@FunctionalInterface public interface Predicate<T> {

boolean test(T t);

}

Function

[Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) is a functional interface; it takes an argument (object of type T) and returns an object (object of type R). The argument and output can be a different type.

### Lambda Expression

Lambda Expression are the way through which we can visualize **functional programming** in the java object oriented world. Objects are the base of java programming language and we can never have a function without an Object, that’s why Java language provide support for using lambda expressions only with functional interfaces.

Since there is only one abstract function in the functional interfaces, there is no confusion in applying the lambda expression to the method. Lambda Expressions syntax is **(argument) -> (body)**. Now let’s see how we can write above anonymous Runnable using lambda expression.

Runnable r1 = () -> System.out.println("My Runnable");

Let’s try to understand what is happening in the lambda expression above.

* Runnable is a functional interface, that’s why we can use lambda expression to create it’s instance.
* Since run() method takes no argument, our lambda expression also have no argument.
* Just like if-else blocks, we can avoid curly braces ({}) since we have a single statement in the method body. For multiple statements, we would have to use curly braces like any other methods.

### Why do we need Lambda Expression

1. **Reduced Lines of Code**  
   One of the clear benefit of using lambda expression is that the amount of code is reduced, we have already seen that how easily we can create instance of a functional interface using lambda expression rather than using anonymous class.
2. **Sequential and Parallel Execution Support**

Another benefit of using lambda expression is that we can benefit from the Stream API sequential and parallel operations support.

To explain this, let’s take a simple example where we need to write a method to test if a number passed is prime number or not.

Traditionally we would write it’s code like below. The code is not fully optimized but good for example purpose, so bear with me on this.

//Traditional approach

private static boolean isPrime(int number) {

if(number < 2) return false;

for(int i=2; i<number; i++){

if(number % i == 0) return false;

}

return true;

}

The problem with above code is that it’s sequential in nature, if the number is very huge then it will take significant amount of time. Another problem with code is that there are so many exit points and it’s not readable. Let’s see how we can write the same method using lambda expressions and stream API.

//Declarative approach

private static boolean isPrime(int number) {

return number > 1

&& IntStream.range(2, number).noneMatch(

index -> number % index == 0);

}

IntStream is a sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. This is the int primitive specialization of Stream.

For more readability, we can also write the method like below.

private static boolean isPrime(int number) {

IntPredicate isDivisible = index -> number % index == 0;

return number > 1

&& IntStream.range(2, number).noneMatch(

isDivisible);

}

If you are not familiar with IntStream, it’s range() method returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

noneMatch() method returns whether no elements of this stream match the provided predicate. It may not evaluate the predicate on all elements if not necessary for determining the result.

1. **Passing Behaviors into methods**

Let’s see how we can use lambda expressions to pass behavior of a method with a simple example. Let’s say we have to write a method to sum the numbers in a list if they match a given criteria. We can use Predicate and write a method like below.

public static int sumWithCondition(List<Integer> numbers, Predicate<Integer> predicate) {

return numbers.parallelStream()

.filter(predicate)

.mapToInt(i -> i)

.sum();

}

Sample usage:

//sum of all numbers

sumWithCondition(numbers, n -> true)

//sum of all even numbers

sumWithCondition(numbers, i -> i%2==0)

//sum of all numbers greater than 5

sumWithCondition(numbers, i -> i>5)

1. **Higher Efficiency with Laziness**

One more advantage of using lambda expression is the lazy evaluation, for example let’s say we need to write a method to find out the maximum odd number in the range 3 to 11 and return square of it.

Usually we will write code for this method like this:

private static int findSquareOfMaxOdd(List<Integer> numbers) {

int max = 0;

for (int i : numbers) {

if (i % 2 != 0 && i > 3 && i < 11 && i > max) {

max = i;

}

}

return max \* max;

}

Above program will always run in sequential order but we can use Stream API to achieve this and get benefit of Laziness-seeking. Let’s see how we can rewrite this code in functional programming way using Stream API and lambda expressions.

public static int findSquareOfMaxOdd(List<Integer> numbers) {

return numbers.stream() .filter(NumberTest::isOdd) //Predicate is functional interface and

.filter(NumberTest::isGreaterThan3) // we are using lambdas to initialize it

.filter(NumberTest::isLessThan11) // rather than anonymous inner classes

.max(Comparator.naturalOrder())

.map(i -> i \* i) .get(); }

public static boolean isOdd(int i) {

return i % 2 != 0; }

public static boolean isGreaterThan3(int i){

return i > 3; }

public static boolean isLessThan11(int i){

return i < 11;

}

Strings

A *String* instance in Java is an object with two fields: a *char[] value* field and an *int hash* field. The *value* field is an array of chars representing the string itself, and the *hash* field contains the *hashCode* of a string which is initialized with zero, calculated during the first *hashCode()* call and cached ever since. As a curious edge case, if a *hashCode* of a string has a zero value, it has to be recalculated each time the *hashCode()* is called.

Important thing is that a *String* instance is immutable: you can't get or modify the underlying *char[]* array. Another feature of strings is that the static constant strings are loaded and cached in a string pool. If you have multiple identical *String* objects in your source code, they are all represented by a single instance at runtime.

**What happens when String value is changed?**

Since strings are immutable the value cannot change but **new object will be created** with changed values. once we create a String object, we can't perform any changes in the existing object. If we try to do so, a new object will be created.

**Memory allotment of String**

Whenever a String Object is created as a literal, the object will be created in String constant pool. This allows JVM to optimize the initialization of String literal.

**For example:**

String str = "Geeks";

The string can also be declared using **new** operator i.e. dynamically allocated. In case of String are dynamically allocated they are assigned a new memory location in heap. This string will not be added to String constant pool.

**For example:**

String str = new String("Geeks");

If you want to store this string in the constant pool then you will need to “intern” it.

**For example:**

String internedString = str.intern();

// this will add the string to string constant pool.

It is preferred to use String literals as it allows JVM to optimize memory allocation.

Why String is immutable?

The benefits of this class as immutable are caching, synchronizing, and performance.

String is the most widely used data Structure. Caching the String literals and reusing them saves a lot of heap space because different variables refer to the same object in the string pool.

### **Security**

**If Strings were mutable, then by the time we execute the update, we can't be sure that the String we received, even after performing security checks, would be safe.** The untrustworthy caller method still has the reference and can change the String between integrity checks. Thus making our query prone to SQL injections in this case. So mutable Strings could lead to degradation of security over time.

It could also happen that the String userName is visible to another thread, which could then change its value after the integrity check.

In general, immutability comes to our rescue in this case because it's easier to operate with sensitive code when values don't change because there are fewer interleavings of operations that might affect the result.

### **Synchronization**

Being immutable automatically makes the String thread safe since they won't be changed when accessed from multiple threads.

Hence **immutable objects, in general, can be shared across multiple threads running simultaneously. They're also thread-safe** because if a thread changes the value, then instead of modifying the same, a new String would be created in the String pool. Hence, Strings are safe for multi-threading.

### **Hashcode Caching**

Since String objects are abundantly used as a data structure, they are also widely used in hash implementations like HashMap, HashTable, HashSet, etc. When operating upon these hash implementations, hashCode() method is called quite frequently for bucketing.

The immutability guarantees Strings that their value won’t change. So **the hashCode() method is overridden in String class to facilitate caching, such that the hash is calculated and cached during the first hashCode() call and the same value is returned ever since.**

**This, in turn, improves the performance of collections that uses hash implementations when operated with String objects.**

On the other hand, mutable Strings would produce two different hashcodes at the time of insertion and retrieval if contents of String was modified after the operation, potentially losing the value object in the Map.

### **Performance**

As we saw previously, String pool exists because Strings are immutable. In turn, it enhances the performance by saving heap memory and faster access of hash implementations when operated with Strings.

Since String is the most widely used data structure, improving the performance of String have a considerable effect on improving the performance of the whole application in general.

 the contract between equals() and hashCode() methods.

1. If obj1.equals(obj2) returns true, hashCode() should be same for the two objects (obj1 & obj2).
2. If hashCode() is same, equals() may or may not return true.

This contract is made to make hashing technique work correctly if you want to use a class as key in HashMap.

**What happens if we don't override hashCode()?**

If we override only equals() and not hashCode(), the class uses its parent implementation. The default implementation which is provided in Object class (root) is to return memory location address of the object. So, if two objects are having same data, equals() will return true but hashCode() will return different values (two different address locations. If you are not using this class as key in HashMap or not using it in HashSet, you won't see any issues with your code.

**What happens if you use this class in HashMap?**

Consider there are two objects key1 and key2 with below properties.

1. key1.equals(key2) =true
2. key1.hashCode() != key2.hashCode()

When you try to put these two keys in a HashMap, what you expect is to have only one key stored (as they are same) but end up having both keys in the map. It is because the two keys will be stored in two different buckets in the map (hash bucket is decided based on hashcode which is different here.

The ideal way is to overrode hashCode() along with equals() method as you may be required to use it as key in HashMap in future.

Hope this answers your question.

PS: There is one more rule for using a class to be immutable if you want use it as key in HashMap. It is because if hashcode is changed for a key after it is inserted in HashMap, map doesn't work correctly.

If two objects are not equal but their hashcode is same, it is called collision. As the return type of hashCode() is intended, you can have a max of 2^32 distinct values of hashcode. So, even for a good hashCode() implementation you will have collisions. So, the motto should be to reduce the number of collisions.

Thus equals() methods compare two entities and return true if they are logically the same. Since equals() is a method defined in the Object class thus the default implementation of the equals() method compares the object references or the memory location where the objects are stored in the heap. Thus by default the equals() method checks the object by using the “==” operator.

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBu |

*StringBuilder* allows manipulating character sequences by appending, deleting and inserting characters and strings. This is a mutable data structure, as opposed to the *String* class which is immutable.

When concatenating two *String* instances, a new object is created, and strings are copied. This could bring a huge garbage collector overhead if we need to create or modify a string in a loop. *StringBuilder* allows handling string manipulations much more efficiently.

*StringBuffer* is different from *StringBuilder* in that it is thread-safe. If you need to manipulate a string in a single thread, use *StringBuilder* instead.

**Difference between String and StringBuffer/StringBuilder:**

String is immutable in nature whereas StringBuffer and StringBuilder are mutable.

**Immutable:**immutable means non-changeable. It creates new objects every time you create strings or assign a new string(change the value).

**Mutable:** mutable means changeable. It is allowed to change values in existing object rather creating new object.

1. String str = "Hello"; //new object is created
2. str = str + "Welcome"; //new object is created internally
3. StringBuffer str1 = "Hello"; //new object created
4. str1 = str1 + "Welcome";
5. StringBuilder str2 = "Hello"; //new object created
6. str2 = str2 + "Welcome";

**Difference between StringBuffer and StringBuilder:**

StringBuffer is synchronized( which means it is thread safe) whereas StringBuilder is not synchronized( which implies it isn’t thread safe).

**Recommendation:**

1. It is recommended to use String when value of string is non-changeable.
2. It is recommended to use StringBuffer when value of string is changeable and application is implemented in multithreaded environment.
3. It is recommended to use StringBuilder when value of string is changeable but application is not using multiple threads.

Exception

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling in Java

The code can experience errors while executing our instructions. Good exception handling can handle errors and gracefully re-route the program to give user still a positive experience.

We usually write code in an idealized environment. The file system contains our files, network is healthy and the JVM always has enough memory( we call this a happy path).

## Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



### **Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

## Difference between Checked and Unchecked Exceptions

### **1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### **2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### **3) Error**

Errors are **conditions that cannot get recovered by any handling techniques**. It surely causes termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime.Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

**Concurrency in Java**

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

If the class extends the Thread class, the thread can be run by creating an instance of the class and call its start() method:

If the class implements the Runnable interface, the thread can be run by passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:

Difference between @Primary and @Qualifier

Ambiguity happens when multiple beans implement the same type and thus Spring fails to resolve the dependency.

**You can use @Primary to give higher preference to a bean when there are multiple beans of the same type.** Because autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process.

**@Primary** is an effective way to use autowiring by type with several instances when you can determine one primary candidate. Alternatively, you can use Spring’s **@Qualifier** annotation to control the selection process using the bean name.

I have around 3 years of experience as Java Developer. I have experience working with spring boot framework spring mvc and I have experience working on restFull webservices, spring JPA and microservices architecture recently working with insurance client.

@Component, @Service and @Repository

These components when annotated give spring the authority to create objects

Dataaccesseception

Logging

**Versioning in rest api**

 Through a **URI path** – you include the version number in the URL path of the endpoint, for example, */api/v1/persons.*

 Through **query parameters** – you pass the version number as a query parameter with a specified name, for example, */api/persons?version=1.*

 Through **custom HTTP headers** – you define a new header that contains the version number in the request.

 Through a **content negotiation** – the version number is included in the “Accept” header together with the accepted content type.

Difference between throw and throws keyword in exception handling?

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.

Actuators

Actuator is mainly **used to expose operational information about the running application** — health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it. Once this dependency is on the classpath, several endpoints are available for us out of the box

**Spring Boot Actuator Endpoints**

Spring Boot Actuator Endpoints are exposed over [JMX](https://www.digitalocean.com/community/tutorials/what-is-jmx-mbean-jconsole-tutorial) and HTTP, most of the times we use HTTP based Actuator endpoints because they are easy to access over the browser, CURL command, shell scripts etc. Some of the useful actuator endpoints are:

1. beans: this endpoint returns the list of all the beans configured in our application.
2. env: provides information about the Spring Environment properties.
3. health: Shows application health
4. info: Displays application information, we can configure this in Spring environment properties.
5. mappings: Displays the list of all [@RequestMapping](https://www.digitalocean.com/community/tutorials/spring-requestmapping-requestparam-pathvariable-example) paths.
6. shutdown: allows us to gracefully shutdown the application.
7. threaddump: provides the thread dump of the application.

Service Discovery

Service Discovery integration with Amazon ECS uses Amazon route 53 to create a namespace for your service, which allows it to be discoverable via DNS.

Amazon ECS creates and manage a registry of service names using Route 53 auto naming API. Names are automatically mapped to a set of DNS records so that you can refer to a service by name in your code and write DNS queries to have the name resolve to the service’s endpoint at runtime.

ECS updated service Registry(Route 53) based on naming convention, task registration, task deregistration and health.

API Gateway

Route the requests to appropriate microservices. Single entry point.

Feature #1 Separate out cross cutting concerns

Feature #2 Separate and consolidate cross cutting concerns across

Feature #3 Replacing multiple client calls with single API call. And, some features of reverse proxy

Feature #4 Routing based on headers, paths and params etc. And, some features of Load Balancer component

AOP

**AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.** It does this by adding additional behavior to existing code without modifying the code itself.

Aspects – Reusable blocks of code that you can inject at runtime.

JoinPoint – cross cutting concern applied

Pointcut – a routine used as selection criteria to select joinPoint to apply that cross cutting concern

Advice – a code that is applied to the jointPoint when selected by the pointcut this is the code that applies cross cutting concern.

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

**public** @interface LogExecutionTime {

}

@Around("@annotation(LogExecutionTime)")

**public** Object **logExecutionTime**(ProceedingJoinPoint joinPoint) **throws** Throwable {

**long** start = System.currentTimeMillis();

**Object** proceed = joinPoint.proceed();

**long** executionTime = System.currentTimeMillis() - start; System.out.println(joinPoint.getSignature() + " executed in " + executionTime + "ms");

**return** proceed;

}

@LogExecutionTime **public** **void** **serve**() **throws** InterruptedException { Thread.sleep(2000);

}

Wrapper classes

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

## Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Why we need Wrapper Class

* Wrapper Class will **convert primitive data types into objects**. The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are **passed by value**).
* The classes in **java.util package** handles only objects and hence **wrapper classes**help in this case also.
* **Data** **structures** in the Collection framework such as **ArrayList and Vector**store only the objects (reference types) and not the**primitive types.**
* The object is needed to support **synchronization** in **multithreading**.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity. | Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object. |
| 5) ArrayList uses the **Iterator** interface to traverse the elements. | A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements. |

Implementation of HashMap

HashMap stores the data in key-value pairs. HashMap contains array of nodes, and the node is represented as class. It uses an array and LinkedList data structure internally storing key and vaue. There are 4 fields in hashMap.

Node, key, value, hash.

* **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
* **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
* **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

The default size of HashMap is 16 (0 to 15).

When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Index minimizes the size of the array. The Formula for calculating the index is:

Index = hashcode(Key) & (n-1)

## Hash Collision

This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

Index=63281940 & (16-1) = 4

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.

# [**Polymorphism: Why use "List list = new ArrayList" instead of "ArrayList list = new ArrayList"? [duplicate]**](https://stackoverflow.com/questions/9852831/polymorphism-why-use-list-list-new-arraylist-instead-of-arraylist-list-n)

The main reason you'd do this is to decouple your code from a specific implementation of the interface. When you write your code like this:

In both your examples, only the expression new creates instances of objects (the right hand side of the =), the left hand side only defines a variable that points to the object that you created.

In both cases, the object that you create is of type ArrayList.

List is an interface, not a class, and interfaces themselves cannot be instantiated. But ArrayList *implements* the interface List, so you can assign an instance of ArrayList to a variable of type List.

The advantage of Example 1 is that you can later decide to create another implementation of List (like LinkedList) and all your code that was using the List type for variables will still work without any change. While in Example 2, if you later decide that you need a different type of List, you need to change the type of all your variables that point to the actual object.

Your questions:

But what I want to know what Example 1 actually means. Does it create a new List? Or does it create a new ArrayList?

It creates a new ArrayList.

Does the resulting object have the properties of a List? Or does the resulting object have the properties of an ArrayList?

The resulting object *has* all the properties of an ArrayList.

However, through the variable list which as type List, you can only access the methods defined in the interfaces List.

But you can use *type-casting* to access the methods in ArrayList like this, if you later need to (but there is little reason as ArrayList doesn't have much beyond what's in List)

List<Integer> list = new ArrayList<Integer>(); //Example 1

ArrayList<Integer> arrayList = (ArrayList<Integer>) list; // type-cast

List<String> listofNums = new ArrayList("A", "B", "C");

List<Integer> numbers = new ArrayList(1, 2, 3, 4, 5);

Optional<String> newList = listofNums.stream()

.map(System.out::println)

List<Integer> anotherList = numbers.stream()

.filter(n -> (n%2 == 0));

Why Functional interface used?

Functional interfaces are introduced as part of Java 8. It is implemented using the annotation called @FunctionalInterface. It ensures that the interface should have only one abstract method. The usage of the abstract keyword is optional as the method defined inside interface is by default abstract. It is important to note that a functional interface can have multiple default methods (it can be said concrete methods which are default) but only one abstract method. The default method has been introduced in interface so that a new method can be appended in the class without affecting the implementing class of the existing interfaces. Because prior to java 8 the implementing class of an interface has to implement all the abstract methods defined in the interface.

The functional interface has been introduced in Java 8 to support the lambda expression in java 8 on the other hand it can be said lambda expression is the instance of functional interface.

Why Optional is used?

Optional is a container object which may or may not contain a non-null value. You must import ***java.util package*** to use this class. If a value is present, **isPresent()** will return true and **get()** will return the value. Additional methods that depend on the presence or absence of a contained value are provided, such as **orElse()** which returns a default value if the value is not present, and **ifPresent()** which executes a block of code if the value is present. This is a ***value-based*** class, i.e their instances are :

* Final and immutable (though may contain references to mutable objects).
* Considered equal solely based on equals(), not based on reference equality(==).
* Do not have accessible constructors.

**Static Methods:**Static methods are the methods in Java that can be called without creating an object of the class. They are referenced by the class name itself or reference to the object of that class.

We can also create an Optional object with the static method of():

**String** name = Optional.ofNullable(nullName).orElse("john");

**String** name = Optional.ofNullable(nullName).orElseGet(() -> "john");

## ****Difference Between****orElse****and****orElseGet()

To a lot of programmers who are new to Optional or Java 8, the difference between orElse() and orElseGet() is not clear. As a matter of fact, these two methods give the impression that they overlap each other in functionality.

However, there's a subtle but very important difference between the two that can affect the performance of our code drastically if not well understood.

Let's create a method called getMyDefault() in the test class, which takes no arguments and returns a default value:

**public** String **getMyDefault**() {

System.out.println("Getting Default Value");

**return** "Default Value";

}

Let's see two tests and observe their side effects to establish both where orElse() and orElseGet() overlap and where they differ:

@Test

**public** **void** **whenOrElseGetAndOrElseOverlap\_thenCorrect**() {

**String** text = null;

**String** defaultText = Optional.ofNullable(text).orElseGet(this::getMyDefault);

assertEquals("Default Value", defaultText);

defaultText = Optional.ofNullable(text).orElse(getMyDefault());

assertEquals("Default Value", defaultText);

}

In the above example, we wrap a null text inside an Optional object and attempt to get the wrapped value using each of the two approaches.

The side effect is:

Getting default value...

Getting default value...

The getMyDefault() method is called in each case. It so happens that **when the wrapped value is not present, then both orElse() and orElseGet() work exactly the same way.**

Now let's run another test where the value is present, and ideally, the default value should not even be created:

@Test

**public** **void** **whenOrElseGetAndOrElseDiffer\_thenCorrect**() {

**String** text = "Text present";

System.out.println("Using orElseGet:");

**String** defaultText

= Optional.ofNullable(text).orElseGet(this::getMyDefault);

assertEquals("Text present", defaultText);

System.out.println("Using orElse:");

defaultText = Optional.ofNullable(text).orElse(getMyDefault());

assertEquals("Text present", defaultText);

}

In the above example, we are no longer wrapping a null value, and the rest of the code remains the same.

Now let's take a look at the side effect of running this code:

Using orElseGet:

Using orElse:

Getting default value...

Notice that when using orElseGet() to retrieve the wrapped value, the getMyDefault() method is not even invoked since the contained value is present.

However, when using orElse(), whether the wrapped value is present or not, the default object is created. So in this case, we have just created one redundant object that is never used.

In this simple example, there is no significant cost to creating a default object, as the JVM knows how to deal with such. **However, when a method such as getMyDefault() has to make a web service call or even query a database, the cost becomes very obvious.**

What problem does default methods solved?

Diamond problem

As simple inheritance allows a child class to derive properties from one super-class. for example, if class B inherits properties from only one super-class A, then it is called simple inheritance, and Java supports them.

Multi-level inheritance allows a child class to inherit properties from a class that can inherit properties from some other classes. For example, class C can inherit its property from B class which itself inherits from A class. Java also supports them.

What Java does not allow is multiple inheritance where one class can inherit properties from more than one class. It is known as the **diamond problem**. In the above figure, we find that class D is trying to inherit form class B and class C, that is not allowed in Java.

It is an ambiguity that can rise as a consequence of allowing multiple inheritance. It is a serious problem for other OPPs languages. It is sometimes referred to as the **deadly diamond of death**.

**How to avoid Diamond Problem With Default Methods in Java 8**

The solution to the diamond problem is **default methods** and **interfaces**. We can achieve multiple inheritance by using these two things.

The default method is similar to the abstract method. The only difference is that it is defined inside the interfaces with the default implementation. We need not to override these methods. Because they are already implementing these interfaces.

The advantage of interfaces is that it can have the same default methods with the same name and signature in two different interfaces. It allows us to implement these two interfaces, from a class. We must override the default methods explicitly with its interface name.

interface MyInterface1{

   public static int num = 100;

   public default void display() {

      System.out.println("display method of MyInterface1");

   }

}

interface MyInterface2{

   public static int num = 1000;

   public default void display() {

      System.out.println("display method of MyInterface2");

   }

}

public class InterfaceExample implements MyInterface1, MyInterface2{

   public void display() {

      MyInterface1.super.display();

      //or,

      MyInterface2.super.display();

   }

   public static void main(String args[]) {

      InterfaceExample obj = new InterfaceExample();

      obj.display();

   }

}

**Spring Boot Starter Parent**

Spring Boot Starter Parent is a starter project that provides the default configuration for spring-based applications. It is added as a parent in the pom.xml file. The **spring-boot-starter-parent**defines **spring-boot-dependencies**as its parent. The **spring-boot-starter-parent** inherits dependency management from **spring-boot-dependencies**. Every Spring Boot release provides a list of dependencies and the latest versions it supports. The dependency management feature of the starter parent allows the common dependencies to omit the **<version>**tag in the pom.xml file.  If a dependency requires a specific version other than the version configured by the starter parent, that can be added using the **<version>**tag. The following are the features of the starter parent project:

* The dependency management feature manages the versions of common dependencies.
* Provide the default compiler level as Java 1.8 and UTF-8 source encoding.
* Provides a default configuration for Maven plugins such as maven-surefire-plugin, maven-jar-plugin, and maven-failsafe-plugin.
* Executes a **repackage**goal with a **repackage execution id**.
* Resource filtering and configuring profile-specific files.

# How Spring Boot Application Works Internally ?

# Spring Boot Application Internal Working.

Spring does not generate any code automatically and not using any xml configuration file . so spring uses internally pragmatically configuration done by spring boot developer that are provided by jar.  
we are using just pre-configured jar . and those jar available in:

**META-INF**/**spring.factories**

Enable  
Disable

To Enable preconfigured jars we just need to define dependency in pom.xml file.

*‘<’dependency’>’  
‘<’groupId’>’****org.springframework.boot’****<’/groupId’>’  
‘<’artifactId’>’****spring-boot-starter-data-jpa’****<’/artifactId’>’  
‘<’/dependency’>’*

This dependency will load all the jars related to JPA repository and stored into spring.factories.  
you can go to maven dependencies then click and open spring-boot-autoconfigure jar in the last you will see META-INF folder inside this spring.factories here you will find your jar org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration.

Based on **@Conditional**and **@Configuration** :

***@Configuration****(proxyBeanMethods = false)****@ConditionalOnBean****(DataSource.class)****@ConditionalOnClass****(JpaRepository.class)****@ConditionalOnMissingBean****({ JpaRepositoryFactoryBean.class, JpaRepositoryConfigExtension.class })****@ConditionalOnProperty****(prefix = “spring.data.jpa.repositories”, name = “enabled”, havingValue = “true”,  
matchIfMissing = true)****@Import****(JpaRepositoriesRegistrar.class)****@AutoConfigureAfter****({ HibernateJpaAutoConfiguration.class, TaskExecutionAutoConfiguration.class })  
public class JpaRepositoriesAutoConfiguration {*

*}*

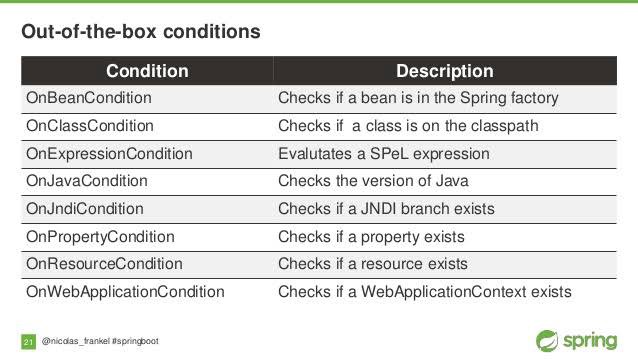
**@ConditionalOnBean**(DataSource.class) :  
— — — — — — — — — — — — — — —  
It will serach for the DataSource bean if it is available then only it will enable JpaRepositoriesAutoConfiguration . So this we need to define DataSource related properties into our property file.

**@ConditionalOnClass**(JpaRepository.class) :  
— — — — — — — — — — — — — — —  
It will serach for the JpaRepository class if it is available then only it will enable JpaRepositoriesAutoConfiguration .

like this :  
**@ConditionalOnMissingBean**({ JpaRepositoryFactoryBean.class, JpaRepositoryConfigExtension.class })  
**@ConditionalOnProperty**(prefix = “spring.data.jpa.repositories”, name = “enabled”, havingValue = “true”, matchIfMissing = true)

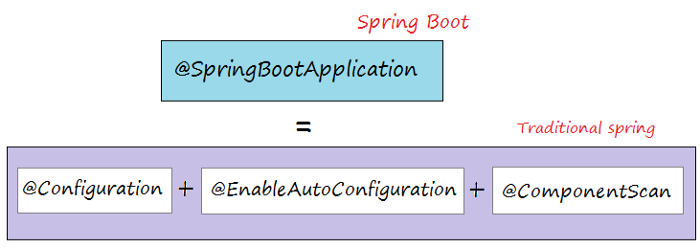
If all conditions are true then only it will enable JpaRepositoriesAutoConfiguration class.

**The mainly Conditions checked by spring boot :**



If all the conditions are satisfied then only spring will enable to the component.

**@SpringBootApplication**is the main annotation that we used on our main method and this annotation is the combination of these three annotations :



**High Level Flow Of Spring Boot And How run Method works :**  
================================

From the run method, the main application context is kicked off which in turn searches for the classes annotated with @Configuration, initializes all the declared beans in those configuration classes, and based upon the scope of those beans, stores those beans in JVM, specifically in a space inside JVM which is known as IOC container. After the creation of all the beans, automatically configures the dispatcher servlet and registers the default handler mappings, messageConverts, and all other basic things.

Basically, spring boot supports three embedded servers:- Tomcat (default), Jetty and Undertow.

**run() internal flow :**  
==========

**1.** create application context  
**2.** check Application Type  
**3.** Register the annotated class beans with the context  
**4.** Creates an instance of TomcatEmbeddedServletContainer : and adds the context. Used to deploy our jar automatically.

**open SpringApplication.class :**  
===============

And find here run(String… args) method inside this method you will see the method createApplicationContext() so first it will create application context and inside createApplicationContext() method it will check application type it is SERVLET type Or REACTIVE or DEFAULT context type based on this it will return context.

Now in DEFAULT\_CONTEXT\_CLASS you will see the class **AnnotationConfigApplicationContext.class**

*public****AnnotationConfigApplicationContext****(Class… annotatedClasses) {  
this();  
register(annotatedClasses);  
refresh();  
}*

open this class its constructor is used to Register the annotated class beans with the context.

The classes which are annotated with **@Component, @Service,** **@Configuration** etc. will be register to the context.

And in the finally run(-) method auto deploy the**jar/war** to server.

@SpringBootApplication is a convenience annotation that adds all of the following:

* @Configuration: Tags the class as a source of bean definitions for the application context.
* @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.
* @ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.

It is the main class that we need to define to make our spring boot application.

[***@SpringBootApplicatio***](http://twitter.com/SpringBootApplicatio)***n*** *public class Application****{*** *public static void main(String[] args) {  
SpringApplication.run(Application.class, args);  
}****}***

If we will open **@SprinBootApplication** Annotation here you will see it contains :

[***@SpringBootConfigurat***](http://twitter.com/SpringBootConfigurat)***ion***[***@EnableAutoConfigurat***](http://twitter.com/EnableAutoConfigurat)***ion***[***@ComponentScan***](http://twitter.com/ComponentScan)*(excludeFilters = {*[***@Filter***](http://twitter.com/Filter)*(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),*[***@Filter***](http://twitter.com/Filter)*(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })  
public*[*@interface*](http://twitter.com/interface)*SpringBootApplication {  
// code here…..  
}*

What is lazy loading in streams?

Streams are lazy because **intermediate operations are not evaluated unless terminal operation** is invoked. Each intermediate operation creates a new stream, stores the provided operation/function and return the new stream. The pipeline accumulates these newly created streams

The time when terminal operation is called, traversal of streams begins and the associated function is performed one by one.

### Java 8 Stream Intermediate And Terminal Operations :

1) The main difference between intermediate and terminal operations is that intermediate operations return a stream as a result and terminal operations return non-stream values like primitive or object or collection or may not return anything.

2) As intermediate operations return another stream as a result, they can be chained together to form a pipeline of operations. Terminal operations can not be chained together.

3) Pipeline of operations may contain any number of intermediate operations, but there has to be only one terminal operation, that too at the end of pipeline.

4) Intermediate operations are lazily loaded. When you call intermediate operations, they are actually not executed. They are just stored in the memory and executed when the terminal operation is called on the stream.

5) As the names suggest, intermediate operations doesn’t give end result. They just transform one stream to another stream. On the other hand, terminal operations give end result.

What is lazy and eager mode in hibernate?

* **Eager Loading** is a design pattern in which data initialization occurs on the spot.
* **Lazy Loading** is a design pattern that we use to defer initialization of an object as long as it's possible.
* The main difference between the two types of fetching is the moment when data gets loaded into a memory.

List<UserLazy> users = sessionLazy.createQuery("From UserLazy").list();

**UserLazy** userLazyLoaded = users.get(3);

**return** (userLazyLoaded.getOrderDetail());

With the lazy initialization approach, *orderDetailSet* will get initialized only when we explicitly call it, using a getter or some other method:

**UserLazy** userLazyLoaded = users.get(3);

But with an eager approach in *UserEager*, it will be initialized immediately in the first line:

List<UserEager> user = sessionEager.createQuery("From UserEager").list();

For lazy loading, we use a proxy object and fire a separate SQL query to load the *orderDetailSet*.

The idea of disabling proxies or lazy loading is considered a bad practice in Hibernate. It can result in fetching and storing a lot of data, irrespective of the need for it.

**Web Services**

A web service is, in short, a machine-to-machine, platform independent service that allows communication over a network.

**SOAP**

SOAP is a messaging protocol. Messages (requests and responses) are **XML documents over HTTP**. **The XML contract is defined by the WSDL** (Web Services Description Language). It provides a set of rules to define the messages, bindings, operations, and location of the service.

SOAP defines a specific xml request and response.

SOAP-ENV: Envelope, Header and Body.

Transport: HTTP or MQ

Service Definition: WSDL

WSDL defines endpoint, all the operations. It specifies request and response structure.

**REST**

REST stands for Representational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods.

Request Header and Body and Methods

Response – status codes

URI to each resource.

* SOAP stands for **S**imple **O**bject **A**ccess **P**rotocol and REST stands for **RE**presentational **S**tate **T**ransfer.
* Since SOAP is a protocol, it follows a strict standard to allow communication between the client and the server whereas REST is an architectural style that doesn’t follow any strict standard but follows six constraints defined by Roy Fielding in 2000. Those constraints are – Uniform Interface, Client-Server, Stateless, Cacheable, Layered System, Code on Demand.
* SOAP uses only XML for exchanging information in its message format whereas REST is not restricted to XML and its the choice of implementer which Media-Type to use like XML, JSON, Plain-text. Moreover, REST can use SOAP protocol but SOAP cannot use REST.
* On behalf of services interfaces to business logic, SOAP uses @WebService whereas REST instead of using interfaces uses URI like @Path.
* SOAP is difficult to implement and it requires more bandwidth whereas REST is easy to implement and requires less bandwidth such as smartphones.
* Benefits of SOAP over REST as SOAP has ACID compliance transaction. Some of the applications require transaction ability which is accepted by SOAP whereas REST lacks in it.
* On the basis of Security, SOAP has SSL( **S**ecure **S**ocket **L**ayer) and WS-security whereas REST has SSL and HTTPS. In the case of Bank Account Password, Card Number, etc. SOAP is preferred over REST. The security issue is all about your application requirement, you have to build security on your own. It’s about what type of protocol you use.
* SOAP cannot make use of REST since SOAP is a protocol without any architectural pattern. REST can make use of SOAP because it is an architectural pattern having protocol.

|  |  |
| --- | --- |
| **SOAP** | **REST** |
| * SOAP stands for Simple Object Access Protocol | * REST stands for Representational State Transfer |
| * SOAP is a protocol. SOAP was designed with a specification. It includes a WSDL file which has the required information on what the web service does in addition to the location of the web service. | * REST is an Architectural style in which a web service can only be treated as a RESTful service if it follows the constraints of being   1. Client Server   2. Stateless   3. Cacheable   4. Layered System   5. Uniform Interface |
| * SOAP cannot make use of REST since SOAP is a protocol and REST is an architectural pattern. | * REST can make use of SOAP as the underlying protocol for web services, because in the end it is just an architectural pattern. |
| * SOAP uses service interfaces to expose its functionality to client applications. In SOAP, the WSDL file provides the client with the necessary information which can be used to understand what services the web service can offer. | * REST use Uniform Service locators to access to the components on the hardware device. For example, if there is an object which represents the data of an employee hosted on a URL as http://demo.guru99 , the below are some of URI that can exist to access them.   http://demo.guru99.com/Employee  http://demo.guru99.com/Employee/1 |
| * SOAP requires more bandwidth for its usage. Since SOAP Messages contain a lot of information inside of it, the amount of data transfer using SOAP is generally a lot.   <?xml version="1.0"?>  <SOAP-ENV:Envelope  xmlns:SOAP-ENV  ="http://www.w3.org/2001/12/soap-envelope"  SOAP-ENV:encodingStyle  =" http://www.w3.org/2001/12/soap-encoding">  <soap:Body>  <Demo.guru99WebService  xmlns="http://tempuri.org/">  <EmployeeID>int</EmployeeID>  </Demo.guru99WebService>  </soap:Body>  </SOAP-ENV:Envelope> | * REST does not need much bandwidth when requests are sent to the server. REST messages mostly just consist of JSON messages. Below is an example of a JSON message passed to a web server. You can see that the size of the message is comparatively smaller to SOAP.   {"city":"Mumbai","state":"Maharastra"} |
| * SOAP can only work with XML format. As seen from SOAP messages, all data passed is in XML format. | * REST permits different data format such as Plain text, HTML, XML, JSON, etc. But the most preferred format for transferring data is JSON. |

### How is WSDL related to SOAP?

You’ve probably seen the term WSDL alongside SOAP. So how are the two terms related?

[SOAP](https://www.w3.org/TR/soap/) is a standard for exchanging messages with a server, or making a remote procedure call (RPC). SOAP sits on top of an existing transport, like HTTP.

WSDL allows us to define:

* The operations you can call on a service
* The messages that you can exchange with a service

But SOAP defines:

* The messages themselves – an Envelope containing a Header and a Body
* A way to return fault or error messages from the service
* A way to enhance messages with other features, like [encryption](https://en.wikipedia.org/wiki/WS-Security)

So WSDL defines the operations and messages for a web service, and SOAP defines the concrete format for sending those messages to a server.

## Structure of a WSDL file

A WSDL document is written in XML and contains the following elements:

| **ELEMENT** | **WHAT IT DOES** |
| --- | --- |
| <types> | Defines the data types (XML elements) that are used by the web service. |
| <message> | Defines the messages that can be exchanged with the web service. Each <message> contains a <part>. |
| <portType> | Defines each operation in the web service, and the messages associated with each operation. |
| <binding> | Defines exactly how each operation will take place over the network (we use SOAP, in the example below). |
| <service> | Defines the physical location of the service (e.g. its **endpoint**). |

**Rest Principles**

1. **Uniform interface**. All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). Resources shouldn’t be too large but should contain every piece of information that the client might need.
2. **Client-server decoupling**. In REST API design, client and server applications must be completely independent of each other. The only information the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application shouldn't modify the client application other than passing it to the requested data via HTTP.
3. **Statelessness**. REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it. In other words, REST APIs do not require any server-side sessions. Server applications aren’t allowed to store any data related to a client request.
4. **Cacheability**. When possible, resources should be cacheable on the client or server side. Server responses also need to contain information about whether caching is allowed for the delivered resource. The goal is to improve performance on the client side, while increasing scalability on the server side.
5. **Layered system architecture**. In REST APIs, the calls and responses go through different layers. As a rule of thumb, don’t assume that the client and server applications connect directly to each other. There may be a number of different intermediaries in the communication loop. REST APIs need to be designed so that neither the client nor the server can tell whether it communicates with the end application or an intermediary.
6. **Code on demand (optional)**. REST APIs usually send static resources, but in certain cases, responses can also contain executable code (such as Java applets). In these cases, the code should only run on-demand.

### SOAP (Simple Object Access Protocol):

SOAP is a method of transferring messages, or small amounts of information, over the Internet. SOAP messages are formatted in XML and are typically sent using HTTP (hypertext transfer protocol).

|  |
| --- |
| SOAP uses WSDL for communication between consumer and provider, whereas REST just uses XML or JSON to send and receive data. |

* WSDL defines contract between client and service and is static by its nature.
* SOAP builds an XML based protocol on top of [HTTP](https://crunchify.com/java-how-to-get-entity-contenttype-in-httpclient/) or sometimes TCP/IP.
* SOAP describes functions, and types of data.
* SOAP is a successor of XML-RPC and is very similar, but describes a standard way to communicate.
* Several programming languages have native support for SOAP, you typically feed it a [web service](https://crunchify.com/create-and-deploy-simple-web-service-and-web-service-client-in-eclipse/) URL and you can call its web service functions without the need of specific code.
* Binary data that is sent must be encoded first into a format such as base64 encoded.
* Has several protocols and technologies relating to it: WSDL, XSDs, SOAP,

REST (REpresentational State Transfer):

Rest is a simple way of sending and receiving data between client and server and it doesn’t have very many standards defined. You can send and receive data as [JSON](https://crunchify.com/category/json/), [XML](https://crunchify.com/tag/pom-xml/) or even plain text. It’s lightweight compared to SOAP.

|  |
| --- |
| REST need not be over HTTP but most of my points below will have an HTTP bias. |

* In case of REST contract between client and service is somewhat complicated and is defined by HTTP, URI, Media Formats and Application Specific Coordination Protocol. It’s highly dynamic unlike WSDL.
* REST is very lightweight, it says wait a minute, we don’t need all of this complexity that SOAP created.
* Typically uses normal [HTTP](https://crunchify.com/how-to-send-http-request-and-capture-response-in-java/) methods instead of a big XML format describing everything. For example to obtain a resource you use HTTP GET, to put a resource on the server you use HTTP PUT. To delete a resource on the server you use HTTP DELETE.
* REST is a very simple in that it uses HTTP GET, POST and PUT methods to update resources on the server.
* REST typically is best used with Resource Oriented Architecture (ROA). In this mode of thinking everything is a resource, and you would operate on these resources.
* As long as your programming language has an HTTP library, and most do, you can consume a REST [HTTP protocol](https://crunchify.com/simple-way-to-get-http-response-header-in-java/) very easily.
* Binary data or binary resources can simply be delivered upon their request.

### SOAP 2.0 WSDL Example:

|  |
| --- |
| <!-- Crunchify's example of WSDL 2.0 -->    <?xml version="1.0" encoding="UTF-8"?>  <definitions xmlns="http://www.w3.org/ns/wsdl"               xmlns:tns="http://www.tmsws.com/wsdl20sample"               xmlns:whttp="http://schemas.xmlsoap.org/wsdl/http/"               xmlns:wsoap="http://schemas.xmlsoap.org/wsdl/soap/"               targetNamespace="http://www.tmsws.com/wsdl20sample">    <!-- Abstract type -->     <types>        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"                  xmlns="http://www.tmsws.com/wsdl20sample"                  targetNamespace="http://crunchify.com/wsdl20">             <xs:element name="request"> ...add your code here... </xs:element>           <xs:element name="response">  ...add your code here... </xs:element>        </xs:schema>     </types>    <!-- Abstract interfaces -->     <interface name="CrunchifyItem">        <fault name="ErrorDescription" element="tns:response"/>        <operation name="Get" pattern="http://www.w3.org/ns/wsdl/in-out">           <input messageLabel="In" element="tns:request"/>           <output messageLabel="Out" element="tns:response"/>        </operation>     </interface>    <!-- Concrete Binding Over HTTP -->     <binding name="HttpBinding" interface="tns:CrunchifyItem"              type="http://www.w3.org/ns/wsdl/http">        <operation ref="tns:Get" whttp:method="GET"/>     </binding>    <!-- Concrete Binding with SOAP-->     <binding name="SoapBinding" interface="tns:CrunchifyItem"              type="http://www.w3.org/ns/wsdl/soap"              wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"              wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">        <operation ref="tns:Get" />     </binding>    <!-- Web Service offering endpoints for both bindings-->     <service name="ServiceOne" interface="tns:CrunchifyItem">        <endpoint name="HttpEndpoint"                  binding="tns:HttpBinding"                  address="http://www.example.com/rest/"/>        <endpoint name="SoapEndpoint"                  binding="tns:SoapBinding"                  address="http://crunchify.com/soap/"/>     </service>  </definitions> |

For Consuming SOAP webservices, we need configuration for Marsheller and WebServiceTemplate.

Spring Boot goals

Quick start of Application

Opiniated – Sprint Boot makes lot of assumptions. It takes an opinion, but we can override.

Monitoring and error handling

Features

Autoconfigure

Starter projects

Actuator – monitor app

Embedded server – tomcat.

Dependency injection – loose coupling

# Marker interface in Java

It is an empty interface (no field or methods). Examples of marker interface are Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces.

It **provides run-time type information about objects, so the compiler and JVM have additional information about the object**.

**Cloneable interface** : Cloneable interface is present in java.lang package. There is a method clone() in [Object](https://www.geeksforgeeks.org/object-class-in-java/) class. A class that implements the Cloneable interface indicates that it is legal for clone() method to make a field-for-field copy of instances of that class.   
Invoking Object’s clone method on an instance of the class that does not implement the Cloneable interface results in an exception CloneNotSupportedException being thrown. By convention, classes that implement this interface should override Object.clone() method.

**Serializable interface** : Serializable interface is present in java.io package. It is used to make an object eligible for saving its state into a file. This is called [Serialization](https://www.geeksforgeeks.org/serialization-in-java/).   
Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable.

**Remote interface** : Remote interface is present in java.rmi package. A remote object is an object which is stored at one machine and accessed from another machine. So, to make an object a remote object, we need to flag it with Remote interface. Here, Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. RMI ([Remote Method Invocation](https://www.geeksforgeeks.org/remote-method-invocation-in-java/)) provides some convenience classes that remote object implementations can extend which facilitate remote object creation.

Abstract class

Java Abstract class **can implement interfaces without even providing the implementation of interface methods**. Java Abstract class is used to provide common method implementation to all the subclasses or to provide default implementation.

An instance of an abstract class cannot be created, we can have references to abstract class type though.

And a constructor of abstract class is called when an instance of an inherited class is created.

In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated but can only be inherited.

Abstract classes can also have final methods (methods that cannot be overridden)

For any abstract java class we are not allowed to create an object i.e., for abstract class instantiation is not possible.

Similar to the interface we can define static methods in an abstract class that can be called independently without an object.

Why do we need abstract classes?

An abstract class can be used as a type of template for other classes. The abstract class will hold common functionality for all classes that extend it.

All animals move and breathe and reproduce so these can be put into the Animal Class.

# **Interface in Java**

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve [*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

Abstract class in java –difference with interface

### **Abstract class vs Interface**

* **Type of methods:** Interface can have only abstract methods. An abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
* **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
* **Type of variables:**Abstract class can have final, non-final, static and non-static variables. The interface has only static and final variables.
* **Implementation:** Abstract class can provide the implementation of the interface. Interface can’t provide the implementation of an abstract class.
* **Inheritance vs Abstraction:** A Java interface can be implemented using the keyword “implements” and an abstract class can be extended using the keyword “extends”.
* **Multiple implementations:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
* **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

When to use abstract class and Interface?

Consider using abstract classes if any of these statements apply to your situation:

* In the java application, there are some related classes that need to share some lines of code then you can put these lines of code within the abstract class and this abstract class should be extended by all these related classes.
* You can define the non-static or non-final field(s) in the abstract class so that via a method you can access and modify the state of the object to which they belong.
* You can expect that the classes that extend an abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

Consider using interfaces if any of these statements apply to your situation:

* It is total abstraction, All methods declared within an interface must be implemented by the class(es) that implements this interface.
* A class can implement more than one interface. It is called multiple inheritances.
* You want to specify the behavior of a particular data type but are not concerned about who implements its behavior.

MultiLevel and Hierarchial inheritance

### 1) Single Inheritance

**Single inheritance** is damn easy to understand. When a class extends another one class only then we  call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be  a **child class** of A.

### 2) Multiple Inheritance

“**Multiple Inheritance**” refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with “multiple inheritance” is that the derived class will have to manage the dependency on two base classes.

### 3) Multilevel Inheritance

**Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A. For more details and example refer – [Multilevel inheritance in Java](https://beginnersbook.com/2013/12/multilevel-inheritance-in-java-with-example/).

### 4) Hierarchical Inheritance

In such kind of inheritance one class is inherited by many**sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.

### 5) Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of**Single** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.

can we override public static void main in java

**You cannot override static methods** and since the public static void main() method is static we cannot override it.

what is the top most parent class in java

Object

is object creation necessary in java

static methods are against OOP. It's not necessary to call a constructor to instantiate objects. Constructors are merely a Java design pattern, but they **have nothing to do with actually creating the instance**. Java is no magic — no programming language is.

Constructor chaining in java

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

Constructor chaining occurs through **inheritance**. A sub class constructor’s task is to call super class’s constructor first. This ensures that creation of sub class’s object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.  
**Why do we need constructor chaining ?**   
This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

different constructors in java

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

### **What is the purpose of a default constructor?**

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

### **Java Parameterized Constructor**

A constructor which has a specific number of parameters is called a parameterized constructor.

### **Why use the parameterized constructor?**

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

## Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Limitation of inheritance

Main disadvantage of using inheritance is that **the two classes (parent and child class) gets tightly coupled**. This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, it cannot be independent of each other.

Dynamic Polymorphism

Method Overriding

Why do we need constructors in java

The purpose of a Java constructor is **to initializes the newly created object before it is used**. ... Typically, the constructor initializes the fields of the object that need initialization. Java constructors can also take parameters, so fields can be initialized in the object at creation time

Can we create constructor of abstract class

**Yes!** **Abstract classes can have constructors**! Yes, when we define a class to be an Abstract Class it cannot be instantiated but that does not mean an Abstract class cannot have a constructor. Each abstract class must have a concrete subclass which will implement the abstract methods of that abstract class.

* If you define your own constructor with arguments inside an abstract class but forget to call your own constructor inside its derived class constructor then JVM will call the constructor by default.
* So if you define your single or multi-argument constructor inside the abstract class then make sure to call the constructor inside the drive class constructor with the [super keyword](https://www.geeksforgeeks.org/super-keyword/).

# **Singleton design pattern in Java**

Singleton Pattern says that just**"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required

#### **Advantage of Singleton design pattern**

* Saves memory because object is not created at each request. Only single instance is reused again and again.

#### **Usage of Singleton design pattern**

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

#### **How to create Singleton design pattern?**

To create the singleton class, we need to have static member of class, private constructor and static factory method.

**Overcome Singleton destruction**

**Overcome reflection issue:** To overcome issue raised by reflection, [enums](https://www.geeksforgeeks.org/enum-in-java/) are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

**Overcome serialization issue:-** To overcome this issue, we have to implement method readResolve() method.

**Cloning:** [Cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/) is a concept to create duplicate objects. Using clone we can create copy of object. Suppose, we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

**Collections**

List

ArrayList

*List* represents an ordered sequence of values where some value may occur more than one time.

*ArrayList* is one of the *List* implementations built atop an array, which is able to dynamically grow and shrink as you add/remove elements. Elements could be easily accessed by their indexes starting from zero. This implementation has the following properties:

* Random access takes *O(1)* time
* Adding element takes amortized constant time *O(1)*
* Inserting/Deleting takes *O(n)* time
* Searching takes *O(n)* time for unsorted array and *O(log n)* for a sorted one

Use case:

ArrayList provides constant time for search operation, so it is better to use ArrayList **if searching is more frequent operation than add and remove operation**.

*ArrayList*is usually the default *List* implementation.

However, there are certain use cases where using *LinkedList* will be a better fit, such as preferences for constant insertion/deletion time (e.g., frequent insertions/deletions/updates), over constant access time and effective memory usage.

The JDK provides a nice way to get an unmodifiable collection out of an existing one:

Collections.unmodifiableList(list);

LinkedList

Linked List is a part of the [Collection framework](https://www.geeksforgeeks.org/collections-in-java-2/) present in [java.util package](https://www.geeksforgeeks.org/java-util-package-java/). This class is an implementation of the [LinkedList data structure](https://www.geeksforgeeks.org/data-structures/linked-list/) which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has a few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access.

* Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index
* It is not [**synchronized**](https://stackoverflow.com/a/1085745/2486904)
* Its [***Iterator***](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Iterator.html) and [***ListIterator***](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ListIterator.html) iterators are [**fail-fast**](https://stackoverflow.com/questions/17377407/what-is-fail-safe-fail-fast-iterators-in-java-how-they-are-implemented) (which means that after the iterator's creation, if the list is modified, a [***ConcurrentModificationException***](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ConcurrentModificationException.html) will be thrown)
* Every element is a node, which keeps a reference to the next and previous ones
* It maintains insertion order
* A LinkedList stores its data as a list of elements and every element is linked to its previous and next element. In this case, the search operation for an item has execution time equal to O(n).
* A LinkedList consumes more memory than an ArrayList because of every node in a LinkedList stores two references, one for its previous element and one for its next element, whereas ArrayList holds only data and its index.

The insertion, addition and removal operations of an item are faster in a LinkedList because there is no need to resize an array or update the index when an element is added to some arbitrary position inside the collection, only references in surrounding elements will change.

## Search Operation

Search Operation in **ArrayList** is pretty fast when compared to the **LinkedList** search operation. ArrayList Method get(int index) gives the performance of O(1) while LinkedList performance is O(n). This is because ArrayList allows random access to the elements in the list as it operates on an **index-based** data structure while LinkedList does not allow **random access** as it does not have indexes to access elements directly, it has to traverse the list to retrieve or access an element from the list.

## Manipulation

Manipulation with **ArrayList** is slow because it internally uses array. If We need to insert or delete element in ArrayList, it may take O(n), as it internally uses array and we may have to shift elements in case of insertion or deletion. On the other hand manipulation with **LinkedList** is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory. If We need to insert or delete element in LinkedList, it will take O(1), as it internally uses doubly.

CopyOnWriteArryList

The design of the *CopyOnWriteArrayList*uses an interesting technique to make it thread-safe without a need for synchronization. When we are using any of the modify methods – such as *add()* or *remove() –*the whole content of the *CopyOnWriteArrayList* is copied into the new internal copy.

Due to this simple fact, **we can iterate over the list in a safe way, even when concurrent modification is happening**.

When we're calling the *iterator()*method on the *CopyOnWriteArrayList,*we get back an *Iterator* backed up by the immutable snapshot of the content of the *CopyOnWriteArrayList*.

Its content is an exact copy of data that is inside an *ArrayList* from the time when the *Iterator* was created. Even if in the meantime some other thread adds or removes an element from the list, that modification is making a fresh copy of the data that will be used in any further data lookup from that list.

The characteristics of this data structure make it particularly useful in cases when we are iterating over it more often than we are modifying it. If adding elements is a common operation in our scenario, then *CopyOnWriteArrayList*won't be a good choice – because the additional copies will definitely lead to sub-par performance.

The design of the *CopyOnWriteArrayList* uses an interesting technique to make it thread-safe without a need for synchronization. When we are using any of the modify methods – such as *add()* or *remove() –* the whole content of the *CopyOnWriteArrayList* is copied into the new internal copy.

Due to this simple fact, **we can iterate over the list in a safe way, even when concurrent**

The *CopyOnWriteArrayList*was created to allow for the possibility of safe iterating over elements even when the underlying list gets modified.

Because of the copying mechanism, the *remove()*operation on the returned *Iterator*is not permitted – resulting with *UnsupportedOperationException:*

CopyOnWriteArrayList<String> list

            = new CopyOnWriteArrayList<>();

        // Initial Iterator

        Iterator itr = list.iterator();

        list.add("GfG");

        System.out.println("List contains: ");

        while (itr.hasNext())

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

        // iterator after adding an element

        itr = list.iterator();

        System.out.println("List contains:");

        while (itr.hasNext())

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

Notes:

List<Integer> intList = Lists.newArrayList(1, 2, 3, 4, 5, 6, 7, 8); Map<Boolean, List<Integer>> groups = intList.stream().collect(Collectors.partitioningBy(s -> s > 6)); List<List<Integer>> subSets = **new** **ArrayList**<List<Integer>>(groups.values());

List<Integer> list = Lists.newArrayList(null, 1, null); **while** (list.remove(null));

List<Integer> listWithDuplicates = Lists.newArrayList(5, 0, 3, 1, 2, 3, 0, 0); List<Integer> listWithoutDuplicates = **new** **ArrayList**<>( **new** **HashSet**<>(listWithDuplicates));

Assert.assertEquals(list1, list2); Assert.assertNotSame(list1, list2); Assert.assertNotEquals(list1, list3);

**void** **removeAll**(List<Integer> list, Integer element) { **int** index; **while** ((index = list.indexOf(element)) >= 0) { list.remove(index); } }

HashMap

**Let's first look at what it means that *HashMap* is a map. A map is a key-value mapping, which means that every key is mapped to exactly one value and that we can use the key to retrieve the corresponding value from a map.**

One might ask why not simply add the value to a list. Why do we need a *HashMap*? The simple reason is performance. If we want to find a specific element in a list, the time complexity is *O(n)* and if the list is sorted, it will be *O(log n)* using, for example, a binary search.

The advantage of a *HashMap* is that the time complexity to insert and retrieve a value is *O(1)* on average. We'll look at how that can be achieved later. Let's first look at how to use *HashMap*.

## HashMap Internals

In this section, we'll look at how HashMap works internally and what are the benefits of using HashMap instead of a simple list, for example.

As we've seen, we can retrieve an element from a HashMap by its key. One approach would be to use a list, iterate over all elements, and return when we find an element for which the key matches. Both the time and space complexity of this approach would be O(n).

**With HashMap, we can achieve an average time complexity of O(1) for the put and get operations and space complexity of O(n).** Let's see how that works.

### 5.1. The Hash Code and Equals

**Instead of iterating over all its elements, HashMap attempts to calculate the position of a value based on its key.**

The naive approach would be to have a list that can contain as many elements as there are keys possible. As an example, let's say our key is a lower-case character. Then it's sufficient to have a list of size 26, and if we want to access the element with key ‘c', we'd know that it's the one at position 3, and we can retrieve it directly.

However, this approach would not be very effective if we have a much bigger keyspace. For example, let's say our key was an integer. In this case, the size of the list would have to be 2,147,483,647. In most cases, we would also have far fewer elements, so a big part of the allocated memory would remain unused.

**HashMap stores elements in so-called buckets and the number of buckets is called capacity.**

When we put a value in the map, the key's hashCode() method is used to determine the bucket in which the value will be stored.

To retrieve the value, HashMap calculates the bucket in the same way – using hashCode(). Then it iterates through the objects found in that bucket and use key's equals() method to find the exact match.

### 5.2. Keys' Immutability

**In most cases, we should use immutable keys. Or at least, we must be aware of the consequences of using mutable keys.**

Let's see what happens when our key changes after we used it to store a value in a map.

For this example, we'll create the MutableKey:

**public** **class** **MutableKey** {

**private** String name;

// standard constructor, getter and setter

@Override

**public** **boolean** **equals**(Object o) {

**if** (this == o) {

**return** true;

}

**if** (o == null || getClass() != o.getClass()) {

**return** false;

}

**MutableKey** that = (MutableKey) o;

**return** Objects.equals(name, that.name);

}

@Override

**public** **int** **hashCode**() {

**return** Objects.hash(name);

}

}

And here goes the test:

**MutableKey** key = **new** **MutableKey**("initial");

Map<MutableKey, String> items = **new** **HashMap**<>();

items.put(key, "success");

key.setName("changed");

assertNull(items.get(key));

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + id;

result = prime \* result + ((name == null) ? 0 : name.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;

DataKey other = (DataKey) obj;

if (id != other.id)

return false;

if (name == null) {

if (other.name != null)

return false;

} else if (!name.equals(other.name))

return false;

return true;

}

As we can see, we're no longer able to get the corresponding value once the key has changed, instead, null is returned. **This is because HashMap is searching in the wrong bucket.**

The above test case may be surprising if we don't have a good understanding of how HashMap works internally.

### 5.3. Collisions

**For this to work correctly, equal keys must have the same hash, however, different keys can have the same hash**. If two different keys have the same hash, the two values belonging to them will be stored in the same bucket. Inside a bucket, values are stored in a list and retrieved by looping over all elements. The cost of this is O(n).

As of Java 8 (see [JEP 180](https://openjdk.java.net/jeps/180)), the data structure in which the values inside one bucket are stored is changed from a list to a balanced tree if a bucket contains 8 or more values, and it's changed back to a list if, at some point, only 6 values are left in the bucket. This improves the performance to be O(log n).

### 5.4. Capacity and Load Factor

To avoid having many buckets with multiple values, the capacity is doubled if 75% (the load factor) of the buckets become non-empty. The default value for the load factor is 75%, and the default initial capacity is 16. Both can be set in the constructor.

TreeMap

TreeMap is a map implementation that keeps its entries sorted according to the natural ordering of its keys or better still using a comparator if provided by the user at construction time.

# **JEP 180: Handle Frequent HashMap Collisions with Balanced Trees**

The principal idea is that once the number of items in a hash bucket grows beyond a certain threshold, that bucket will switch from using a linked list of entries to a balanced tree. In the case of high hash collisions, this will improve worst-case performance from O(n) to O(log n).

This technique has already been implemented in the latest version of the java.util.concurrent.ConcurrentHashMap class, which is also slated for inclusion in JDK 8 as part of [JEP 155](https://bugs.openjdk.java.net/browse/JDK-8046145). Portions of that code will be re-used to implement the same idea in the HashMap and LinkedHashMap classes. Only the implementations will be changed; no interfaces or specifications will be modified. Some user-visible behaviors, such as iteration order, will change within the bounds of their current specifications.

## ****Internal Implementation of****TreeMap

TreeMap implements NavigableMap interface and bases its internal working on the principles of [red-black trees](https://www.baeldung.com/cs/red-black-trees):

**public** **class** **TreeMap**<K,V> **extends** **AbstractMap**<K,V>

**implements** **NavigableMap**<K,V>, Cloneable, java.io.Serializable

The principle of red-black trees is beyond the scope of this article, however, there are key things to remember in order to understand how they fit into TreeMap.

**First of all**, a red-black tree is a data structure that consists of nodes; picture an inverted mango tree with its root in the sky and the branches growing downward. The root will contain the first element added to the tree.

The rule is that starting from the root, any element in the left branch of any node is always less than the element in the node itself. Those on the right are always greater. What defines greater or less than is determined by the natural ordering of the elements or the defined comparator at construction as we saw earlier.

This rule guarantees that the entries of a treemap will always be in sorted and predictable order.

**Secondly**, a red-black tree is a self-balancing binary search tree. This attribute and the above guarantee that basic operations like search, get, put and remove take logarithmic time O(log n).

Being self-balancing is key here. As we keep inserting and deleting entries, picture the tree growing longer on one edge or shorter on the other.

This would mean that an operation would take a shorter time on the shorter branch and longer time on the branch which is furthest from the root, something we would not want to happen.

Therefore, this is taken care of in the design of red-black trees. For every insertion and deletion, the maximum height of the tree on any edge is maintained at O(log n) i.e. the tree balances itself continuously.

## ****Choosing the Right Map****

Having looked at [HashMap](https://www.baeldung.com/java-hashmap) and [LinkedHashMap](https://www.baeldung.com/java-linked-hashmap) implementations previously and now TreeMap, it is important to make a brief comparison between the three to guide us on which one fits where.

**A hash map** is good as a general-purpose map implementation that provides rapid storage and retrieval operations. However, it falls short because of its chaotic and unorderly arrangement of entries.

This causes it to perform poorly in scenarios where there is a lot of iteration as the entire capacity of the underlying array affects traversal other than just the number of entries.

**A linked hash map** possesses the good attributes of hash maps and adds order to the entries. It performs better where there is a lot of iteration because only the number of entries is taken into account regardless of capacity.

**A tree map** takes ordering to the next level by providing complete control over how the keys should be sorted. On the flip side, it offers worse general performance than the other two alternatives.

We could say a**linked hash map reduces the chaos in the ordering of a hash map without incurring the performance penalty of a tree map**.

* We should use a *TreeMap*if we want to keep our entries sorted
* We should use a *HashMap*if we prioritize performance over memory consumption
* Since a *TreeMap* has a more significant locality, we might consider it if we want to access objects that are relatively close to each other according to their natural ordering
* *HashMap*can be tuned using the *initialCapacity* and *loadFactor*, which isn't possible for the *TreeMap*
* We can use the *LinkedHashMap*if we want to preserve insertion order while benefiting from constant time access

HashSet

* It stores unique elements and permits nulls
* It's backed by a *HashMap*
* It doesn't maintain insertion order
* It's not thread-safe

## WeakHashMap****as an Efficient Memory Cache****

Let's say that we want to build a cache that keeps big image objects as values, and image names as keys. We want to pick a proper map implementation for solving that problem.

Using a simple HashMap will not be a good choice because the value objects may occupy a lot of memory. What's more, they'll never be reclaimed from the cache by a GC process, even when they are not in use in our application anymore.

Ideally, we want a Map implementation that allows GC to automatically delete unused objects. When a key of a big image object is not in use in our application in any place, that entry will be deleted from memory.

Fortunately, the WeakHashMap has exactly these characteristics. Let's test our WeakHashMap and see how it behaves:

We're creating a *WeakHashMap*instance that will store our *BigImage*objects. We are putting a *BigImage*object as a value and an *imageName* object reference as a key. The *imageName*will be stored in a map as a *WeakReference* type.

Next, we set the *imageName* reference to be *null*, therefore there are no more references pointing to the *bigImage* object. The default behavior of a *WeakHashMap*is to reclaim an entry that has no reference to it on next GC, so this entry will be deleted from memory by the next GC process.

We are calling a *System.gc()*to force the JVM to trigger a GC process. After the GC cycle, our *WeakHashMap*will be empty:

## ****How****HashSet****Maintains Uniqueness?****

When we put an object into a HashSet, it uses the object's hashcode value to determine if an element is not in the set already.

Each hash code value corresponds to a certain bucket location which can contain various elements, for which the calculated hash value is the same. **But two objects with the same hashCode might not be equal**.

So, objects within the same bucket will be compared using the equals() method.

## ****Performance of****HashSet

The performance of a HashSet is affected mainly by two parameters – its Initial Capacity and the Load Factor.

The expected time complexity of adding an element to a set is O(1) which can drop to O(n) in the worst case scenario (only one bucket present) – therefore, **it's essential to maintain the right HashSet's capacity.**

An important note: since JDK 8, [the worst case time complexity is O(log\*n).](https://openjdk.java.net/jeps/180)

The load factor describes what is the maximum fill level, above which, a set will need to be resized.

We can also create a HashSet with custom values for initial capacity and load factor:

TreeSet

* It stores unique elements
* It doesn't preserve the insertion order of the elements
* It sorts the elements in ascending order
* It's not thread-safe

**In this implementation, objects are sorted and stored in ascending order according to their natural order**. The TreeSet uses a self-balancing binary search tree, more specifically [a Red-Black tree](https://www.baeldung.com/cs/red-black-trees).

Simply put, being a self-balancing binary search tree, each node of the binary tree comprises of an extra bit, which is used to identify the color of the node which is either red or black. During subsequent insertions and deletions, these “color” bits helps in ensuring that the tree remains more or less balanced.

When compared to a *HashSet*the performance of a *TreeSet* is on the lower side. Operations like *add*, *remove* and *search* take *O(log n)* time while operations like printing *n* elements in sorted order require *O(n)* time.

A *TreeSet* should be our primary choice if we want to keep our entries sorted as a *TreeSet* may be accessed and traversed in either ascending or descending order, and the performance of ascending operations and views is likely to be faster than that of descending ones.

The Principle of Locality – is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed, depending on the memory access pattern.

|  |
| --- |
| **Java Reflection API**  **Java Reflection** is a *process of examining or modifying the run time behavior of a class at run time*.  The **java.lang.Class** class provides many methods that can be used to get metadata, examine and change the run time behavior of a class. |

Mainly used in:

* IDE (Integrated Development Environment) e.g., Eclipse, MyEclipse, NetBeans etc.
* Debugger
* Test Tools etc.

### **Commonly used methods of Class class:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public String getName() | returns the class name |
| 2) public static Class forName(String className)throws ClassNotFoundException | loads the class and returns the reference of Class class. |
| 3) public Object newInstance()throws InstantiationException,IllegalAccessException | creates new instance. |
| 4) public boolean isInterface() | checks if it is interface. |
| 5) public boolean isArray() | checks if it is array. |
| 6) public boolean isPrimitive() | checks if it is primitive. |
| 7) public Class getSuperclass() | returns the superclass class reference. |
| 8) public Field[] getDeclaredFields()throws SecurityException | returns the total number of fields of this class. |
| 9) public Method[] getDeclaredMethods()throws SecurityException | returns the total number of methods of this class. |
| 10) public Constructor[] getDeclaredConstructors()throws SecurityException | returns the total number of constructors of this class. |
| 11) public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException | returns the method class instance. |

### **How to get the object of Class class?**

There are 3 ways to get the instance of Class class. They are as follows:

* forName() method of Class class
* getClass() method of Object class
* the .class syntax

**What is a co-variant return type?**

When we override the method in a subclass, we must keep the return type of the method same or subtype of the original return type. This subtype of return type is termed as a co-variant return

**Threads**

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

Synchronized

The synchronized keyword is all about different threads reading and writing to the same variables, objects and resources.

synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.

*In a very, very small nutshell:* When you have two threads that are reading and writing to the same 'resource', say a variable named foo, you need to ensure that these threads access the variable in an atomic way. Without the synchronized keyword, your thread 1 may not see the change thread 2 made to foo, or worse, it may only be half changed. This would not be what you logically expect.

Interrupt and join

Join() - **When we invoke the join() method on a thread, the calling thread goes into a waiting state. It remains in a waiting state until the referenced thread terminates.**

**Interrupt() -** The **interrupt()** method of thread class is used to interrupt the thread. If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked) then using the interrupt() method, we can interrupt the thread execution by throwing InterruptedException.

Synchronisation

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

### **Why use Synchronization?**

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with the object.

**Volatile**

**volatile vs synchronized:** Before we move on let’s take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility**: It means that changes made by one thread to shared data are visible to other threads.

Java’s synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modify the value of the shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other threads trying to enter the block at the same time will be blocked and put to sleep.

In some cases, we may only desire visibility and not atomicity. The use of synchronized in such a situation is overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of the volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, the use of volatile is limited to a very restricted set of cases as most of the times atomicity is desired. For example, a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is really a compound read-modify-write sequence of operations that must execute atomically.

**Atomic in Java**

In [**multithreading**](https://www.geeksforgeeks.org/multi-threading-models-in-process-management/), the shared entity mostly leads to a problem when [concurrency](https://www.geeksforgeeks.org/java-concurrency-yield-sleep-and-join-methods/) is incorporated. A shared entity such as, **mutable** object or variable, might be changed, which may result in the inconsistency of the program or [database](https://www.geeksforgeeks.org/dbms/). So, it becomes crucial to deal with the shared entity while accessed concurrently. An **atomic variable** can be one of the alternatives in such a scenario.

**Using lock analogy or synchronization:** Synchronization or Locking can solve our problem, but it compromises time efficiency or performance. First, it mandates resource and thread scheduler to control lock. Second, when multiple threads attempt to acquire a lock, only one of them wins, rest are suspended or blocked. Suspending or blocking of threads can have a huge impact on performance.

## Difference between volatile, atomic, and synchronized in Java

Most of the difference between these three synchronization construct comes from the fact mentioned above. Though all three offer the same memory semantics guaranteed by the [happens-before rule](https://javarevisited.blogspot.com/2020/01/what-is-happens-before-in-java-concurrency.html) and visibility guarantee, there is a significant difference between synchronized vs. atomic and volatile variables

1) The first and significant difference between synchronized keyword and volatile and atomic variables comes in terms of [**locking**](https://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html). The synchronized keyword is used to implement a lock-based concurrent algorithm, and that's why to suffer from the limitation of locking. A volatile and atomic variable gives you the power to implement non-blocking algorithms, which is more scalable.  
  
2) On the difference between atomic and volatile variables, they provide the same **visibility guarantee**, but the atomic variable also provides the ability to make compound action, like the read-modify-write atomic. The volatile variable can not be used when one variable's value depends upon others or its own increment value.  
  
3) Atomic variables perform better because they use concurrency support provided by hardware for various atomic operations, like**compare-and-swap** or read-modify-write.

**Deadlocks**

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

What is a Deadlock situation?

A Deadlock is a situation in which two or more threads are waiting on each other to release a resource. Each thread is waiting for a resource that is held by the other waiting thread. At times there is a circular wait when more than two threads are waiting on each other’s resources.

**Detect Dead Lock condition**

We can also detect deadlock by running this program on cmd. We have to collect Thread Dump. Command to collect depends on OS type. If we are using Windows and Java 8, command is jcmd $PID Thread.print

wait, notify and notifyAll

The wait() method causes the current thread to wait indefinitely until another thread either invokes notify() for this object or notifyAll().

Thread safety

Java supports multithreading out of the box. This means that by running bytecode concurrently in separate worker threads, the [JVM](https://www.baeldung.com/jvm-vs-jre-vs-jdk) is capable of improving application performance.

Although multithreading is a powerful feature, it comes at a price. In multithreaded environments, we need to write implementations in a thread-safe way. This means that different threads can access the same resources without exposing erroneous behavior or producing unpredictable results. **This programming methodology is known as “thread-safety”.**

**Thread interference**

Thread interference in java is **a condition which occurs when more than one threads, executing simultaneously, access same piece of data**. When more than one threads have access to same data, it is possible that data may get corrupted or one may not get the desired output

**How To Avoid Thread Interference or How To Acheive Thread Safeness?**

Following are some methods which are used to avoid thread interference in java.(These methods will be discussed in detail in subsequent articles).

* By declaring the method as synchronized.
* By declaring the variables as final.
* By declaring the variable as volatile.
* By creating the immutable objects.
* By using Atomic operations.
* By restricting the access to same object by multiple threads.

[*Future Object*](https://www.geeksforgeeks.org/future-and-futuretask-in-java/)

*The result of the task submitted for execution to an executor can be accessed using the java.util.concurrent.The future object returned by the executor. Future can be thought of as a promise made to the caller by the executor. The future interface is mainly used to get the results of Callable results. whenever the task execution is completed, it is set in this Future object by the executor.*

start() and run()

**New Thread creation:**When a program calls the start() method, a new thread is created and then the run() method is executed. But if we directly call the run() method then no new thread will be created and run() method will be executed as a normal method call on the current calling thread itself and no multi-threading will take place.

# Reflection in Java

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

* The required classes for reflection are provided under java.lang.reflect package.
* Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
* Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

Reflection can be used to get information about –

1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
3. **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

Thread.yield()

yield() provides a mechanism to inform the “scheduler” that **the current thread is willing to relinquish its current use of processor but it'd like to be scheduled back soon as possible.**

Reentrant lock

* The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.
* As the name says, ReentrantLock allows threads to enter into the lock on a resource more than once. When the thread first enters into the lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlocks request, hold count is decremented by one and when hold count is 0, the resource is unlocked.
* Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

**Semaphore**

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

We can use semaphores to limit the number of concurrent threads accessing a specific resource.

## What Is a Binary Semaphore?

A binary [semaphore](https://www.baeldung.com/java-semaphore) provides a signaling mechanism over the access of a single resource. In other words, a binary semaphore**provides a mutual exclusion that allows only one thread to access a critical section at a time**.

For that, it keeps only one permit available for access. Hence, **a binary semaphore has only two states: one permit available or zero permits available**.

**Countdown Latch**

CountDownLatch is used to make sure that a task waits for other threads before it starts.

**Working of CountDownLatch:**  
When we create an object of CountDownLatch, we specify the number of threads it should wait for, all such thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.

**CyclicBarrier**

CyclicBarrier is used to make threads wait for each other. It is used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread. In other words, a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output. After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.

# **Java CyclicBarrier vs CountDownLatch**

A CountDownLatch is a construct that a thread waits on while other threads count down on the latch until it reaches zero.

A CyclicBarrier is a reusable construct where a group of threads waits together until all of the threads arrive. At that point, the barrier is broken and an action can optionally be taken.

* 1. In short, **CyclicBarrier maintains a count of threads** whereas **CountDownLatch maintains a count of tasks**.
  2. **when the barrier trips in CyclicBarrier, the count resets to its original value.** **CountDownLatch is different because the count never resets.**

**CountDownLatch** countDownLatch = **new** **CountDownLatch**(2);

**Thread** t = **new** **Thread**(() -> {

countDownLatch.countDown();

countDownLatch.countDown();

});

t.start();

countDownLatch.await();

assertEquals(0, countDownLatch.getCount());

**CyclicBarrier** cyclicBarrier = **new** **CyclicBarrier**(2);

**Thread** t = **new** **Thread**(() -> {

**try** { cyclicBarrier.await(); cyclicBarrier.await();

} **catch** (InterruptedException | BrokenBarrierException e) {

// error handling } });

t.start();

assertEquals(1, cyclicBarrier.getNumberWaiting()); assertFalse(cyclicBarrier.isBroken());

*CyclicBarrier* allows a number of threads to wait on each other, whereas *CountDownLatch* allows one or more threads to wait for a number of tasks to complete.

In short, ***CyclicBarrier* maintains a count of *threads*** whereas ***CountDownLatch* maintains a count of *tasks***.

## SynchronousQueue vs TransferQueue

SynchronousQueue uses dual queues — for waiting producers and consumers — and protect both queues with single lock. LinkedTransferQueue uses CAS operations to perform non-blocking algorithms and allows to avoid serialization bottlenecks.

yield()

The **yield()**basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.

### sleep()

This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

**Wait() and Sleep()**

***wait()* is an instance method that's used for thread synchronization.**

It can be called on any object, as it's defined right on*java.lang.Object,*but it can **only be called from a synchronized block**. It releases the lock on the object so that another thread can jump in and acquire a lock.

On the other hand, *Thread.sleep()* is a static method that can be called from any context.***Thread.sleep()* pauses the current thread and does not release any locks.**

**Executor service**

*Executor* and *ExecutorService* are two related interfaces of *java.util.concurrent* framework. *Executor* is a very simple interface with a single *execute* method accepting *Runnable* instances for execution. In most cases, this is the interface that your task-executing code should depend on.

*ExecutorService* extends the *Executor* interface with multiple methods for handling and checking the lifecycle of a concurrent task execution service (termination of tasks in case of shutdown) and methods for more complex asynchronous task handling including *Futures*.

Executor Framework

Java provides its own multi-threading framework called the Java Executor Framework.

Java executor framework ([java.util.concurrent.Executor](https://www.geeksforgeeks.org/java-util-concurrent-executor-interface-with-examples/)), released with the JDK 5 is used to run the Runnable objects without creating new threads every time and mostly re-using the already created threads. We all know that there are two ways to create a thread in java.

//Executes only one thread

ExecutorService es = Executors.newSingleThreadExecutor();

//Internally manages thread pool of 2 threads

ExecutorService es = Executors.newFixedThreadPool(2);

//Internally manages thread pool of 10 threads to run scheduled tasks

ExecutorService es = Executors.newScheduledThreadPool(10);

CustomThreadPoolExecutor executor =

**new** CustomThreadPoolExecutor(10, 20, 5, TimeUnit.SECONDS,

blockingQueue, **new** ThreadPoolExecutor.AbortPolicy());

1. **Fixed thread pool executor** – Creates a thread pool that reuses a fixed number of threads to execute any number of tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. It is the best fit for most off the real-life use-cases.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10); |

1. **Cached thread pool executor** – Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. DO NOT use this thread pool if tasks are long-running. It can bring down the system if the number of threads goes beyond what the system can handle.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool(); |

1. **Scheduled thread pool executor** – Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(10); |

1. **Single thread pool executor** – Creates single thread to execute all tasks. Use it when you have only one task to execute.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newSingleThreadExecutor(); |

1. **Work stealing thread pool executor** – Creates a thread pool that maintains enough threads to support the given parallelism level. Here parallelism level means the maximum number of threads which will be used to execute a given task, at a single point of time, in multi-processor machines.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newWorkStealingPool(4);  Runnable runnableTask = () -> {          try {              TimeUnit.MILLISECONDS.sleep(1000);              System.out.println("Current time :: " + LocalDateTime.now());          } catch (InterruptedException e) {              e.printStackTrace();          }      }; Execute Runnable tasks We can execute runnables using the following methods :  **void execute(Runnable task)** – executes the given command at some time  in the future.  **Future submit(Runnable task)** – submits a runnable task for execution and  returns a Future representing that task. The Future’s get() method will  return null upon successful completion.  **Future submit(Runnable task, T result)** – Submits a runnable task for  execution and returns a Future representing that task. The Future’s  get() method will return the given result upon successful completion. Execute Callable tasks We can execute callable tasks using the following methods :   * **Future submit(callableTask)** – submits a value-returning task for   execution and returns a Future representing the pending results of the task.   * **List<Future> invokeAll(Collection tasks)** – executes the given tasks,   returning a list of Futures holding their status and results **when all**  **complete**. Notice that result is available only when all tasks are completed. Note that a completed task could have terminated either normally or by  throwing an exception.   * **List<Future> invokeAll(Collection tasks, timeOut, timeUnit)** –   executes the given tasks, returning a list of Futures holding their status and results **when all complete or the timeout expires**.  There are three methods to invoke shutdown:   * **void shutdown()** – Initiates an orderly shutdown in which previously   submitted tasks are executed, but no new tasks will be accepted.   * **List<Runnable> shutdownNow()** – Attempts to stop all actively   executing tasks, halts the processing of waiting tasks, and returns a list  of the tasks that were awaiting execution.   * **void awaitTermination()** – It blocks until all tasks have completed   execution after a shutdown request, or the timeout occurs, or the current  thread is interrupted, whichever happens first. |

Thread pool

**A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.** Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

* Java provides the Executor framework which is centered around the Executor interface, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
* They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
* To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.

**Thread Local**

The *TheadLocal*construct allows us to store data that will be **accessible only**by**a specific thread**.

Let's say that we want to have an *Integer* value that will be bundled with the specific thread:

ThreadLocal<Integer> threadLocalValue = **new** **ThreadLocal**<>();

Next, when we want to use this value from a thread, we only need to call a *get()* or *set()* method. Simply put, we can imagine that *ThreadLocal* stores data inside of a map with the thread as the key.

As a result, when we call a *get()* method on the *threadLocalValue*, we'll get an *Integer* value for the requesting thread:

**Completable Future**

The *Future* interface was added in Java 5 to serve as a result of an asynchronous computation, but it did not have any methods to combine these computations or handle possible errors.

**Java 8 introduced the *CompletableFuture* class.** Along with the *Future* interface, it also implemented the *CompletionStage* interface. This interface defines the contract for an asynchronous computation step that we can combine with other steps.

We don't need to use the ExecutorService explicitly. The **CompletableFuture internally uses**[**ForkJoinPool**](https://www.baeldung.com/java-fork-join)**to handle the task asynchronously**

**CompletableFuture<T>**class implements **Future<T>**interface in Java. **CompletableFuture**can be used as a Future that has explicitly completed. The **Future**interface doesn’t provide a lot of features, we need to get the result of asynchronous computation using the **get()** method, which is blocked, so there is no scope to run multiple dependent tasks in a **non-blocking** fashion whereas **CompleteFuture**class can provide the functionality to chain multiple dependent tasks that run **asynchronously**, so we can create a chain of tasks where the next task is triggered when the result of the current task is available.

**ArrayBlockingQueue** class is a bounded blocking queue backed by an array. By bounded, it means that the size of the Queue is fixed. Once created, the capacity cannot be changed. Attempts to put an element into a full queue will result in the operation blocking. Similarly attempts to take an element from an empty queue will also be blocked. Boundness of the ArrayBlockingQueue can be achieved initially bypassing capacity as the parameter in the constructor of ArrayBlockingQueue. This queue orders elements **FIFO (first-in-first-out)**. It means that the head of this queue is the oldest element of the elements present in this queue.

The tail of this queue is the newest element of the elements of this queue. The newly inserted elements are always inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

### ReentrantReadWriteLock

### **Read Lock** – if no thread acquired the write lock or requested for it then multiple threads can acquire the read lock

**Write Lock** – if no threads are reading or writing then only one thread can acquire the write lock

### StampedLock

StampedLock is introduced in Java 8. It also supports both read and write locks. However, lock acquisition methods return a stamp that is used to release a lock or to check if the lock is still valid:

## ****Working With****Conditions

The Condition class provides the ability for a thread to wait for some condition to occur while executing the critical section.

This can occur when a thread acquires the access to the critical section but doesn't have the necessary condition to perform its operation. For example, a reader thread can get access to the lock of a shared queue, which still doesn't have any data to consume.

**Thread starvation**

 Starvation of thread in java is said to occur when a particular thread does not get access to the object or the resource which leads to an increase in waiting and execution time.

Starvation is said to occur when two or more threads are allocated to the C.P.U. (Central Processing Unit) and takes a lot of time in execution, due to which other waiting threads cannot get the C.P.U. for its execution to carry on.

**Causes Of Starvation:**

There are many reasons for causes of starvation of threads in java, some of them are described below:

* **High Priority Running Thread:** There may be a case where a high priority thread is running by occupying the C.P.U. and it needs heavy processing which requires a lot of time in completion, so for this work to get completely executed the other threads which have a low priority order have to wait for a long time which leads to starvation.
* **Synchronized Block In Java:** There may be a case where the order in which the threads are allowed to enter the [synchronized block](https://www.java67.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) is granted the resources in the same order as they are programmed to be scheduled, which results in waiting for the resources and the objects by another thread leading to starvation, where the other threads other than a particular thread are given the C.P.U. for its execution.
* **Threads Waiting On An Object Remains Waiting Forever:** The [notify() method in java](https://javarevisited.blogspot.com/2015/07/how-to-use-wait-notify-and-notifyall-in.html#axzz6e8hmwujv) has no track on the threads that which particular thread is wakening if there are multiple threads, therefore there may be a risk that any of the thread is being processed and the other threads are never called for execution.

Some of the important points to remove starvation of threads are given as follows:

* By implementation of the Thread.yield() method, so that when the thread in the process after releasing the lock gets a fair chance to occupy the C.P.U. and can get some time to complete its execution till the original thread again gets the control over the C.P.U.
* One can also use the [Thread.sleep()](http://www.java67.com/2015/06/how-to-pause-thread-in-java-using-sleep.html) method to given chance to other Threads for execution.

## ThreadLocalRandom****Over****Random

**ThreadLocalRandom is a combination of the**[***ThreadLocal***](https://www.baeldung.com/java-threadlocal)**and Random classes and is isolated to the current thread.** Thus, it achieves better performance in a multithreaded environment by simply avoiding any concurrent access to instances of Random.

The random number obtained by one thread is not affected by the other thread, whereas java.util.Random provides random numbers globally.

Also, unlike Random, ThreadLocalRandom doesn't support setting the seed explicitly. Instead, it overrides the setSeed(long seed) method inherited from Random to always throw an UnsupportedOperationException if called.

ThreadLocalRandom completely removes this contention, as each thread has its own instance of Random and, consequently, its own confined seed.

that sharing one global Random instance leads to sub-optimal performance in high contention. However, using one dedicated instance per thread is also overkill.

**Instead of a dedicated instance of Random per thread, each thread only needs to maintain its own seed value**. As of Java 8, the [*Thread*](https://github.com/openjdk/jdk14u/blob/d48548f5b7713e0d51b107a5e2dfd60383edbd88/src/java.base/share/classes/java/lang/Thread.java#L2059) class itself has been retrofitted to maintain the seed value:

fair lock

**Livelock**

Livelock is another concurrency problem and is similar to deadlock. In livelock, **two or more threads keep on transferring states between one another** instead of waiting infinitely as we saw in the deadlock example. Consequently, the threads are not able to perform their respective tasks.

**Livelock** occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently.

fail-fast & fail-safe

Difference Between Fail Fast and Fail Safe Iterators

The Major difference between Fail Fast and Fail Safe iterator is that the **Fail Safe does not throw any ConcurrentModificationException in modifying the object during** the iteration process, contrary to fail fast, which throws an exception in such scenarios.

Object level locking and class level locking

**In synchronization, there are two types of locks on threads:**

**Object-level lock:**Every object in java has a unique lock. Whenever we are using a synchronized keyword, then only the lock concept will come into the picture. If a thread wants to execute then synchronized method on the given object. First, it has to get a lock-in that object. Once the thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases the lock. Acquiring and release lock internally is taken care of by JVM and the programmer is not responsible for these activities.

**Class level lock:**Every class in Java has a unique lock which is nothing but a class level lock. If a thread wants to execute a static synchronized method, then the thread requires a class level lock. Once a thread got the class level lock, then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread releases the lock.

In multithreading environment, two or more threads can access the shared resources simultaneously which can lead the inconsistent behaviour of the system. Java uses concept of locks to restrict concurrent access of shared resources or objects. Locks can be applied at two levels −

* Object Level Locks − It can be used when you want non-static method or non-static block of the code should be accessed by only one thread.
* Class Level locks − It can be used when we want to prevent multiple threads to enter the synchronized block in any of all available instances on runtime. It should always be used to make static data thread safe.

| **Sr. No.** | **Key** | **Object Level Lock** | **Class Level Lock** |
| --- | --- | --- | --- |
| 1 | Basic | It can be used when you want non-static method or non-static block of the code should be accessed by only one thread | It can be used when we want to prevent multiple threads to enter the synchronized block in any of all available instances on runtime |
| 2 | Static/Non Static | It should always be used to make non-static data thread safe. | It should always be used to make static data thread safe. |
| 3 | Number of Locks | Every object the class may have their own lock | Multiple objects of class may exist but there is always one class’s class object lock available |

Notes:

A Semaphore is an object used to limit the number of threads simultaneously accessing a resource.

volatile variables are written to and read from the main memory. **The CPU does not cache the value of a volatile variable.**

Volatile variables have the visibility features of synchronized but not the atomicity features

*AtomicInteger*should be used only when atomic operations are needed. Also, the race condition can still exist between two separate atomic operations.

The effect of the volatile keyword is approximately that each individual read or write operation on that variable is made atomically visible to all threads.

The effect of the volatile keyword is approximately that each individual read or write operation on that variable is made atomically visible to all threads.

Notably, however, an operation that requires more than one read/write -- such as i++, which is equivalent to i = i + 1, which does one read and one write -- is *not* atomic, since another thread may write to i between the read and the write.

The Atomic classes, like AtomicInteger and AtomicReference, provide a wider variety of operations atomically, specifically including increment for AtomicInteger.

**A CountDownLatch is useful when we need a mechanism to notify one or more threads that a set of operations performed by other threads has finished.**

**Method overloading with different return type**

When a class has two or more methods by the same name but different parameters, at the time of calling based on the parameters passed respective method is called (or respective method body will be bonded with the calling line dynamically). This mechanism is known as **method overloading**.

**Overloading with same arguments and different return type −**

No, you cannot overload a method based on different return type but same argument type and number in java.

In overloading it is must that the both methods have −

* same name.
* different parameters (different type or, different number or both).

The return type doesn’t matter. If they don’t have different parameters, they both are still considered as same method and a compile time error will be generated.

Covariant

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type.

# **Autowiring in Spring**

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

## Autowiring Modes

There are many autowiring modes:

|  |  |  |
| --- | --- | --- |
| **No.** | **Mode** | **Description** |
| 1) | no | It is the default autowiring mode. It means no autowiring bydefault. |
| 2) | byName | The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method. |
| 3) | byType | The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method. |
| 4) | constructor | The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters. |
| 5) | autodetect | It is deprecated since Spring 3. |

In Spring, you can use @Autowired annotation to auto-wire bean on the setter method, constructor, or a field. Moreover, it can autowire the property in a particular bean.

### @Autowired on Properties

In the below example, when the annotation is directly used on properties, Spring looks for and injects Department when Employee is created. This is how it eliminates the need for getters and setters.

### @Autowired on Setters

In the below example, when the annotation is used on the setter method, the setter method is called with the instance of Department when Employee is created.

### @Autowired on Constructors

In the below example, the annotation is used on a constructor, an instance of Department is injected as an argument to the constructor when Employee is created.

# 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.*

**Example**

* Java

|  |
| --- |
| // Java Program to Create An Immutable Class    // Importing required classes  import java.util.HashMap;  import java.util.Map;    // Class 1  // An immutable class  final class Student {        // Member attributes of final class      private final String name;      private final int regNo;      private final Map<String, String> metadata;        // Constructor of immutable class      // Parameterized constructor      public Student(String name, int regNo,                     Map<String, String> metadata)      {            // This keyword refers to current instance itself          this.name = name;          this.regNo = regNo;            // Creating Map object with reference to HashMap          // Declaring object of string type          Map<String, String> tempMap = new HashMap<>();            // Iterating using for-each loop          for (Map.Entry<String, String> entry :               metadata.entrySet()) {              tempMap.put(entry.getKey(), entry.getValue());          }            this.metadata = tempMap;      }        // Method 1      public String getName() { return name; }        // Method 2      public int getRegNo() { return regNo; }        // Note that there should not be any setters        // Method 3      // User -defined type      // To get meta data      public Map<String, String> getMetadata()      {            // Creating Map with HashMap reference          Map<String, String> tempMap = new HashMap<>();            for (Map.Entry<String, String> entry :               this.metadata.entrySet()) {              tempMap.put(entry.getKey(), entry.getValue());          }          return tempMap;      }  }    // Class 2  // Main class  class GFG {        // Main driver method      public static void main(String[] args)      {            // Creating Map object with reference to HashMap          Map<String, String> map = new HashMap<>();            // Adding elements to Map object          // using put() method          map.put("1", "first");          map.put("2", "second");            Student s = new Student("ABC", 101, map);            // Calling the above methods 1,2,3 of class1          // inside main() method in class2 and          // executing the print statement over them          System.out.println(s.getName());          System.out.println(s.getRegNo());          System.out.println(s.getMetadata());            // Uncommenting below line causes error          // s.regNo = 102;            map.put("3", "third");          // Remains unchanged due to deep copy in constructor          System.out.println(s.getMetadata());          s.getMetadata().put("4", "fourth");          // Remains unchanged due to deep copy in getter          System.out.println(s.getMetadata());      }  } |

**Output**

ABC

101

{1=first, 2=second}

{1=first, 2=second}

{1=first, 2=second}

In this example, we have created a final class named Student. It has three final data members, a parameterized constructor, and getter methods. Please note that there is no setter method here. Also, note that we don’t need to perform deep copy or cloning of data members of wrapper types as they are already immutable.

 Why Immutable classes:

* Simplicity - each class is in one state only
* Thread Safe - because the state cannot be changed, no synchronization is required
* Writing in an immutable style can lead to more robust code. Imagine if Strings weren't immutable; Any getter methods that returned a String would require the implementation to create a defensive copy before the String was returned - otherwise a client may accidentally or maliciously break that state of the object.

In general it is good practise to make an object immutable unless there are severe performance problems as a result. In such circumstances, mutable builder objects can be used to build immutable objects e.g. StringBuilder

# **Quick Guide to Spring Bean Scopes**

The scope of a bean defines the life cycle and visibility of that bean in the contexts we use it.

The latest version of the Spring framework defines 6 types of scopes:

* singleton
* prototype
* request
* session
* application
* websocket

## ****Singleton Scope****

When we define a bean with the singleton scope, the container creates a single instance of that bean; all requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean. This scope is the default value if no other scope is specified.

## ****Prototype Scope****

A bean with the prototype scope will return a different instance every time it is requested from the container. It is defined by setting the value prototype to the*@Scope*annotation in the bean definition

The *request* scope creates a bean instance for a single HTTP request, while the s*ession* scope creates a bean instance for an HTTP Session.

The *application*scope creates the bean instance for the lifecycle of a *ServletContext*, and the *websocket*scope creates it for a particular *WebSocket*session.

Bean

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

## ****Inversion of Control****

Simply put, [Inversion of Control](https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring) (IoC) is **a process in which an object defines its dependencies without creating them.** This object delegates the job of constructing such dependencies to an IoC container.

Let's start with the declaration of a couple of domain classes before diving into IoC.

**When a Spring IoC container constructs objects of those types, all the objects are called Spring beans, as they are managed by the IoC container.**

**Annoations**

@ComponentScan configures which **packages to scan for classes with annotation configuration**. We can specify the base package names directly with one of the basePackages or value arguments (value is an alias for basePackages):

@Component is a class level annotation. During the component scan, **Spring Framework automatically detects classes annotated with @Component:**

DAO or Repository classes usually represent the database access layer in an application, and should be annotated with @Repository:

One advantage of using this annotation is that **it has automatic persistence exception translation enabled**. When using a persistence framework, such as Hibernate, native exceptions thrown within classes annotated with *@Repository* will be automatically translated into subclasses of Spring's *DataAccessExeption*.

**To enable exception translation**, we need to declare our own *PersistenceExceptionTranslationPostProcessor* bean:

@Bean

**public** PersistenceExceptionTranslationPostProcessor **exceptionTranslation**() {

**return** **new** **PersistenceExceptionTranslationPostProcessor**();

}

The **business logic** of an application usually resides within the service layer, so we’ll use the @Service annotation to indicate that a class belongs to that layer:

*@Controller* is a class level annotation, which tells the Spring Framework that this class serves as a **controller in Spring MVC**:

Configuration classes can **contain bean definition methods** annotated with @Bean:

MVC framework

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram −



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet* −

* After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
* The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
* The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
* Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* which is an extension of the plain*ApplicationContext* with some extra features necessary for web applications.

# Internal Working of HashMap in Java

hashmap’s get and put method works internally. What operations are performed. How the hashing is done. How the value is fetched by key. How the key-value pair is stored.  
As in [previous article](https://www.geeksforgeeks.org/java-util-hashmap-in-java), HashMap contains an array of Node and Node can represent a class having following objects : 

1. int hash
2. K key
3. V value
4. Node next

Now we will see how this works. First we will see the hashing process. 

**Hashing**

Hashing is a process of converting an object into integer form by using the method hashCode(). Its necessary to write hashCode() method properly for better performance of HashMap. Here I am taking key of my own class so that I can override hashCode() method to show different scenarios. My Key class is 

//custom Key class to override hashCode()

// and equals() method

class Key

{

String key;

Key(String key)

{

this.key = key;

}

@Override

public int hashCode()

{

return (int)key.charAt(0);

}

@Override

public boolean equals(Object obj)

{

return key.equals((String)obj);

}

}

Here overrided hashCode() method returns the first character’s ASCII value as hash code. So whenever the first character of key is same, the hash code will be same. You should not approach this criteria in your program. It is just for demo purpose. As HashMap also allows null key, so hash code of null will always be 0.

[**hashCode() method**](https://www.geeksforgeeks.org/equals-hashcode-methods-java/)

hashCode() method is used to get the hash Code of an object. hashCode() method of object class returns the memory reference of object in integer form. Definition of hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because there is not any direct method in java to fetch the reference of object. It is possible to provide your own implementation of hashCode().   
In HashMap, hashCode() is used to calculate the bucket and therefore calculate the index. 

[**equals() method**](https://www.geeksforgeeks.org/equals-hashcode-methods-java/)

equals method is used to check that 2 objects are equal or not. This method is provided by Object class. You can override this in your class to provide your own implementation.   
HashMap uses equals() to compare the key whether they are equal or not. If equals() method return true, they are equal otherwise not equal. 

**Buckets**

A bucket is one element of HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes. Buckets are different in capacity. A relation between bucket and capacity is as follows: 

capacity = number of buckets \* load factor

A single bucket can have more than one nodes, it depends on hashCode() method. The better your hashCode() method is, the better your buckets will be utilized. 

**Index Calculation in Hashmap**

Hash code of key may be large enough to create an array. hash code generated may be in the range of integer and if we create arrays for such a range, then it will easily cause outOfMemoryException. So we generate index to minimize the size of array. Basically following operation is performed to calculate index. 

index = hashCode(key) & (n-1).

where n is number of buckets or the size of array. In our example, I will consider n as default size that is 16.

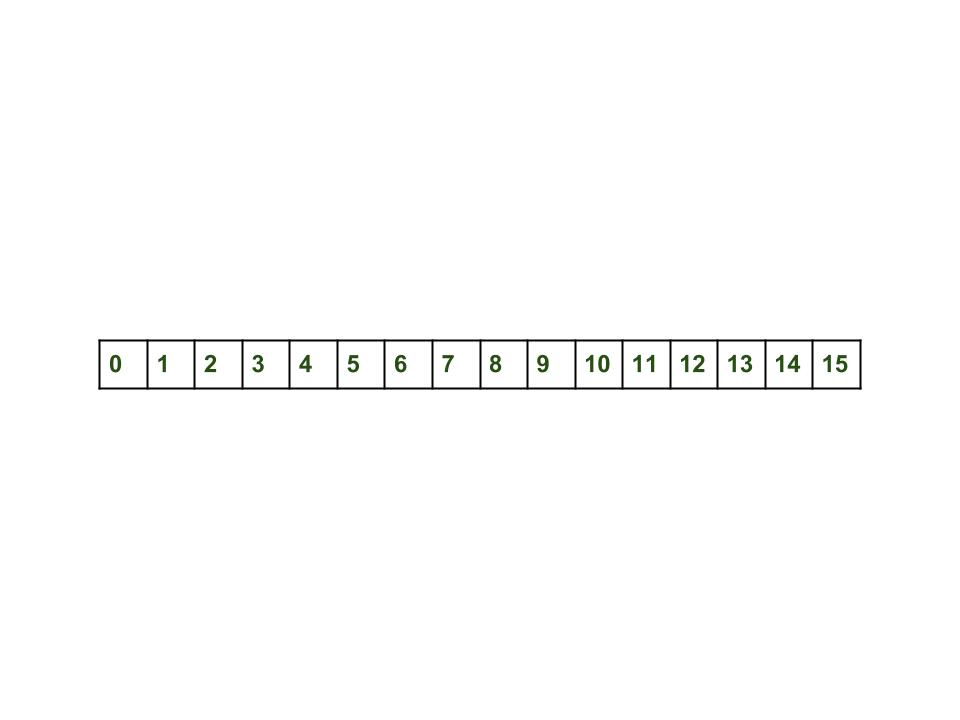
**Why the above method is used to calculate the index**

Using a bitwise AND operator is similar to doing bit masking wherein only the lower bits of the hash integer are considered which in turn provides a very efficient method of calculating the modulus based on the length of the hashmap.  
 q

* **Initially Empty hashMap:** Here, the hashmap is size is taken as 16.

HashMap map = new HashMap();

* HashMap :

[](https://media.geeksforgeeks.org/wp-content/uploads/Hashmap_working.jpg)

* **Inserting Key-Value Pair:** Putting one key-value pair in above HashMap

map.put(new Key("vishal"), 20);

* **Steps:**
  1. Calculate hash code of Key {“vishal”}. It will be generated as 118.
  2. Calculate index by using index method it will be 6.
  3. Create a node object as :

{

int hash = 118

// {"vishal"} is not a string but

// an object of class Key

Key key = {"vishal"}

Integer value = 20

Node next = null

}

2. Place this object at index 6, if no other object is presented there.

* **Inserting another Key-Value Pair:** Now, putting other pair that is,

map.put(new Key("sachin"), 30);

* **Steps:**
  1. Calculate hashCode of Key {“sachin”}. It will be generated as 115.
  2. Calculate index by using index method it will be 3.
  3. Create a node object as :

{

int hash = 115

Key key = {"sachin"}

Integer value = 30

Node next = null

}



* **In Case of collision:** Now, putting another pair that is,

map.put(new Key("vaibhav"), 40);

* **Steps:**
  1. Calculate hash code of Key {“vaibhav”}. It will be generated as 118.
  2. Calculate index by using index method it will be 6.
  3. Create a node object as :

{

int hash = 118

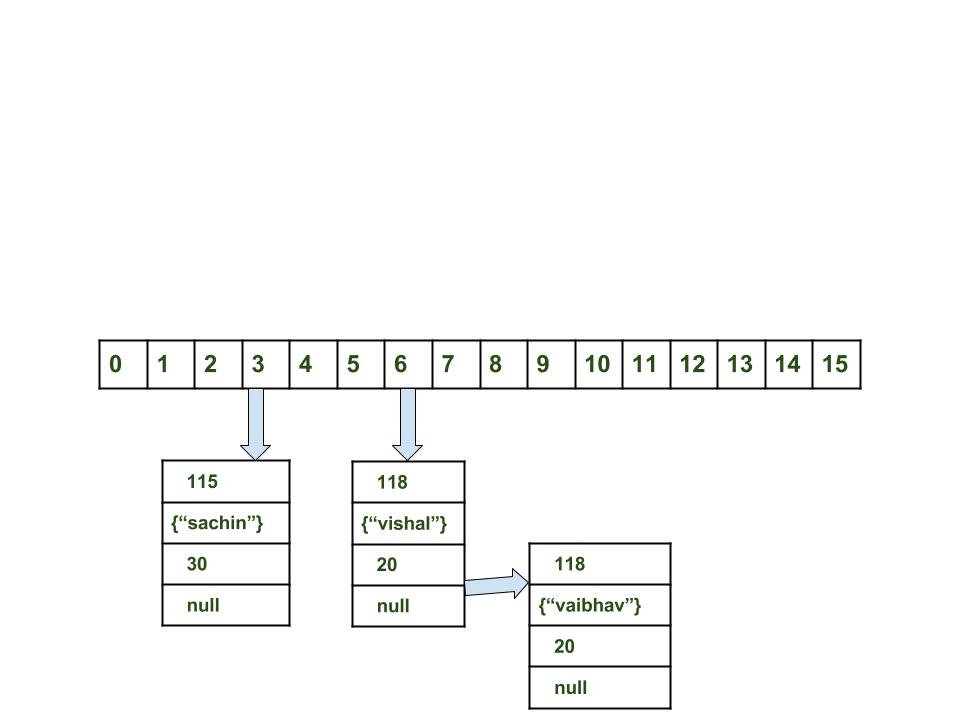
Key key = {"vaibhav"}

Integer value = 40

Node next = null

}

1. Place this object at index 6 if no other object is presented there.
2. In this case a node object is **found at the index 6** – this is a case of collision.
3. In that case, check via hashCode() and equals() method that if both the keys are same.
4. If keys are same, replace the value with current value.
5. Otherwise connect this node object to the previous node object via linked list and both are stored at index 6.   
   Now HashMap becomes :

[](https://media.geeksforgeeks.org/wp-content/uploads/Hashmap_working_3.jpg)

**Using get method()**

Now lets try some get method to get a value. get(K key) method is used to get a value by its key. If you don’t know the key then it is not possible to fetch a value. 

* **Fetch the data for key sachin:**

map.get(new Key("sachin"));

* **Steps:**
  1. Calculate hash code of Key {“sachin”}. It will be generated as 115.
  2. Calculate index by using index method it will be 3.
  3. Go to index 3 of array and compare first element’s key with given key. If both are equals then return the value, otherwise check for next element if it exists.
  4. In our case it is found as first element and returned value is 30.
* Fetch the data for key vaibahv:

map.get(new Key("vaibhav"));

* **Steps:**
  1. Calculate hash code of Key {“vaibhav”}. It will be generated as 118.
  2. Calculate index by using index method it will be 6.
  3. Go to index 6 of array and compare first element’s key with given key. If both are equals then return the value, otherwise check for next element if it exists.
  4. In our case it is not found as first element and next of node object is not null.
  5. If next of node is null then return null.
  6. If next of node is not null traverse to the second element and repeat the process 3 until key is not found or next is not null.
  7. Time complexity is almost constant for put and get method until rehashing is not done.
  8. In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
  9. If key given already exist in HashMap, the value is replaced with new value.
  10. hash code of null key is 0.
  11. When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

**Set implementation**

whenever we create a HashSet, it internally creates a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as it’s key and constant Object called **“PRESENT”** as it’s value. So we can say that **a Set achieves uniqueness internally through HashMap**. Now the whole story comes around [how a HashMap and put() method internally works](https://www.geeksforgeeks.org/internal-working-of-hashmap-java/).

As we know in a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) each key is unique and when we call put(Key, Value) method, it returns the previous value associated with key, or null if there was no mapping for key. So in add() method we check the return value of map.put(key, value) method with null value.

1. If map.put(key, value) returns null, then the statement “map.put(e, PRESENT) == null” will return true and element is added to the HashSet(internally HashMap).
2. If map.put(key, value) returns old value of the key, then the statement “map.put(e, PRESENT) == null” will return false and element is not added to the HashSet(internally HashMap).

As LinkedHashSet extends HashSet, so it internally calls constructors of HashSet using [super()](https://www.geeksforgeeks.org/super-keyword/).

ActiveMQ so many messages

Sonarqube major and minor issues

Can Functional interface extends an interface having 2 methods

When to use list and map

Soap and rest

Spring and springboot

Exception handling

Spring data Jpa transactional

Zuul

Eurekha

API gateway

@Query

How to call methods of default methods

# [**How can I change server in spring boot application?**](https://stackoverflow.com/questions/63054114/how-can-i-change-server-in-spring-boot-application)

You will need to update pom.xml, add the dependency for spring-boot-starter-jetty. Also, you will need to exclude default added spring-boot-starter-tomcat dependency.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-undertow</artifactId>

</dependency>

@Component and @Bean do two quite different things, and shouldn't be confused.

@Component (and @Service and @Repository) are used to auto-detect and auto-configure beans using classpath scanning. There's an implicit one-to-one mapping between the annotated class and the bean (i.e. one bean per class). Control of wiring is quite limited with this approach, since it's purely declarative.

@Bean is used to *explicitly* declare a single bean, rather than letting Spring do it automatically as above. It decouples the declaration of the bean from the class definition, and lets you create and configure beans exactly how you choose.

# [**What is the difference between Hibernate and Spring Data JPA**](https://stackoverflow.com/questions/23862994/what-is-the-difference-between-hibernate-and-spring-data-jpa)

Hibernate is a JPA implementation, while Spring Data JPA is a JPA data access abstraction. Spring Data JPA cannot work without a JPA provider.

Spring Data offers a solution to the [DDD Repository pattern](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design) or the legacy GenericDao custom implementations. It can also generate JPA queries on your behalf through method name conventions.

With Spring Data, you may use Hibernate, EclipseLink, or any other JPA provider. A very interesting benefit of using Spring or Java EE is that you can control transaction boundaries declaratively using the @Transactional annotation.

Spring JDBC is much more lightweight, and it's intended for native querying, and if you only intend to use JDBC alone, then you are better off using Spring JDBC to deal with the JDBC verbosity.

Therefore, Hibernate and Spring Data are complementary rather than competitors.

DDL & DML

DDL stands for Data definition language. This language is used to change the structure of database

CREATE, ALTER,DROP, TRUNCATE

DML

Data Manipulation language. DML is used to manage the data in the database.

INSERT, DELETE, UPDATE.

ORDER BY

This command is used to sort the data returned in ascending order

@Transactional

@Transactional annotation is used when you want the certain method/class(=all methods inside) to be executed in a transaction.

It is used to **combine more than one writes** on a database **as a single atomic operation**.

When somebody call the method annotated with @Transactional **all or none of the writes** on the database **is executed**.

We can use [*@Transactional*](https://www.baeldung.com/transaction-configuration-with-jpa-and-spring) to wrap a method in a database transaction.

It allows us to set propagation, isolation, timeout, read-only, and rollback conditions for our transaction. We can also specify the transaction manager.

# ACID Properties in DBMS

A [**transaction**](https://www.geeksforgeeks.org/sql-transactions/) is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.   
In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.

**Atomicity**   
By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.   
—**Abort**: If a transaction aborts, changes made to database are not visible.   
—**Commit**: If a transaction commits, changes made are visible.   
Atomicity is also known as the ‘All or nothing rule’.

**Consistency**   
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,   
The total amount before and after the transaction must be maintained.   
Total **before T** occurs = **500 + 200 = 700**.   
Total **after T occurs** = **400 + 300 = 700**.   
Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

**Isolation**   
This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

**Durability:**   
This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

EntrySet

The java.util.HashMap.entrySet() method in Java is used to create a set out of the same elements contained in the hash map. It basically returns a set view of the hash map or we can create a new set and store the map elements into them.

Exception

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program’s instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception such as the name and description of the exception and the state of the program when the exception occurred.

How to call methods of abstract class

Since you cannot instantiate an abstract class you cannot access its instance methods too. You can call only static methods of an abstract class (since an instance is not required).

The only way to access the non-static method of an abstract class is to extend it, implement the abstract methods in it (if any) and then using the subclass object you need to invoke the required methods.

When protected modifier used

The protected modifier **specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package**.

You use protected when

* **Your class is designed for inheritance** - You expect the users of your library to inherit from the class that you are designing. Very often the class will be abstract.
* **The class provides special functionality to its derived classes that must not be visible to other classes** - You know that derived classes must have access to information that would otherwise be private, **or**
* **The derived classes must provide functionality to the base class** - See [*Template Method Pattern*](http://en.wikipedia.org/wiki/Template_method_pattern) for information about this use of protected methods.

What is synchronization

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Remove Duplicate elements in list

List<Integer> numbersList = new ArrayList<>(Arrays.asList(1,1,2,3,4,5,6,7,5,9);

numbersList.stream().distinct().collect(Collectors.toList());

Another method:

List<Integer> numbersList = new ArrayList<>(Arrays.asList(1,1,2,3,4,5,6,7,5,9);

Map<Integer, Integer> map = new HashMap<>();

numbersList.forEach(e -> map.put(e, map.getOrDefault(e, 0) + 1));

List<Integer> values = map.entrySet().stream()

.filter(e -> e.getValue() > 1)

.map(Map.Entry::getKey)

.collect(Collectors.toList());

**Mockito**

@Service

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

@Autowired **private** RestTemplate restTemplate;

**public** Employee **getEmployee**(String id) {

**ResponseEntity** resp = restTemplate.getForEntity("http://localhost:8080/employee/" + id, Employee.class);

**return** resp.getStatusCode() == HttpStatus.OK ? resp.getBody() : null;

}

}

@RunWith(MockitoJUnitRunner.class) **public** **class** **EmployeeServiceTest** {

@Mock

RestTemplate restTemplate;

@InjectMocks

private EmployeeService empService = new EmployeeService();

@Test

public void mocktTest(){

Employee emp = new Employee(“E001”, “Sidharth”);

Mockito.

when(restTemplete.getForEntity(

“<http://localhost:8080/employee/E001>”, Employee.class))

.thenReturn(new ResponseEntity(emp, HttpStatus.OK));

Employee employee = empService.getEmployee(id);

Assert,assertEquals(emp, employee);

}

**MockMvc**

@Test

**public** **void** getAllEmployeesAPI() **throws** Exception

{

mvc.perform(MockMvcRequestBuilders

.get("/employees")

.accept(MediaType.APPLICATION\_JSON))

.andDo(print())

.andExpect(status().isOk())

.andExpect(MockMvcResultMatchers.jsonPath("$.employees").exists())

.andExpect(MockMvcResultMatchers.jsonPath("$.employees[\*].employeeId").isNotEmpty());

}

**Mock static**

@Test

**public** **void** mockStaticMethodTest() {

*//Mock static methods*

PowerMockito.mockStatic(Service.**class**);

*//Set expectation*

Mockito.when(Service.staticMessage()).thenReturn("New Message from Mock!");

*//invoke the method*

String message = Service.staticMessage();

*//Assert the stub response*

Assert.assertEquals(message, "New Message from Mock!");

}

PowerMockito.verifyStatic(Service.**class**, Mockito.times(1));

Service.staticMessage();

**Mock final**

**public** **void** mockFinalMethodTest() {

*//Mock final method*

Service serviceMock = PowerMockito.mock(Service.**class**);

*//Set expectation*

Mockito.when(serviceMock.finalMessage()).thenReturn("New Message from Mock!");

*//invoke the method*

String message = serviceMock.finalMessage();

*//Assert the stub response*

Assert.assertEquals(message, "New Message from Mock!");

}

Mockito.verify(serviceMock).finalMessage();

**Mock private**

@Test

**public** **void** mockPrivateMethodTest() **throws** Exception {

Service mock = PowerMockito.spy(**new** Service());

Mockito.doReturn("New Message from Mock!").when(mock,"privateMessage");

String privateMessage = Whitebox.invokeMethod(mock, "privateMessage");

Assert.assertEquals(privateMessage, "New Message from Mock!");

}

PowerMockito.verifyPrivate(mock, times(1)).invoke("privateMessage");

**JPA test**

@DataJpaTest

@Import(JpaConfig.**class**)

**public** **class** TestBootstrappingEntityManager {

@Autowired

**private** TestEntityManager em;

@Autowired

**private** EmployeeRepository repository;

@Test

**public** **void** contextLoads() {

Assertions.assertNotNull(em);

}

@Test

**void** verifyBootstrappingByPersistingAnEmployee() {

Employee emp = **new** Employee();

emp.setEmail("demo-user@email.com");

emp.setFirstName("demo");

emp.setLastName("user");

Assertions.assertNull(emp.getEmployeeId());

em.persist(emp);

Assertions.assertNotNull(emp.getEmployeeId());

}

Mocking save method of repository

@Test

public void testCreateUserWhenSaved() {

//Create one sample userDto object with test data

when(mockedUserRepository.findOne(userDto.getId())).thenReturn(null);

when(mockedUserRepository.findOneByLogin(userDto.getId())).thenReturn(null);

//Create sample User object and set all the properties

User newUser=new User();

when(mockedUserRepository.save(Mockito.any(User.class)).thenReturn(newUser);

User returnedUser=userService.create(userDto);

//You have two ways to test, you can either verify that save method was invoked by

//verify method

verify(mockedUserRepository, times(1)).save(Mockito.any(User.class));

//or by assertion statements, match the authToken in the returned object to be equal

//to the one set by you in the mocked object

Assert.assertEquals(returnedUser.getAuthToken(),newUser.getAuthToken());

}

@Test

public void save\_book\_OK() throws Exception {

Book newBook = new Book(1L, "Mockito Guide", "mkyong");

when(mockRepository.save(any(Book.class))).thenReturn(newBook);

mockMvc.perform(post("/books")

.content("{json}")

.header(HttpHeaders.CONTENT\_TYPE, MediaType.APPLICATION\_JSON))

.andExpect(status().isCreated());

}

**Asserting Exceptions**

@Test **public** **void** **whenExceptionThrown\_thenAssertionSucceeds**() {

**Exception** exception = assertThrows(NumberFormatException.class, () -> { Integer.parseInt("1a"); });

**String** expectedMessage = "For input string";

**String** actualMessage = exception.getMessage(); assertTrue(actualMessage.contains(expectedMessage));

}

**Junit @TimeOut**

@BeforeEach

@Timeout(2)

**void** setUp() {

*// fails if execution time exceeds 2 seconds*

}

@Test

**void** testGetValue() **throws** InterruptedException {

Assertions.assertTimeout(Duration.ofSeconds(3), () -> {

getValue();

});

}

String getValue() **throws** InterruptedException {

TimeUnit.SECONDS.sleep(2);

**return** "";

}

**Different assertions**

**AssertArrayEquals**

**AssertEquals**

**AssertTrue and AssertFalse**

**AsseetNull and AssertNotNull**

**AssertSame and AssertNotSame**

**Fail**

**AssertAll**

**AssertIterableEquals**

**AssertLinesMatch**

**AssertNotEquals**

**AssertThrows**

**AssertTimeout and AssertTimeoutPreemptively**

In Java, how will you decide when to use a List, Set or a Map collection?

1. If we want a Collection that does not store duplicate values, then we use a Set based collection.

2. If we want to frequently access elements operations based on an index value then we use a List based collection. E.g. ArrayList

3. If we want to maintain the insertion order of elements in a collection then we use a List based collection.

4. For fast search operation based on a key, value pair, we use a HashMap based collection.

5. If we want to maintain the elements in a sorted order, then we use a TreeSet based collection.

Main differences between a HashSet and TreeSet are:

1. Ordering: In a HashSet elements are stored in a random order. In a TreeSet, elements are stored according to natural ordering.

2. Null Value Element: We can store null value object in a HashSet. A TreeSet does not allow to add a null value object.

3. Performance: HashSet performs basic operations like add(), remove(), contains(), size() etc in a constant size time. A TreeSet performs these operations at the order of log(n) time.

4. Speed: A HashSet is better than a TreeSet in performance for most of operations like add(), remove(), contains(), size() etc .

5. Internal Structure: a HashMap in Java internally backs a HashSet. A NavigableMap backs a TreeSet internally.

6. Features: A TreeSet has more features compared to a HashSet. It has methods like pollFirst(), pollLast(), first(), last(), ceiling(), lower() etc.

7. Element Comparison: A HashSet uses equals() method for comparison. A TreeSet uses compareTo() method for comparison to maintain ordering of elements.

In which scenario, LinkedList is better than ArrayList in Java?

ArrayList is more popular than LinkedList in Java due to its ease of use and random access to elements feature. But LinkedList is better in the scenario when we do not need random access to elements or there are a lot of insertion, deletion of elements.

The main difference between a HashSet and a HashMap are:

1. Base class: A HashSet class implements the Set interface. Whereas a HashMap class implements the Map interface.

2. Storage: A HashSet is used to store distinct objects. A HashMap is used for storing key & value pairs, so that these can be retrieved by key later on.

3. Duplicate Elements: A HashSet does not allow storing duplicate elements. A HashMap also does not allow duplicate keys. But we can store duplicate values in a HashMap.

4. Null Elements: In a HashSet we can store a single null value. In a HashMap we can store single null key, but any number of null values.

5. Element Type: A HashSet contains only values of objects as its elements. Whereas a HashMap contains entries(key value pairs).

6. Iteration: By using an Iterator we can iterate a HashSet. But a HashMap has to be converted into Set for iteration.

Main differences between a List and a Set are:

1. Order: List collection is an ordered sequence of elements. A Set is just a distinct collection of elements that is unordered.

2. Positional Access: When we use a List, we can specify where exactly we want to insert an element. In a Set there is no order, so we can insert element anywhere without worrying about order.

3. Duplicate: In a List we can store duplicate elements. A Set can hold only unique elements.

Main differences between a Set and a Map in Java are:

1. Duplicate Elements: A Set does not allow inserting duplicate elements. A Map does not allow using duplicate keys, but it allows inserting duplicate values for unique keys.

2. Null values: A Set allows inserting maximum one null value. In a Map we can have single null key at most and any number of null values.

3. Ordering: A Set does not maintain any order of elements. Some of sub-classes of a Set can sort the elements in an order like LinkedHashSet. A Map does not maintain any order of its elements. Some of its sub-classes like TreeMap store elements of the map in ascending order of keys.

Main differences between ArrayList and LinkedList data structures are:

1. Data Structure: An ArrayList is an indexed based dynamic array. A LinkedList is a Doubly Linked List data structure.

2. Insertion: It is easier to insert new elements in a LinkedList, since there is no need to resize an array. Insertion in ArrayList is O(n), since it may require resizing of array and copying its contents to new array.

3. Remove elements: LinkedList has better performance in removal of elements than ArrayList.

4. Memory Usage: LinkedList uses more memory than ArrayList, since it has to maintain links for next and previous nodes as well.

5. Access: LinkedList is slower in accessing an element, since we have to traverse the list one by one to access the right location.

ArrayList

* ***add()*** – takes *O(1)* time; however, worst-case scenario, when a new array has to be created and all the elements copied to it, it's *O(n)*
* ***add(index, element)*** – on average runs in *O(n)* time
* ***get()*** – is always a constant time *O(1)* operation
* ***remove()*** – runs in linear *O(n)* time. We have to iterate the entire array to find the element qualifying for removal.
* ***indexOf()***– also runs in linear time. It iterates through the internal array and checks each element one by one, so the time complexity for this operation always requires *O(n)* time.
* ***contains()*** – implementation is based on *indexOf(),*so it'll also run in *O(n)* time.

LinkedList

* ***add()*** – appends an element to the end of the list. It only updates a tail, and therefore, it's *O(1)* constant-time complexity.
* ***add(index, element)*** – on average runs in *O(n)* time
* ***get()*** – searching for an element takes *O(n)*time.
* ***remove(element)*** – to remove an element, we first need to find it. This operation is *O(n).*
* ***remove(index)*** – to remove an element by index, we first need to follow the links from the beginning; therefore, the overall complexity is *O(n).*
* ***contains()*** – also has *O(n)* time complexity

ArrayDeque

LIFO & FIFO – O(1)

3x better than LinkedList

HashSet

Very fast lookup, insertion and deletion

LinkedHashSet

Insertion order preserved + still very fast– O(1)

TreeSet

Sorting+ still very fast - O(1)

# [**What is meant by local repository and remote repository in Maven?**](https://stackoverflow.com/questions/13697021/what-is-meant-by-local-repository-and-remote-repository-in-maven-speak)

A local repository is a local directory structure that caches artifacts downloaded from remote repositories, or those that are manually installed (eg from the command line option).

**Maven local repository** reside in the developer’s machine. Whenever you run maven goals which require these dependencies, maven will download the dependencies from remote servers and store them into developer’s machine.

By default, Maven create the local repository inside user home directory i.e. C:/Users/superdev/.m2 directory. You can change the location of the local repository in [setting.xml](https://howtodoinjava.com/maven/maven-settings-file/) file using localRepository tag.

## Central repository

**Maven central repository** is located at [http://repo.maven.apache.org/maven2/](https://repo.maven.apache.org/maven2/). Whenever you run build job, maven first try to find dependency from local repository. If it is not there, then, by default, maven will trigger the download from this central repository location.

A remote repository is a web service (defined by a URL) that contains versioned artifacts. This might be as simple as an Apache server, or a full-blown Maven repository, such as [Artifactory](http://www.jfrog.com/home/v_artifactory_opensource_overview), that allows uploading, permissions based on a user directory, etc.

## Remote repository

Apart from central repository, you may have needed artifacts deployed on other remote locations. For example, in your corporate office there may be projects or modules specific to organization only. In this cases, organization can create remote repository and deploy these **private artifacts**. This remote repository will be accessible only inside organization.

These **maven remote repository** work exactly same way as maven’s central repository. Whenever an artifact is needed from these repositories, it is first downloaded to developer’s local repository and then it is used.

**Containerization**

Containerization is the process of bundling the software code with all its libraries, frameworks, and other dependencies so that it can be packaged and shipped as a single entity.

The software or application within the container can be moved and run consistently in any environment in any infrastructure be it on-premise or cloud. It is independent of that environment or infrastructure’s operating system. The container is basically a fully functional and portable computing environment.

Docker

#### **CONTAINERS**

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

#### **VIRTUAL MACHINES**

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Containarized environment

How microservices connect?

MQ

Event hub

Azure Event Hubs is a big data streaming platform and event ingestion service. It can receive and process millions of events per second. Data sent to an event hub can be transformed and stored by using any real-time analytics provider or batching/storage adapters.

The following scenarios are some of the scenarios where you can use Event Hubs:

* Anomaly detection (fraud/outliers)
* Application logging
* Analytics pipelines, such as clickstreams
* Live dashboards
* Archiving data
* Transaction processing
* User telemetry processing
* Device telemetry streaming

**Why use Event Hubs?**

Data is valuable only when there's an easy way to process and get timely insights from data sources. Event Hubs provides a distributed stream processing platform with low latency and seamless integration, with data and analytics services inside and outside Azure to build your complete big data pipeline.

Event Hubs represents the "front door" for an event pipeline, often called an *event ingestor* in solution architectures. An event ingestor is a component or service that sits between event publishers and event consumers to decouple the production of an event stream from the consumption of those events. Event Hubs provides a unified streaming platform with time retention buffer, decoupling event producers from event consumers.

Why cloud?

## Reduced IT costs

Moving to cloud computing may reduce the cost of managing and maintaining your IT systems. Rather than purchasing expensive systems and equipment for your business, you can reduce your costs by using the resources of your cloud computing service provider. You may be able to reduce your operating costs because:

* the cost of system upgrades, new hardware and software may be included in your contract
* you no longer need to pay wages for expert staff
* your energy consumption costs may be reduced
* there are fewer time delays.

## Scalability

Your business can scale up or scale down your operation and storage needs quickly to suit your situation, allowing flexibility as your needs change. Rather than purchasing and installing expensive upgrades yourself, your cloud computer service provider can handle this for you. Using the cloud frees up your time so you can get on with running your business.

## Business continuity

Protecting your data and systems is an important part of [business continuity planning](https://www.business.qld.gov.au/running-business/protecting-business/risk-management/continuity-planning). Whether you experience a natural disaster, power failure or other crisis, having your data stored in the cloud ensures it is backed up and protected in a secure and safe location. Being able to access your data again quickly allows you to conduct business as usual, minimising any downtime and loss of productivity.

## Collaboration efficiency

Collaboration in a cloud environment gives your business the ability to communicate and share more easily outside of the traditional methods. If you are working on a project across different locations, you could use cloud computing to give employees, contractors and third parties access to the same files. You could also choose a cloud computing model that makes it easy for you to share your records with your advisers (e.g. a quick and secure way to share accounting records with your accountant or financial adviser).

## Flexibility of work practices

Cloud computing allows employees to be more flexible in their work practices. For example, you have the ability to access data from home, on holiday, or via the commute to and from work (providing you have an internet connection). If you need access to your data while you are off-site, you can connect to your virtual office, quickly and easily.

## Access to automatic updates

Access to automatic updates for your IT requirements may be included in your service fee. Depending on your cloud computing service provider, your system will regularly be updated with the latest technology. This could include up-to-date versions of software, as well as upgrades to servers and computer processing power.

### **What is SaaS?**

Software as a service (or SaaS) is a way of delivering applications over the Internet—as a service. Instead of installing and maintaining software, you simply access it via the Internet, freeing yourself from complex software and hardware management.

SaaS applications are sometimes called Web-based software, on-demand software, or hosted software. Whatever the name, SaaS applications run on a SaaS provider’s servers. The provider manages access to the application, including security, availability, and performance.

**PASS**

Platform as a service (PaaS) is a complete development and deployment environment in the cloud, with resources that enable you to deliver everything from simple cloud-based apps to sophisticated, cloud-enabled enterprise applications. You purchase the resources you need from a [cloud service provider](https://azure.microsoft.com/en-in/overview/choosing-a-cloud-service-provider/) on a pay-as-you-go basis and access them over a secure Internet connection.

Like [IaaS](https://azure.microsoft.com/en-in/overview/what-is-iaas/), PaaS includes infrastructure—servers, storage and networking—but also middleware, development tools, business intelligence (BI) services, database management systems and more. PaaS is designed to support the complete web application lifecycle: building, testing, deploying, managing and updating.

PaaS allows you to avoid the expense and complexity of buying and managing software licenses, the underlying application infrastructure and middleware, container orchestrators such as [Kubernetes](https://azure.microsoft.com/en-in/topic/what-is-kubernetes/) or the development tools and other resources. You manage the applications and services you develop and the cloud service provider typically manages everything else.

**IAAS**

Infrastructure as a service (IaaS) is a type of cloud computing service that offers essential compute, storage and networking resources on demand, on a pay-as-you-go basis. IaaS is one of the four types of cloud services, along with software as a service ([SaaS](https://azure.microsoft.com/en-in/overview/what-is-saas/)), platform as a service ([PaaS](https://azure.microsoft.com/en-in/overview/what-is-paas/)) and [serverless](https://azure.microsoft.com/en-in/overview/serverless-computing/).

Migrating your organisation's infrastructure to an IaaS solution helps you reduce maintenance of on-premises data centres, save money on hardware costs and gain real-time business insights. IaaS solutions give you the flexibility to scale your IT resources up and down with demand. They also help you quickly provision new applications and increase the reliability of your underlying infrastructure.

IaaS lets you bypass the cost and complexity of buying and managing physical servers and datacentre infrastructure. Each resource is offered as a separate service component and you only pay for a particular resource for as long as you need it. A [cloud computing service provider](https://azure.microsoft.com/en-in/overview/choosing-a-cloud-service-provider/) like [Azure](https://azure.microsoft.com/en-in/overview/what-is-azure/iaas/) manages the infrastructure, while you purchase, install, configure and manage your own software—including operating systems, middleware and applications.

**Serverless**

Serverless computing enables developers to build applications faster by eliminating the need for them to manage infrastructure. With serverless applications, the cloud service provider automatically provisions, scales and manages the infrastructure required to run the code.

In understanding the definition of serverless computing, it is important to note that servers are still running the code. The serverless name comes from the fact that the tasks associated with infrastructure provisioning and management are invisible to the developer. This approach enables developers to increase their focus on the business logic and deliver more value to the core of the business. Serverless computing helps teams increase their productivity and bring products to market faster and it allows organisations to better optimise resources and stay focused on innovation.

Exception detailing @Service and @Bean

Microservices

A [microservice](https://www.bmc.com/blogs/devops-microservices/) is a small, single service offered by a company. It derives from the distributed computing architecture that connects many small services, rather than having one large service. The microservice can then be delivered through an application programming interface (API).

An API is a method of communication between a requester and a host, most often accessible through an IP address. The API can communicate multiple types of information to users, such as:

* Data you want to share
* A function you want to provide

Highly maintainable and testable

Loosely coupled

Independently deployable

Organized around business capabilities

Owned by a small team

Microservices (or microservices architecture) are a [cloud native](https://www.ibm.com/in-en/cloud/learn/cloud-native) architectural approach in which a single application is composed of many loosely coupled and independently deployable smaller components, or services. These services typically

* have their own technology stack, inclusive of the database and data management model;
* communicate with one another over a combination of [REST APIs](https://www.ibm.com/in-en/cloud/learn/rest-apis), event streaming, and [message brokers](https://www.ibm.com/in-en/cloud/learn/message-brokers); and
* are organized by business capability, with the line separating services often referred to as a bounded context.

While much of the discussion about microservices has revolved around architectural definitions and characteristics, their value can be more commonly understood through fairly simple business and organizational benefits:

* Code can be updated more easily - new features or functionality can be added without touching the entire application
* Teams can use different stacks and different programming languages for different components.
* Components can be scaled independently of one another, reducing the waste and cost associated with having to scale entire applications because a single feature might be facing too much load.

Advantages and Disadvatages

Why Microservices?

### **What are microservices?**

Microservices are a modern approach to software whereby application code is delivered in small, manageable pieces, independent of others.

### **Why build microservices?**

Their small scale and relative isolation can lead to many additional benefits, such as easier maintenance, improved productivity, greater fault tolerance, better business alignment, and more.

**Principles of microservices:**

* **Single responsibility:** It is one of the principles defined as a part of SOLID design pattern. It states that a single unit, either a class, a method, or a microservice should have one and only one responsibility. Each microservice must have a single responsibility and provide a single functionality. You can also say that: the number of microservices you should develop is equal to the number of functionalities you require. The database is also decentralized and generally, each microservice has its own database.
* **Built around business capabilities:** In today’s world, where so many technologies exist, there is always a technology which is best suited for implementing a particular functionality. But in monolithic applications, it was a major drawback, as we can’t use different technology for each functionality and hence, need to compromise in particular areas. A microservice shall never restrict itself from adopting appropriate technology stack or backend database storage which is most suitable for solving the business purpose i.e. each microservice can use different technology based on business requirements.
* **Design for failure:** Microservices must be designed with failure cases in mind. Microservices must exploit the advantage of this architecture and going down of one microservice should not affect the whole system, other functionalities must remain accessible to the user. But this was not the case in the Monolithic applications, where failure of one module leads to downfall of the whole application.
* **Highly Available**
* **Continous Integrated**
* **Loosely coupled**
* **Decentralized data**
* **Scalable**

**Advantages of microservices:**

* It is easy to manage as it is relatively smaller in size.
* If there’s any update in one of the microservices, then we need to redeploy only that microservice.
* Microservices are self-contained and hence, deployed independently. Their start-up and deployment time are relatively less.
* It is very easy for a new developer to on-board the project as he needs to understand only a particular microservice providing the functionality he will be working on and not the whole system.
* If a particular microservice is facing a large load because of the users using that functionality in excess then we need to scale out that microservice only. Hence, microservices architecture supports horizontal scaling.
* Each microservice can use different technology based on the business requirements.
* If a particular microservice goes down due to some bug, then it doesn’t affect other microservices and the whole system remains intact, continues providing other functionalities to the users.

**Disadvantages of microservices:**

* Being a distributed system, it is much more complex than monolithic applications. Its complexity increases with the increase in number of microservices.
* Skilled developers are required to work with microservices architecture which can identify the microservices and manage their inter-communications.
* Independent deployment of microservices is complicated.
* Microservices are costly in terms of network usage as they need to interact with each other and all these remote calls results into network latency.
* Microservices are less secure relative to monolithic applications due to the inter-services communication over the network.
* Debugging is difficult as the control flows over many microservices and to point out why and where exactly the error occurred is a difficult task.

Small services

Loose coupling

Scalability

Can we have two DBs for one microservice

Swagger Documentation

Can we define customized properties in yaml files.

Why are you looking for job change? That too from big org?

What are the challenges you faced recently in java?

Last work you have done?

this and super keyword in java?

Can we override static methods?

Prototype Pattern

Usually we program to an interface or a supertype – it is a good object-oriented design principle.

In these cases, we have access to the methods and variables that are defined in the interface. This is a drawback of abstraction.

Docker

Whenever we run a docker container, it is a part of bridge network(internal docker network). We will not be able to access the container unless it is exposed. Container port to host port.

Enumeration

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

Enums are used to create our own data type like classes. The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java.

**class** EnumExample1{

//defining enum within class

**public** **enum** Season { WINTER, SPRING, SUMMER, FALL }

//creating the main method

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

//printing all enum

**for** (Season s : Season.values()){

System.out.println(s);

}

System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));

System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());

System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());

}}

Accessors in java

One method to **get** the value of the private data item -> **Getter** or **Accessor** or **Get** method  
One method to **set** the value of the private data item -> **Setter** or **Mutator** or  **Set** method

Primitives and Non-primitives?

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

byte, short, int, long, float, double, boolean and char

Non-primitive data types - such as [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp)

Can we copy constructors?How?

A copy constructor in a Java class is a [constructor](https://www.baeldung.com/java-constructors) that**creates an object using another object of the same Java class**.

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

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

**private** String name;

**public** **Employee**(Employee employee) {

this.id = employee.id;

this.name = employee.name;

}

}

## Copy Constructor Vs clone() Method

Both the copy constructor and the clone() method are used to create a copy of an existing object of the class. But the use of copy constructor is easier and better in comparison to the clone() method because of the reasons given below:

* If we are using the clone() method it is necessary to import the **Cloneable** The method may throw the exception **CloneNotSupportException.** So, handling the exception in a program is a complex task. While in copy constructor there are no such complexities.
* We cannot assign a value if the fields are **final**. While in the copy constructor we can assign values to the final fields.
* The object returned by the clone() method must be **typecast**. While in copy constructor there is no such requirement.

Which methods will get invoked first static/abstract?

A **static block** is a block of code with a static keyword. In general, these are used to initialize the static members of a class. JVM executes static blocks before the main method at the time loading a class.

Collections and stream? Why stream came when collections were there?

A **Collection**is an in-memory data structure, which holds all the values that the data structure currently has. Every element in the Collection has to be computed before we add it to the Collection. Operations such as searching, sorting, insertion, manipulation, and deletion can be performed on a Collection. It provides many interfaces like ([Set](https://www.geeksforgeeks.org/set-in-java/), [List](https://www.geeksforgeeks.org/list-interface-java-examples/), [Queue](https://www.geeksforgeeks.org/list-interface-java-examples/), [Deque](https://www.geeksforgeeks.org/deque-interface-java-example/)) and Classes like ([ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/), [Vector](https://www.geeksforgeeks.org/java-util-vector-class-java/), [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/), [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/), [HashSet](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/)).

On the other hand, IStream is an API that is introduced in Java 8 which is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result. The Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods that can be pipelined to produce the desired result.

**The features of the Java stream are:**

* A stream is not a data structure instead it takes input from the Collections, Arrays, or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods. Streams only provide the result as per the pipelined methods and don’t change the original data structure.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Java Collections framework is used for storing and manipulating group of data. It is an in-memory data structure and every element in the collection should be computed before it can be added in the collections.

Stream API is only used for processing group of data. It does not modify the actual collection, they only provide the result as per the pipelined methods.

| **Sr. No.** | **Key** | **Collections** | **Streams** |
| --- | --- | --- | --- |
| 1 | Basic | It is used for storing and manipulating group of data | Stream API is only used for processing group of data |
| 2 | Package | All the classes and interfaces of this API is in the Java.util package | All the classes and interfaces of this API is in the java.util.stream  package |
| 3 | Eager/Lazy | All the elements in the collections are computed in the beginning. | In streams, intermediate operations are lazy. |
| 4. | Data Modification | In collections, we can remove or add elements. | We can’t modify streams. |
| 5 | External /Internal iterator | Collections perform iteration over the collection. | Stream perform iteration   internally. |

**Collection vs Collections:**

| **Collection** | **Collections** |
| --- | --- |
| It is an interface. | It is a utility class. |
| It is used to represent a group of individual objects as a single unit. | It defines several utility methods that are used to operate on collection. |
| The Collection is an interface that contains a static method since java8. The Interface can also contain abstract and default methods. | It contains only static methods. |

Java Collectors

Java Collectors is a utility class that provides many useful implementations of the Collector interface. The Collector implementation is used with the [Stream collect()](https://www.journaldev.com/32457/java-stream-collect-method-examples) method. This class was introduced in Java 8 along with Stream API. The Collectors is a final class and all the methods are static that returns the Collector instance.

## Java Collectors Examples

Let’s look at the examples of Collectors functions with simple examples.

### 1. toCollection(Supplier)

This function returns a Collector that accumulates the input elements into a collection.

List<String> strList = Arrays.asList("a", "b", "c", "b", "a");

// toCollection()

Collection<String> strCollection = strList.parallelStream().collect(Collectors.toCollection(HashSet::new));

System.out.println(strCollection); // [a, b, c]

Set<String> strSet = strList.parallelStream().collect(Collectors.toCollection(HashSet::new));

System.out.println(strSet); // [a, b, c]

List<String> strList1 = strList.parallelStream().sorted(String::compareToIgnoreCase)

.collect(Collectors.toCollection(ArrayList::new));

System.out.println(strList1); // [a, a, b, b, c]

### 2. Java Collectors toList()

It returns a Collector that accumulates the input elements into a new List.

List<String> strList = Arrays.asList("a", "b", "c", "b", "a");

List<String> uppercaseList = strList.parallelStream().map(String::toUpperCase).collect(Collectors.toList());

System.out.println(uppercaseList); // [A, B, C, B, A]

List<String> uppercaseUnmodifiableList = strList.parallelStream().map(String::toUpperCase)

.collect(Collectors.toUnmodifiableList());

System.out.println(uppercaseUnmodifiableList); // [A, B, C, B, A]

Java Collectors toList

### 3. Java Collectors toSet()

This method returns the Collector that accumulates the input elements into a new Set.

List<String> strList = Arrays.asList("a", "b", "c", "b", "a");

Set<String> uppercaseSet = strList.parallelStream().map(String::toUpperCase).collect(Collectors.toSet());

System.out.println(uppercaseSet); // [A, B, C]

Set<String> uppercaseUnmodifiableSet = strList.parallelStream().map(String::toUpperCase)

.collect(Collectors.toUnmodifiableSet());

System.out.println(uppercaseUnmodifiableSet); // [A, B, C]

### 4. Java Collectors toMap(Function, Function)

This static method returns a Collector to accumulate input elements into a new Map. We have to specify the mapping functions to generate the map keys and values.

Map<String, String> map = Stream.of("a", "b", "c")

.collect(Collectors.toMap(Function.identity(), String::toUpperCase));

System.out.println(map); // {a=A, b=B, c=C}

// Duplicate Keys will throw: Exception in thread "main"

// java.lang.IllegalStateException: Duplicate key a (attempted merging values A

// and A)

Map<String, String> mapD = Stream.of("a", "b", "c", "b", "a")

.collect(Collectors.toMap(Function.identity(), String::toUpperCase, String::concat));

System.out.println(mapD); // {a=AA, b=BB, c=C}

// above are HashMap, use below to create different types of Map

TreeMap<String, String> mapTree = Stream.of("a", "b", "c", "b")

.collect(Collectors.toMap(Function.identity(), String::toUpperCase, String::concat, TreeMap::new));

System.out.println(mapTree); {a=A, b=BB, c=C}

### 5. Java Collectors joining()

It returns a Collector that concatenate the input CharSequence elements into a new string. There are few overloaded methods to specify the delimiter and suffix/prefix strings.

String concat = Stream.of("a", "b").collect(Collectors.joining());

System.out.println(concat); // ab

String csv = Stream.of("a", "b").collect(Collectors.joining(","));

System.out.println(csv); // a,b

String csv1 = Stream.of("a", "b").collect(Collectors.joining(",", "[", "]"));

System.out.println(csv1); // [a,b]

String csv2 = Stream.of("a", new StringBuilder("b"), new StringBuffer("c")).collect(Collectors.joining(","));

System.out.println(csv2); // a,b

### 6. Java Collectors mapping(Function, Collector)

This method returns a Collector that applies the Function to the input elements and then accumulate them to the given Collector.

Set<String> setStr = Stream.of("a", "a", "b")

.collect(Collectors.mapping(String::toUpperCase, Collectors.toSet()));

System.out.println(setStr); // [A, B]

Set<String> setStr1 = Stream.of("a", "a", "b")

.collect(Collectors.flatMapping(s -> Stream.of(s.toUpperCase()), Collectors.toSet()));

System.out.println(setStr1); // [A, B]

### 7. filtering(Predicate, Collector)

It returns a Collector that applies the Predicate to the input elements and accumulate them to the given Collector if the predicate returns true.

List<String> strList2 = List.of("1", "2", "10", "100", "20", "999");

Set<String> set = strList2.parallelStream()

.collect(Collectors.filtering(s -> s.length() < 2, Collectors.toSet()));

System.out.println(set); // [1, 2]

### 8. collectingAndThen(Collector, Function)

This static method returns a Collector that accumulates the input elements into the given Collector and then perform an additional finishing function.

List<String> strList2 = List.of("1", "2", "10", "100", "20", "999");

List<String> unmodifiableList = strList2.parallelStream()

.collect(Collectors.collectingAndThen(Collectors.toList(), Collections::unmodifiableList));

System.out.println(unmodifiableList); // [1, 2, 10, 100, 20, 999]

### 9. counting()

This function returns a Collector that counts the number of the input elements.

Long evenCount = Stream.of(1, 2, 3, 4, 5).filter(x -> x % 2 == 0).collect(Collectors.counting());

System.out.println(evenCount); // 2

### 10. minBy(Comparator)

It returns a Collector that returns the minimum element based on the given comparator.

Optional<Integer> min = Stream.of(1, 2, 3, 4, 5).collect(Collectors.minBy((x, y) -> x - y));

System.out.println(min); // Optional[1]

### 11. maxBy(Comparator)

This method returns a Collector that returns the maximum element based on the given comparator.

Optional<Integer> max = Stream.of(1, 2, 3, 4, 5).collect(Collectors.minBy((x, y) -> y - x));

System.out.println(max); // Optional[5]

### 12. summingInt(ToIntFunction)

This static method returns a Collector that produces the sum of a integer-valued function applied to the input elements. There are similar functions for long and double – summingLong(ToLongFunction) and summingDouble(ToDoubleFunction).

List<String> strList3 = Arrays.asList("1", "2", "3", "4", "5");

Integer sum = strList3.parallelStream().collect(Collectors.summingInt(Integer::parseInt));

System.out.println(sum); // 15

Long sumL = Stream.of("12", "23").collect(Collectors.summingLong(Long::parseLong));

System.out.println(sumL); // 35

Double sumD = Stream.of("1e2", "2e3").collect(Collectors.summingDouble(Double::parseDouble));

System.out.println(sumD); // 2100.0

### 13. averagingInt(ToIntFunction)

It returns a Collector that produces the arithmetic mean of an integer-valued function applied to the input elements. There are similar functions for long and double – averagingLong(ToLongFunction) and averagingDouble(ToDoubleFunction).

List<String> strList4 = Arrays.asList("1", "2", "3", "4", "5");

Double average = strList4.parallelStream().collect(Collectors.averagingInt(Integer::parseInt));

System.out.println(average); // 3.0

Double averageL = Stream.of("12", "23").collect(Collectors.averagingLong(Long::parseLong));

System.out.println(averageL); // 17.5

Double averageD = Stream.of("1e2", "2e3").collect(Collectors.averagingDouble(Double::parseDouble));

System.out.println(averageD); // 1050.0

### 14. Java Collectors groupingBy(Function)

This method returns a Collector implementing a “group by” operation on input elements. The final result is a HashMap. There are few overloaded methods to specify the Supplier (final Map type, default is HashMap) and Collector (Value List type, default is ArrayList).

Map<Integer, List<Integer>> mapGroupBy = Stream.of(1, 2, 3, 4, 5, 4, 3)

.collect(Collectors.groupingBy(x -> x \* 10));

System.out.println(mapGroupBy); // {50=[5], 20=[2], 40=[4, 4], 10=[1], 30=[3, 3]}

### 15. groupingByConcurrent(Function)

It works in the same way as groupingBy() Collector. The only difference is that the Collector is concurrent and unordered. It will have better performance than the groupingBy() Collector but the ordering will not be maintained.

Map<Integer, List<Integer>> mapGroupBy = Stream.of(1, 2, 3, 4, 5, 4, 3)

.collect(Collectors.groupingByConcurrent(x -> x \* 10));

System.out.println(mapGroupBy); // {50=[5], 20=[2], 40=[4, 4], 10=[1], 30=[3, 3]}

### 16. partitioningBy(Predicate)

This method returns a Collector which partitions the input elements according to a Predicate, and organizes them into a Map<Boolean, List<T>>.

Map<Boolean, List<Integer>> mapPartitionBy = Stream.of(1, 2, 3, 4, 5, 4, 3)

.collect(Collectors.partitioningBy(x -> x % 2 == 0));

System.out.println(mapPartitionBy); // {false=[1, 3, 5, 3], true=[2, 4, 4]}

### 17. Java Collectors reducing(BinaryOperator)

It returns a Collector which performs a reduction of its input elements under a specified BinaryOperator. This is mostly used in a multi-level reduction, such as specifying downstream Collector with groupingBy() and partitioningBy() methods.

Map<Boolean, Optional<Integer>> reducing = Stream.of(1, 2, 3, 4, 5, 4, 3).collect(Collectors.partitioningBy(

x -> x % 2 == 0, Collectors.reducing(BinaryOperator.maxBy(Comparator.comparing(Integer::intValue)))));

System.out.println(reducing); // {false=Optional[5], true=Optional[4]}

### 18. summarizingInt(ToIntFunction)

It returns a Collector which applies an int-producing mapping function to each input element, and returns summary statistics for the resulting values such as min, max, average, count, and count.

IntSummaryStatistics summarizingInt = Stream.of("12", "23", "35")

.collect(Collectors.summarizingInt(Integer::parseInt));

System.out.println(summarizingInt);

//IntSummaryStatistics{count=3, sum=70, min=12, average=23.333333, max=35}

Synchronized and normal collections? Difference?

* Here, synchronize means only one thread is allowed to operate on an object at a time or in other words the object(which is synchronized) can't be modified by multiple threads simultaneously.
* **Synchronized Collection** can be modified by one thread at a time(i.e. it is not possible to modify or access Synchronized Collection by multiple threads simultaneously).
* **Synchronized Collection** has low performance than Concurrent Collection because at a time only one thread is allowed to operate on an object so it increases the waiting time of the threads.
* **Synchronized Collection** acquires the lock on the entire Collection object which provides thread-safety.
* **SynchronizedMap** is a static inner class of Collections class which is available in java.util.Collections.
* In **SynchronizedMap** it acquires the lock on entire Map object and it wraps all the methods of Map interface with synchronized keyword.

### **Concurrent Collection**

* Now, we will see what is Concurrent? Here, concurrent means only multiple threads are allowed to operate on an object at a time or in other words the object (which is concurrent) can be modified by multiple threads simultaneously.
* **Concurrent Collection** can be modified by multiple threads at a time(i.e. it is possible to modify or access Concurrent Collection by multiple threads simultaneously).
* **Concurrent Collection** has high performance than **Synchronized Collection** because at a time multiple threads are allowed to operate on an object so it decreases the waiting time of the threads.
* More than one threads can perform read-write operation concurrently still it provides Thread Safety.
* **ConcurrentHashMap** is a class introduced in Java 5 that is available in java.util package.
* **ConcurrentHashMap** divides Map object into different parts and every thread acquires the lock on every part.
* As we know default concurrency level is 16 that means maximum 16 threads are allowed to access an object simultaneously by default and we can increase and decrease concurrency level if we want.
* **ConcurrentHashMap** does not allow null keys and null values.

Return type of overrided child classes?

Java version 5.0 onwards it is possible to have different return types for an overriding method in the child class, but the child’s return type should be a subtype of the parent’s return type. The overriding method becomes **variant** with respect to return type.

The co-variant return type is based on the [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle).

Now geeks you must be wondering about why to use for which we will be listing down the advantages as follows:

* It helps to avoid confusing type casts present in the class hierarchy and thus making the code readable, usable and maintainable.
* We get the liberty to have more specific return types when overriding methods.
* Help in preventing run-time ClassCastExceptions on returns

// Java Program to Demonstrate Different Return Types

// if Return Type in Overridden method is Sub-type

// Class 1

class A {

}

// Class 2

class B extends A {

}

// Class 3

// Helper class (Base class)

class Base {

// Method of this class of class1 return type

A fun()

{

// Display message only

System.out.println("Base fun()");

return new A();

}

}

// Class 4

// Helper class extending above class

class Derived extends Base {

// Method of this class of class1 return type

B fun()

{

// Display message only

System.out.println("Derived fun()");

return new B();

}

}

// Class 5

// Main class

public class GFG {

// Main driver method

public static void main(String args[])

{

// Creating object of class3 type

Base base = new Base();

// Calling method fun() over this object

// inside main() method

base.fun();

// Creating object of class4 type

Derived derived = new Derived();

// Again calling method fun() over this object

// inside main() method

derived.fun();

}

}

Messaging in microservices?

There are several different styles of asynchronous communication:

* Request/response - a service sends a request message to a recipient and expects to receive a reply message promptly
* Notifications - a sender sends a message a recipient but does not expect a reply. Nor is one sent.
* Request/asynchronous response - a service sends a request message to a recipient and expects to receive a reply message eventually
* Publish/subscribe - a service publishes a message to zero or more recipients
* Publish/asynchronous response - a service publishes a request to one or recipients, some of whom send back a reply

Common logging in microservices?

#### 1. Use a correlation ID

A correlation ID is a unique identifier that developers use to segregate sets of operations and track individual requests. It doesn't matter how the correlation ID is generated, so long as it is unique, accessible to downstream services and diligently logged along with other important service call data.

If the transaction passing through multiple services has a correlation ID, troubleshooters can do an ID search in the logs and view data about the individual service calls, including the number of times a service was used. This way, the correlation ID can identify which service the transaction failure stemmed from.

#### 2. Provide informative application logs

When an error occurs, the log should include all the needed information pertaining to the issue. The more information troubleshooters have from the microservices' logs, the easier and more quickly they can [ascertain what went wrong](https://www.techtarget.com/searchapparchitecture/tip/Troubleshooting-microservices-performance-problems).

Logs should, at a bare minimum, include the following information:

* Service name
* Username
* IP address
* Correlation ID
* Message received time in UTC
* Time taken
* Method name
* Call stack

@componentscan and @Component?

While developing an application, we need to tell the Spring framework to look for Spring-managed components. **@ComponentScan enables Spring to scan for things like configurations, controllers, services, and other components we define**.

**The @EnableAutoConfiguration annotation enables Spring Boot to auto-configure the application context**. **Therefore, it automatically creates and registers beans based on both the included jar files in the classpath and the beans defined by us.**

@Autowired?

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

How do you define bean?

 A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container.

| **Sr. No.** | **Key** | **@Bean** | **@Component** |
| --- | --- | --- | --- |
| 1 | Auto detection | It is used to explicitly declare a single bean, rather than letting Spring do it automatically. | If any class is annotated with @Component it will be automatically detect by using classpath scan. |
| 2 | Spring Container | Bean can be created even class is outside the spring container | We can’t create bean if class is outside spring container |
| 3 | Class/Method  Level Annotation | It is a method level annotation | It is a class level annotation |
| 4 | @Configuration | It works only when class is also annotated with @Configuration | It works without @Configuration annotation |
| 5 | Use Case | We should use @bean, if you want specific implementation based on dynamic condition. | We can’t write specific implementation based on dynamic condition |

@Bean is used to explicitly declare a single bean, rather than letting Spring do it automatically. It decouples the declaration of the bean from the class definition and lets you create and configure beans exactly how you choose.

@Bean is a method-level annotation. We generally use it to configure beans in Java code (if you are not using XML configuration) and then call it from a class using the ApplicationContext.getBean() method.

@Configuration

class MyConfiguration{

@Bean

public User getUser() {

return new User();

}

}

class User{

}

How did you implemented microservices in your project?

How do you see issues in production env?

What will we do if we accidently commited wrong code to prod?

In Java, a thread is a lightweight process that runs within another process or thread. It is an independent path of execution in an application. Each thread runs in a separate stack frame. By default Java starts one thread when the main method of a class is called.

Daemon Thread

A daemon thread in Java is a low priority thread that does not prevent the JVM from exiting when the program finishes. The thread keeps running. Garbage Collection is an example of daemon thread.

Synchronization

The concept of Synchronization in Java is used in Multi-threading programming. It is a feature in Java that helps in controlling the access of multiple threads to a shared resource. It is used to prevent Deadlock between multiple threads.

# CopyOnWriteArrayList in Java

**CopyOnWriteArrayList class** is introduced in JDK 1.5, which implements the [**List interface**](https://www.geeksforgeeks.org/list-interface-java-examples/). It is an enhanced version of [**ArrayList**](https://www.geeksforgeeks.org/arraylist-in-java/)in which all modifications (add, set, remove, etc) are implemented by making a fresh copy. It is found in **java.util.concurrent** package. It is a data structure created to be used in a concurrent environment.

* As the name indicates, CopyOnWriteArrayList creates a Cloned copy of underlying ArrayList, for every update operation at a certain point both will be synchronized automatically, which is taken care of by JVM. Therefore, there is no effect for threads that are performing read operation.
* It is costly to use because for every update operation a cloned copy will be created. Hence, CopyOnWriteArrayList is the best choice if our frequent operation is read operation.
* The underlined data structure is a grow-able array.
* It is a thread-safe version of ArrayList.
* Insertion is preserved, duplicates, null, and heterogeneous Objects are allowed.
* The main important point about CopyOnWriteArrayList is the [Iterator](https://www.geeksforgeeks.org/iterators-in-java/) of CopyOnWriteArrayList can not perform remove operation otherwise we get Run-time exception saying **UnsupportedOperationException.**add() and set() methods on CopyOnWriteArrayList iterator also throws **UnsupportedOperationException.**Also Iterator of CopyOnWriteArrayList will never throw **ConcurrentModificationException**.

**Declaration:**

*public class CopyOnWriteArrayList<E> extends Object implements List<E>, RandomAccess, Cloneable, Serializable*

Here, E is the type of elements held in this collection.

**Note:** The class implements **Serializable**, **Cloneable**, **Iterable<E>**, **Collection<E>**, [List<E>](https://www.geeksforgeeks.org/list-interface-java-examples/), **RandomAccess** interfaces.

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

HashTable Vs HashMap

Both of them provide:

* Fail-fast iteration
* Unpredictable iteration order

But there are some differences too:

* *HashMap* doesn't provide any *Enumeration, while Hashtable* provides not fail-fast *Enumeration*
* *Hashtable* doesn't allow *null* keys and *null* values, while *HashMap* do allow one *null* key and any number of *null* values
* *Hashtable*‘s methods are synchronized while *HashMaps*‘s methods are not

@Controller and @RestController

**Difference between @Controller and @RestController in Spring MVC/BOOT**

1. The **@Controller**is a annotation to mark class as Controller Class in Spring While **@RestController** is used in REST Web services and similar to **@Controller** and **@ResponseBody**.
2. The **@Controller** annotation indicates that the class is controller like web Controller while **@RestController** annotation indicates that the class is controller where **@RequestMapping** Method assume **@ResponseBody** by Default(i.e REST APIs).
3. The key difference is that you do not need to use **@ResponseBody** on each and every handler method once you annotate the class with **@RestController**.
4. **@Controller** create a Map of Model Object and find a view while **@RestController** simply return object and object data directly written into http response as JSON orXML.

Schema of database

A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It’s the database designers who design the schema to help programmers understand the database and make it useful.

# **Indexing in DBMS**

* Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
* The index is a type of data structure. It is used to locate and access the data in a database table quickly.

Primary Index

* If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.
* As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.

Comparater and Comparable? Diff?

**Using Comparable Interface**

A comparable object is capable of comparing itself with another object. The class itself must implements the **java.lang.Comparable** interface to compare its instances.   
Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.

**Using Comparator**

Unlike Comparable, Comparator is external to the element type we are comparing. It’s a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.  
Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.  
To compare movies by Rating, we need to do 3 things : 

1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
2. Make an instance of the Comparator class.
3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

//A Java program to demonstrate Comparator interface

import java.io.\*;

import java.util.\*;

// A class 'Movie' that implements Comparable

class Movie implements Comparable<Movie>

{

private double rating;

private String name;

private int year;

// Used to sort movies by year

public int compareTo(Movie m)

{

return this.year - m.year;

}

// Constructor

public Movie(String nm, double rt, int yr)

{

this.name = nm;

this.rating = rt;

this.year = yr;

}

// Getter methods for accessing private data

public double getRating() { return rating; }

public String getName() { return name; }

public int getYear() { return year; }

}

// Class to compare Movies by ratings

class RatingCompare implements Comparator<Movie>

{

public int compare(Movie m1, Movie m2)

{

if (m1.getRating() < m2.getRating()) return -1;

if (m1.getRating() > m2.getRating()) return 1;

else return 0;

}

}

// Class to compare Movies by name

class NameCompare implements Comparator<Movie>

{

public int compare(Movie m1, Movie m2)

{

return m1.getName().compareTo(m2.getName());

}

}

// Driver class

class Main

{

public static void main(String[] args)

{

ArrayList<Movie> list = new ArrayList<Movie>();

list.add(new Movie("Force Awakens", 8.3, 2015));

list.add(new Movie("Star Wars", 8.7, 1977));

list.add(new Movie("Empire Strikes Back", 8.8, 1980));

list.add(new Movie("Return of the Jedi", 8.4, 1983));

// Sort by rating : (1) Create an object of ratingCompare

// (2) Call Collections.sort

// (3) Print Sorted list

System.out.println("Sorted by rating");

RatingCompare ratingCompare = new RatingCompare();

Collections.sort(list, ratingCompare);

for (Movie movie: list)

System.out.println(movie.getRating() + " " +

movie.getName() + " " +

movie.getYear());

// Call overloaded sort method with RatingCompare

// (Same three steps as above)

System.out.println("\nSorted by name");

NameCompare nameCompare = new NameCompare();

Collections.sort(list, nameCompare);

for (Movie movie: list)

System.out.println(movie.getName() + " " +

movie.getRating() + " " +

movie.getYear());

// Uses Comparable to sort by year

System.out.println("\nSorted by year");

Collections.sort(list);

for (Movie movie: list)

System.out.println(movie.getYear() + " " +

movie.getRating() + " " +

movie.getName()+" ");

}

}

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price. | The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. |
| 2) Comparable **affects the original class**, i.e., the actual class is modified. | Comparator **doesn't affect the original class**, i.e., the actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is present in **java.lang** package. | A Comparator is present in the **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method. |

Can we have 2 db in spring?

Yes.with @ConfigurationProperties

## Multiple Database Configurations in Spring Boot

Following is the application.properties file that contains configurations for multiple databases. You can notice that properties starting from spring.user.datasource has user database configuration and properties starting from spring.booking.datasource has booking datasource configurations. These configurations are used in the coming sections while configuring entitymanager and transactionmanager for respective DB connections.

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories(

entityManagerFactoryRef = "userEntityManager",

transactionManagerRef = "userTransactionManager",

basePackages = "com.devglan.user.dao"

)

public class UserDBConfig {

@Bean

@ConfigurationProperties(prefix = "spring.user.datasource")

public DataSource postgresqlDataSource() {

return DataSourceBuilder

.create()

.build();

}

@Bean(name = "userEntityManager")

public LocalContainerEntityManagerFactoryBean postgresqlEntityManagerFactory(EntityManagerFactoryBuilder builder) {

return builder

.dataSource(postgresqlDataSource())

.properties(hibernateProperties())

.packages(UserDetails.class)

.persistenceUnit("userPU")

.build();

}

@Bean(name = "userTransactionManager")

public PlatformTransactionManager postgresqlTransactionManager(@Qualifier("userEntityManager") EntityManagerFactory entityManagerFactory) {

return new JpaTransactionManager(entityManagerFactory);

}

private Map hibernateProperties() {

Resource resource = new ClassPathResource("hibernate.properties");

try {

Properties properties = PropertiesLoaderUtils.loadProperties(resource);

return properties.entrySet().stream()

.collect(Collectors.toMap(

e -> e.getKey().toString(),

e -> e.getValue())

);

} catch (IOException e) {

return new HashMap();

}

}

}

## How to Create an immutable class in Java?

To create an immutable class in Java, you have to do the following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables.
4. Make all **mutable fields final** so that its value can be assigned only once.
5. Initialize all the fields via a [constructor](https://www.journaldev.com/18899/constructor-in-java) performing deep copy.
6. Perform [cloning](https://www.journaldev.com/60/java-clone-object-cloning-java) of objects in the getter methods to return a copy rather than returning the actual object reference.

To understand points 4 and 5, let’s run the sample Final class that works well and values don’t get altered after instantiation.

package com.journaldev.java;

import java.util.HashMap;

import java.util.Iterator;

public final class FinalClassExample {

private final int id;

private final String name;

private final HashMap<String,String> testMap;

public int getId() {

return id;

}

public String getName() {

return name;

}

/\*\*

\* Accessor function for mutable objects

\*/

public HashMap<String, String> getTestMap() {

//return testMap;

return (HashMap<String, String>) testMap.clone();

}

/\*\*

\* Constructor performing Deep

\* @param i

\* @param n

\* @param hm

\*/

public FinalClassExample(int i, String n, HashMap<String,String> hm){

System.out.println("Performing Deep for Object initialization");

this.id=i;

this.name=n;

HashMap<String,String> tempMap=new HashMap<String,String>();

String key;

Iterator<String> it = hm.keySet().iterator();

while(it.hasNext()){

key=it.next();

tempMap.put(key, hm.get(key));

}

this.testMap=tempMap;

}

/\*\*

\* Constructor performing Shallow

\* @param i

\* @param n

\* @param hm

\*/

/\*\*

public FinalClassExample(int i, String n, HashMap<String,String> hm){

System.out.println("Performing Shallow for Object initialization");

this.id=i;

this.name=n;

this.testMap=hm;

}

\*/

/\*\*

\* To test the consequences of Shallow and how to avoid it with Deep for creating immutable classes

\* @param args

\*/

public static void main(String[] args) {

HashMap<String, String> h1 = new HashMap<String,String>();

h1.put("1", "first");

h1.put("2", "second");

String s = "original";

int i=10;

FinalClassExample ce = new FinalClassExample(i,s,h1);

//Lets see whether its copy by field or reference

System.out.println(s==ce.getName());

System.out.println(h1 == ce.getTestMap());

//print the ce values

System.out.println("ce id:"+ce.getId());

System.out.println("ce name:"+ce.getName());

System.out.println("ce testMap:"+ce.getTestMap());

//change the local variable values

i=20;

s="modified";

h1.put("3", "third");

//print the values again

System.out.println("ce id after local variable change:"+ce.getId());

System.out.println("ce name after local variable change:"+ce.getName());

System.out.println("ce testMap after local variable change:"+ce.getTestMap());

HashMap<String, String> hmTest = ce.getTestMap();

hmTest.put("4", "new");

System.out.println("ce testMap after changing variable from accessor methods:"+ce.getTestMap());

}

}

## Why Deep Copy is important for immutability?

Let’s comment the constructor providing [deep copy](https://www.journaldev.com/17129/java-deep-copy-object) and uncomment the constructor providing a shallow copy.

Also, uncomment the return statement in the getTestMap() method that returns the actual object reference.

Run the program after all the changes are done. It will produce the following output.

It’s happening because of the direct reference to the original object in the getter function.

Iterating over a Map?

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

System.out.println(entry.getKey() + “ : “ + entry.getValue());

}

**for** (String key : map.keySet()) { System.out.println(key + ":" + map.get(key)); }

Iterator<Integer, Integer> iterator = map.entrySet().iterator();

While(iterator.hasNext()){

Map.Entry<Integer, Integer> entry = iterator.next();

System.out.println(entry.getKey() + “ : “ entry.getValue());

}

map.forEach((k, v) -> System.out.println((“Key : “ + k + “ “ + “ Value : “ + v)));

map.entrySet().stream()

.forEach( e -> System.out.println(e.getKey() + “ “ + e.getValue()));

**Stack Memory**

**It is used for static memory allocation and the execution of a thread. It contains primitive values that are specific to a method and references to objects referred from the method that are in the heap.**

**Heap space**

**It is used for dynamic allocation of java objects and JRE classes at runtime.**

**Objects are always stored on heap space and references to these objects are stored in stack memory.**

**OutOfMemoryError**

In java, all the objects are stored in a heap. They are allocated using a new operator. Usually this occurs when the JVM cannot allocate an object because it is out of memory. NO more memory could be allocated by GC.

It usually means that you’re doing something wrong, either holding on to an objects too long or trying to process too much data at a time. Sometimes, it indicates a problem that’s out of your control, such as a third party that caches strings or an application server doesn’t clean up after deploys. And sometimes, it has nothing to do with objects on the heap.

What are the main differences  
between an interface with default  
method and an abstract class in Java  
8?  
An interface with a default method appears same as an Abstract  
class in Java. But there are subtle differences between two.  
1. Instance variable: An interface cannot have instance  
variables. An abstract class can have instance variables.  
2. Constructor: An interface cannot have a constructor. An  
abstract class can have constructor.  
3. Concrete Method: An interface cannot have concrete  
methods other than default method. An abstract class is  
allowed to define concrete methods with implementation.  
4. Lambda: An interface with exactly one default method can  
be used for lambda expression. An abstract class cannot be  
used for lambda expression

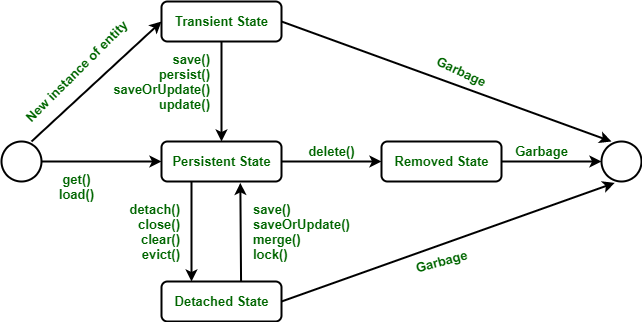
Why return type of main method is void?

**Java main method doesn't return anything**, that's why it's return type is void. This has been done to keep things simple because once the main method is finished executing, java program terminates. So there is no point in returning anything, there is nothing that can be done for the returned object by JVM.

Lifecycle of hibernate entity?

There are mainly four states of the Hibernate Lifecycle :

1. Transient State
2. Persistent State
3. Detached State
4. Removed State



**Hibernate Lifecycle**

As depicted from the above media one can co-relate how they are plotted in order to plot better in our mind. Now we will be discussing the states to better interpret hibernate lifecycle. It is as follows:

**State 1:** Transient State

The transient state is the first state of an entity object. When we instantiate an object of a [POJO class](https://www.geeksforgeeks.org/pojo-vs-java-beans/) using the new operator then the object is in the transient state. This object is not connected with any hibernate session. As it is not connected to any Hibernate Session, So this state is not connected to any database table. So, if we make any changes in the data of the POJO Class then the database table is not altered. Transient objects are independent of Hibernate, and they exist in the **heap memory**.

https://media.geeksforgeeks.org/wp-content/uploads/20210626215802/GFGTransientState.png

**Changing new object to Transient State**

There are two layouts in which transient state will occur as follows:

1. When objects are generated by an application but are not connected to any session.
2. The objects are generated by a closed session.

**State 2:** Persistent State

Once the object is connected with the Hibernate Session then the object moves into the Persistent State. So, there are two ways to convert the Transient State to the Persistent State :

1. Using the hibernated session, save the entity object into the database table.
2. Using the hibernated session, load the entity object into the database table.

In this state. each object represents one row in the database table. Therefore, if we make any changes in the data then hibernate will detect these changes and make changes in the database table.



**Converting Transient State to Persistent State**

Following are the methods given for the persistent state:

* session.persist(e);
* session.save(e);
* session.saveOrUpdate(e);
* session.update(e);
* session.merge(e);
* session.lock(e);

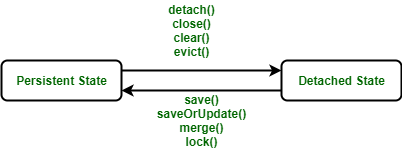
**State 3:** Detached State

For converting an object from Persistent State to Detached State, we either have to close the session or we have to clear its cache. As the session is closed here or the cache is cleared, then any changes made to the data will not affect the database table. Whenever needed, the detached object can be reconnected to a new hibernate session. To reconnect the detached object to a new hibernate session, we will use the following methods as follows:

* merge()
* update()
* load()
* refresh()
* save()
* update()

Following are the methods used for the detached state :

* session.detach(e);
* session.evict(e);
* session.clear();
* session.close();



**State 4:** Removed State

In the hibernate lifecycle it is the last state. In the removed state, when the entity object is deleted from the database then the entity object is known to be in the removed state. It is done by calling the ***delete() operation***. As the entity object is in the removed state, if any change will be done in the data will not affect the database table.

https://media.geeksforgeeks.org/wp-content/uploads/20210626233515/GFGRemovedState.png

Authentication and authorization?

Authentication is **the process of determining whether someone or something is, in fact, who or what it says it is**. Authentication technology provides access control for systems by checking to see if a user's credentials match the credentials in a database of authorized users or in a data authentication server.

**authentication is the process of verifying who someone is, whereas authorization is the process of verifying what specific applications, files, and data a user has access to**

Spring security?

Java Generics?

**Generics** means **parameterized types**. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

The **Object** is the superclass of all other classes, and Object reference can refer to any object. These features lack type safety. Generics add that type of safety feature. We will discuss that type of safety feature in later examples.

**Generic Method:** Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

**Generic Classes:** A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, ​are known as parameterized classes or parameterized types.

// Java program to show working of user defined

// Generic classes

// We use < > to specify Parameter type

class Test<T> {

// An object of type T is declared

T obj;

Test(T obj) { this.obj = obj; } // constructor

public T getObject() { return this.obj; }

}

// Driver class to test above

class Main {

public static void main(String[] args)

{

// instance of Integer type

Test<Integer> iObj = new Test<Integer>(15);

System.out.println(iObj.getObject());

// instance of String type

Test<String> sObj

= new Test<String>("GeeksForGeeks");

System.out.println(sObj.getObject());

}

}

**1. Code Reuse:** We can write a method/class/interface once and use it for any type we want.

**2. Type Safety:** Generics make errors to appear compile time than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students, and if by mistake the programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

**3. Individual Type Casting is not needed:** If we do not use generics, then, in the above example, every time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data, we need not typecast it every time.

**4. Implementing Generic Algorithms:** By using generics, we can implement algorithms that work on different types of objects, and at the same, they are type-safe too.

Hastable Vs HashMap?

* HashMap is non-synchronized. It is not thread-safe and can’t be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
* HashMap allows one null key and multiple null values whereas Hashtable doesn’t allow any null key or value.
* HashMap is generally preferred over HashTable if thread synchronization is not needed.

Exception handling?controllerAdvice?

@ControllerAdvice is a specialization of the @Component annotation which allows to handle exceptions across the whole application in one global handling component. It can be viewed as an interceptor of exceptions thrown by methods annotated with @RequestMapping and similar.

It declares @ExceptionHandler, @InitBinder, or @ModelAttribute methods to be shared across multiple @Controller classes.

ResponseEntityExceptionHandler is a convenient base class for @ControllerAdvice classes that wish to provide centralized exception handling across all @RequestMapping methods through @ExceptionHandler methods. It provides an methods for handling internal Spring MVC exceptions. It returns a ResponseEntity in contrast to DefaultHandlerExceptionResolver which returns a ModelAndView.

public class NoDataFoundException extends RuntimeException {

public NoDataFoundException() {

super("No data found");

}

}

@ControllerAdvice

public class ControllerAdvisor extends ResponseEntityExceptionHandler {

@ExceptionHandler(CityNotFoundException.class)

public ResponseEntity<Object> handleCityNotFoundException(

CityNotFoundException ex, WebRequest request) {

Map<String, Object> body = new LinkedHashMap<>();

body.put("timestamp", LocalDateTime.now());

body.put("message", "City not found");

return new ResponseEntity<>(body, HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(NoDataFoundException.class)

public ResponseEntity<Object> handleNodataFoundException(

NoDataFoundException ex, WebRequest request) {

Map<String, Object> body = new LinkedHashMap<>();

body.put("timestamp", LocalDateTime.now());

body.put("message", "No cities found");

return new ResponseEntity<>(body, HttpStatus.NOT\_FOUND);

}

@Override

protected ResponseEntity<Object> handleMethodArgumentNotValid(

MethodArgumentNotValidException ex, HttpHeaders headers,

HttpStatus status, WebRequest request) {

Map<String, Object> body = new LinkedHashMap<>();

body.put("timestamp", LocalDate.now());

body.put("status", status.value());

List<String> errors = ex.getBindingResult()

.getFieldErrors()

.stream()

.map(x -> x.getDefaultMessage())

.collect(Collectors.toList());

body.put("errors", errors);

return new ResponseEntity<>(body, HttpStatus.BAD\_REQUEST);

}

}

@JoinColomn in JPA?

The annotation javax.persistence.JoinColumn marks a column as a join column for an entity association or an element collection.

@Entity

public class Office {

@OneToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "addressId")

private Address address;

}

The above code example will create a foreign key linking the *Office* entity with the primary key from the *Address* entity. The name of the foreign key column in the *Office* entity is specified by *name* property.

Object oriented Programming?

Object-oriented programming (OOP) is **a computer programming model that organizes software design around data, or objects, rather than functions and logic**.

Static methods in java library?

Functional Programming?

Functional programming (also called FP) is **a way of thinking about software construction by creating pure functions**. It avoid concepts of shared state, mutable data observed in Object Oriented Programming. Functional langauges empazies on expressions and declarations rather than execution of statements.

Junit4 and Junit 5?

## 1. Different Annotations

Most of the annotations in both versions are the same, but a few differ. Here is a quick comparison.

|  |  |  |
| --- | --- | --- |
| **Feature** | **JUnit 4** | **Junit 5** |
| Declare a test method | @Test | @Test |
| Execute before all test methods in the current class | @BeforeClass | [@BeforeAll](https://howtodoinjava.com/junit5/before-all-annotation-example/) |
| Execute after all test methods in the current class | @AfterClass | [@AfterAll](https://howtodoinjava.com/junit5/after-all-annotation-example/) |
| Execute before each test method | @Before | [@BeforeEach](https://howtodoinjava.com/junit5/before-each-annotation-example/) |
| Execute after each test method | @After | [@AfterEach](https://howtodoinjava.com/junit5/after-each-annotation-example/) |
| Disable a test method/class | @Ignore | [@Disabled](https://howtodoinjava.com/junit5/junit-5-disabled-test-example/) |
| Test factory for dynamic tests | NA | @TestFactory |
| Nested tests | NA | @Nested |
| Tagging and filtering | @Category | [@Tag](https://howtodoinjava.com/junit5/junit-5-tag-annotation-example/) |
| Register custom extensions | NA | @ExtendWith |

## 2. More differences between JUnit 5 and JUnit 4

### 2.1. Architecture

JUnit 4 has everything bundled into a single jar file.

JUnit 5 is composed of 3 sub-projects i.e. JUnit Platform, JUnit Jupiter and JUnit Vintage.

* **JUnit Platform**: It defines the TestEngine API for developing new testing frameworks that run on the platform.
* **JUnit Jupiter**: It has all new JUnit annotations and TestEngine implementation to run tests written with these annotations.
* **JUnit Vintage**: To support running JUnit 3 and JUnit 4 written tests on the JUnit 5 platform.

### 2.2. Required JDK Version

Junit 4 requires Java 5 or higher.

Junit 5 requires Java 8 or higher.

### 2.3. Assertions

In Junit 4, [org.junit.Assert](https://junit.org/junit4/javadoc/4.12/org/junit/Assert.html) has all assert methods to validate expected and resulting outcomes.  
They accept extra parameters for error messages as the FIRST argument in the method signature. e.g.

public static void assertEquals(long expected, long actual)

public static void assertEquals(String message, long expected, long actual)

In JUnit 5, [org.junit.jupiter.Assertions](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html) contain most of assert() methods including additional assertThrows() and assertAll() methods.   
JUnit 5 assertions methods also have overloaded methods to support parsing error messages to be printed in case test fails e.g.

public static void assertEquals(long expected, long actual)

public static void assertEquals(long expected, long actual, String message)

public static void assertEquals(long expected, long actual, Supplier messageSupplier)

### 2.4. Assumptions

In Junit 4, [org.junit.Assume](https://junit.org/junit4/javadoc/4.12/org/junit/Assume.html) contains methods for stating assumptions about the conditions in which a test is meaningful. It has the following five methods:

1. assumeFalse()
2. assumeNoException()
3. assumeNotNull()
4. assumeThat()
5. assumeTrue()

In Junit 5, [org.junit.jupiter.api.Assumptions](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assumptions.html) contain methods for stating assumptions about the conditions in which a test is meaningful. It has the following three methods:

1. assumeFalse()
2. assumingThat​()
3. assumeTrue()

### 2.5. Tagging and Filtering

In Junit 4, @category annotation is used.

In Junit 5, @tag annotation is used.

### 2.6. Test Suites

In Junit 4, @RunWith and @Suite annotation. e.g.

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({

ExceptionTest.class,

TimeoutTest.class

})

public class JUnit4Example

{

}

In Junit 5, [@Suite](https://howtodoinjava.com/junit5/junit5-test-suites-examples/), @SelectPackages and @SelectClasses e.g.

import org.junit.platform.runner.JUnitPlatform;

import org.junit.platform.suite.api.SelectPackages;

import org.junit.runner.RunWith;

@Suite

@SelectPackages("com.howtodoinjava.junit5.examples")

public class JUnit5Example

{

}

### 2.7. Non-public Test Methods are Allowed

* JUnit 5 test classes and test methods are not required to be public. We can now make them [package protected](https://howtodoinjava.com/java/oops/java-access-modifiers/).  
  JUnit internally uses reflection to find test classes and test methods. Reflection can discover them even if they have limited visibility so there is no need for them to be public.
* JUnit test classes also can have non-public [constructors](https://howtodoinjava.com/java/oops/java-constructors/). They can even have arguments. It means having a public no-args constructor is not mandatory in JUnit 5.

class AppTest {

private AppTest(TestInfo testInfo) {

System.out.println("Working on test " + testInfo.getDisplayName());

}

@Test

void test(){

assertTrue(true);

}

}

### 2.8. 3rd Party Integration

In Junit 4, there is no integration support for 3rd party plugins and IDEs. They have to rely on [reflection](https://howtodoinjava.com/java/reflection/real-usage-examples-of-reflection-in-java/).

@RunWith(SpringRunner.class)

It provides a bridge between Spring Boot test features and JUnit. Whenever we are using any Spring Boot testing features in our JUnit tests, this annotation will be required.

## What is the API Gateway pattern?

When you design and build large or complex microservice-based applications with multiple client apps, a good approach to consider can be an [API Gateway](https://microservices.io/patterns/apigateway.html). This pattern is a service that provides a single-entry point for certain groups of microservices. It's similar to the [Facade pattern](https://en.wikipedia.org/wiki/Facade_pattern) from object-oriented design, but in this case, it's part of a distributed system. The API Gateway pattern is also sometimes known as the "backend for frontend" ([BFF](https://samnewman.io/patterns/architectural/bff/)) because you build it while thinking about the needs of the client app.

Therefore, the API gateway sits between the client apps and the microservices. It acts as a reverse proxy, routing requests from clients to services. It can also provide other cross-cutting features such as authentication, SSL termination, and cache.

## Why consider API Gateways instead of direct client-to-microservice communication

In a microservices architecture, the client apps usually need to consume functionality from more than one microservice. If that consumption is performed directly, the client needs to handle multiple calls to microservice endpoints. What happens when the application evolves and new microservices are introduced or existing microservices are updated? If your application has many microservices, handling so many endpoints from the client apps can be a nightmare. Since the client app would be coupled to those internal endpoints, evolving the microservices in the future can cause high impact for the client apps.

Therefore, having an intermediate level or tier of indirection (Gateway) can be convenient for microservice-based applications. If you don't have API Gateways, the client apps must send requests directly to the microservices and that raises problems, such as the following issues:

* **Coupling**: Without the API Gateway pattern, the client apps are coupled to the internal microservices. The client apps need to know how the multiple areas of the application are decomposed in microservices. When evolving and refactoring the internal microservices, those actions impact maintenance because they cause breaking changes to the client apps due to the direct reference to the internal microservices from the client apps. Client apps need to be updated frequently, making the solution harder to evolve.
* **Too many round trips**: A single page/screen in the client app might require several calls to multiple services. That approach can result in multiple network round trips between the client and the server, adding significant latency. Aggregation handled in an intermediate level could improve the performance and user experience for the client app.
* **Security issues**: Without a gateway, all the microservices must be exposed to the "external world", making the attack surface larger than if you hide internal microservices that aren't directly used by the client apps. The smaller the attack surface is, the more secure your application can be.
* **Cross-cutting concerns**: Each publicly published microservice must handle concerns such as authorization and SSL. In many situations, those concerns could be handled in a single tier so the internal microservices are simplified.

Spring-cloud-sleuth

One of the problems developers encounter as their microservice apps grow is tracing requests that propagate from one microservice to the next. It can quite daunting to try and figure out how a requests travels through the app, especially when you may not have any insight into the implementation of the microservice you are calling.

[Spring Cloud Sleuth](https://cloud.spring.io/spring-cloud-sleuth/) is meant to help with this exact problem. It introduces unique IDs to your logging which are consistent between microservice calls which makes it possible to find how a single request travels from one microservice to the next.

Spring Cloud Sleuth adds two types of IDs to your logging, one called a trace ID and the other called a span ID. The span ID represents a basic unit of work, for example sending an HTTP request. The trace ID contains a set of span IDs, forming a tree-like structure. The trace ID will remain the same as one microservice calls the next. Let's take a look at a simple example which uses Spring Cloud Sleuth to trace a request.

Start out by going to [start.spring.io](http://start.spring.io) and create a new Spring Boot app that has a dependency on Sleuth (spring-cloud-starter-slueth). Generate the project to download the code. It is good practice to give your application a name and also necessary to have meaningful tracing from Sleuth. Create a file called bootstrap.yml in src/main/resources. Within that file add the property spring.application.name and set it to whatever you would like to call your application. The name you give your application will show up as part of the tracing produced by Sleuth.

## What is Eureka Server?

Eureka Server is service discovery for your microservices, where all client applications can register by themselves and other microservices look up the Eureka Server to get independent microservices to get the job complete.

Eureka Server is also known as Discovery Server and it contains all the information about client microservices running on which IP address and port.

### **Spring Boot Starter Application**

Now you need to open EurekaServerApplication.javaand add the annotation @EnableEurekaServer   on the top of the class as shown below.

### **Application Properties File**

You need to add the below list of properties in application.properties located at src/main/resources on your application.

1

spring.application.name=eureka-server

2

server.port=8761

3

eureka.client.register-with-eureka=false

4

eureka.client.fetch-registry=false

* **spring.application.name** is a unique name for your application.
* **server.port** in which your application will be bound and wewill use default port 8761 for eureka server.
* **eureka.client.fetch-registry** doesn't register itself in eureka server.
* **eureka.client.register-with-eureka** is determines if service register itself as a client in eureka server

### **Java Spring Server Application**

Now you need to open the SpringBootApplication EurekaClientApplication.java and add the annotation @EnableDiscoveryClient   on the top of the class as shown below.

### **Application Properties File**

You need to add the below list of properties in application.properties located at src/main/resources.

1

spring.application.name=eureka-client-service

2

server.port=8081

3

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

**eureka.client.service-url.defaultZone** determines the address where the Eureka Server is running so the client application can register itself in Eureka Server.

Circuit-Breaker

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

A circuit breaker acts as a proxy for operations that might fail. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.

The proxy can be implemented as a state machine with the following states that mimic the functionality of an electrical circuit breaker:

* **Closed**: The request from the application is routed to the operation. The proxy maintains a count of the number of recent failures, and if the call to the operation is unsuccessful the proxy increments this count. If the number of recent failures exceeds a specified threshold within a given time period, the proxy is placed into the **Open** state. At this point the proxy starts a timeout timer, and when this timer expires the proxy is placed into the **Half-Open** state.

The purpose of the timeout timer is to give the system time to fix the problem that caused the failure before allowing the application to try to perform the operation again.

* **Open**: The request from the application fails immediately and an exception is returned to the application.
* **Half-Open**: A limited number of requests from the application are allowed to pass through and invoke the operation. If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the **Closed** state (the failure counter is reset). If any request fails, the circuit breaker assumes that the fault is still present so it reverts back to the **Open** state and restarts the timeout timer to give the system a further period of time to recover from the failure.

Callable and Runnable?

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. **A Runnable, however, does not return a result and cannot throw a checked exception.**

| **Runnable interface** | **Callable interface** |
| --- | --- |
| It is a part of [*java.lang*](https://www.geeksforgeeks.org/object-class-in-java/) package since Java 1.0 | It is a part of the [*java.util.concurrent*](https://www.geeksforgeeks.org/java-util-concurrent-package/)package since Java 1.5. |
| It cannot return the return of computation. | It can return the result of the parallel processing of a task. |
| It cannot throw a checked Exception. | It can throw a checked Exception. |
| In a runnable interface, one needs to override the run() method in Java. | In order to use Callable, you need to override the call() |

## BeanFactory vs ApplicationContext in Spring Framework

## Overview

The Spring Framework comes with two IOC containers – [*BeanFactory*](https://www.baeldung.com/spring-beanfactory) and [*ApplicationContext*](https://www.baeldung.com/spring-classpathxmlapplicationcontext). The BeanFactory is the most basic version of IOC containers, and the ApplicationContext extends the features of BeanFactory.

In this quick tutorial, we'll understand the significant differences between these two IOC containers with practical examples.

## 1. Lazy Loading vs. Eager Loading

**BeanFactory loads beans on-demand, while ApplicationContext loads all beans at startup**. Thus, BeanFactory is lightweight as compared to ApplicationContext.

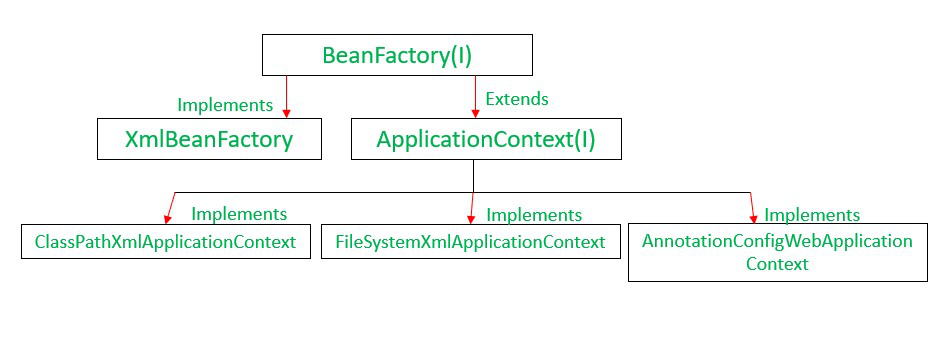
## 2.Enterprise Application Features

ApplicationContext enhances BeanFactory in a more framework-oriented style and provides several features that are suitable for enterprise applications.

For instance, it **provides**[**messaging (i18n or internationalization)**](https://www.baeldung.com/spring-classpathxmlapplicationcontext#2-internationalization-with-messagesource) functionality, [**event publication**](https://www.baeldung.com/spring-events) functionality, **annotation-based dependency injection**, and **easy integration with Spring AOP features**.

## 4. Automatic Registration of BeanFactoryPostProcessor and BeanPostProcessor

The **ApplicationContext automatically registers BeanFactoryPostProcessor and BeanPostProcessor** at startup. On the other hand, the BeanFactory does not register these interfaces automatically.



**Why Generics?**

Generics is a concept in Java where you can enable a class, interface and, method, accept all (reference) types as parameters. In other words it is the concept which enables the users to choose the reference type that a method, constructor of a class accepts, dynamically.

By defining a class generic you are making it type-safe i.e. it can act up on any datatype.

## Creating generic types

To create a class of generic type by using generic parameter T or, GT as −

class Student <T>{

   T obj;

}

Where T (generic parameter) represents the datatype of the object you can pass to the constructor of this class. This will be determined at the compilation time.

While instantiating the class you need to/can choose the type of the generic parameter as −

Student<Float> obj = new Student<Float>();

while instantiating the Student class you can pass desired type of object as a parameter as −

class Student<T>{

   T age;

   Student(T age){

      this.age = age;

}

   public void display() {

      System.out.println("Value of age: "+this.age);

   }

}

public class GenericsExample {

   public static void main(String args[]) {

      Student<Float> std1 = new Student<Float>(25.5f);

      std1.display();

      Student<String> std2 = new Student<String>("25");

      std2.display();

      Student<Integer> std3 = new Student<Integer>(25);

      std3.display();

   }

}

* **Type check at compile time −**usually, when you use types (regular objects), when you pass an incorrect object as a parameter, it will prompt an error at the run time.
* Whereas, when you use generics the error will be at the compile time which is easy to solve.
* **Code reuse −** You can write a method or, Class or, interface using generic type once and you can use this code multiple times with various parameters.
* For certain types, with formal types, you need to cast the object and use. Using generics (in most cases) you can directly pass the object of required type without relying on casting.
* Using generic types you can implement various generic algorithms.

Memory Management in Java

Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

The biggest advantage of garbage collection is that it removes the burden of manual memory allocation/deallocation from us so that we can focus on solving the problem at hand.

The heap is a large bulk of memory intended for allocation of objects. When you create an object with the new keyword, it gets allocated on the heap. However, the reference to this object lives on the stack.

The stack is a part of memory that contains information about nested method calls down to the current position in the program. It also contains all local variables and references to objects on the heap defined in currently executing methods.

This structure allows the runtime to return from the method knowing the address whence it was called, and also clear all local variables after exiting the method. Every thread has its own stack.

An object becomes eligible for Garbage collection or GC if it is not reachable from any live threads or by any static references.

The most straightforward case of an object becoming eligible for garbage collection is if all its references are null. Cyclic dependencies without any live external reference are also eligible for GC. So if object A references object B and object B references Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.

Another obvious case is when a parent object is set to null. When a kitchen object internally references a fridge object and a sink object, and the kitchen object is set to null, both fridge and sink will become eligible for garbage collection alongside their parent, kitchen.

**Types of GC**

There are **four** types of the garbage collector in [Java](https://www.javatpoint.com/java-tutorial) that can be used according to the requirement:

* Serial Garbage Collector
* Parallel Garbage Collector
* Concurrent Mark Sweep (CMS) Garbage Collector
* Garbage First (G1) Garbage Collector

### **Serial Garbage Collector**

Serial Garbage collector is well-matched for **single-threaded environments**. It uses the **only** thread for garbage collection. It works by holding all the threads of an application. It means that threads of the application **freeze** by the serial garbage collector during the garbage collection process and the process is known as **stop the world event**. Avoid the use of serial GC in the server environment. We can use it for simple programs. If you want to use the serial garbage collector, execute the **-XX:+UseSerialGC** JVM argument to activate it.

### **Parallel Garbage Collector**

Parallel Garbage Collector is the **default GC** used by the JVM. The working of the parallel garbage collector is the same as the serial garbage collector. The only difference between serial and parallel garbage collector is that serial garbage collector uses a **single** thread for garbage collection process while the parallel garbage collector uses **multiple** threads for the garbage collection. Parallel GC can use multiple CPUs to speed up the application throughput. So, it is also known as **throughput collector**. It is used if we want to execute a long process (like batch processing) and where long pauses are acceptable. If you want to use the parallel garbage collector, execute the **-XX:+UseParallelGC** JVM argument to activate it.

### **Concurrent Mark and Sweep (CMS) Garbage Collector**

CMS uses multiple threads that scan the heap and during the scanning, it marks the instances for eviction, after scanning, it sweeps the marked instances. It does not freeze the application's threads during the garbage collection. GC threads concurrently execute with the application's threads. For this reason, it uses more CPU in comparison to other GC. It is also known as the **concurrent low pause collector**. It also freezes all the threads of the application only if it satisfies the following two scenarios:

* while marking the referenced objects in the tenured generation region.
* if any change is made to the heap memory in parallel during the garbage collection process.

We can use multiple CPUs for better application throughput. We should use a CMS garbage collector if we have more CPUs for use. Hence, it has an advantage over the parallel garbage collector. If you want to use a CMS garbage collector, execute the **-XX:+USeParNewGC**JVM argument to activate it. We can also set the number of GC threads by using the **-XX:ParallelCMSThreads=<n>** JVM argument.

### **Garbage First (G1) Garbage Collector**

The G1 garbage collector is used if we have a large (more than 4GB) memory (heap space). It divides the heap into equal-sized (usually 1MB to 32MB) chunks, prioritizes them, and then performs the parallel garbage collection on that chunks based on the priority.

The Eden, survivors, and old areas use this equal-sized region for the memory allocation of the objects. Apart from these memory regions, there are two more types of regions presented in the G1 GC:

* **Humongous:** It is used if the object sized is large.
* **Available:** It represents the unoccupied space.

## Stop the World Event

It is a situation when the garbage collector performs the garbage collection (GC) and stops all the application's threads until the GC process is not completed. The process is known as **Stop the World (STW)**events.

## Improvement in Garbage Collector Since Java 8

In Java 8, the G1 garbage collector has been updated. The updated GC provides the **-XX:+UseStringDeduplication** JVM argument that optimizes the heap memory. It removes the duplicate String values to a single char[] array.

Memory changes from Java8

Metaspace is a new memory space – starting from the Java 8 version; **it has replaced the older PermGen memory space**. The most significant difference is how it handles memory allocation.

Specifically, **this native memory region grows automatically by default**.

We also have new flags to tune the memory:

* MetaspaceSize and MaxMetaspaceSize – we can set the Metaspace upper bounds.
* MinMetaspaceFreeRatio – is the minimum percentage of class metadata capacity free after [**garbage collection**](https://www.baeldung.com/jvm-garbage-collectors)
* MaxMetaspaceFreeRatio – is the maximum percentage of class metadata capacity free after a garbage collection to avoid a reduction in the amount of space

Additionally, the garbage collection process also gains some benefits from this change. The garbage collector now automatically triggers the cleaning of the dead classes once the class metadata usage reaches its maximum metaspace size.

Therefore,**with this improvement, JVM reduces the chance to get the OutOfMemory error**.

**What is PermGen ?**

Short form for Permanent Generation, PermGen is the memory area in Heap that is used by the JVM to **store class and method objects**. If your application loads lots of classes, PermGen utilization will be high. PermGen also **holds ‘interned’ Strings**

The size of the PermGen space is configured by the Java command line option**-XX:MaxPermSize**

Typically 256 MB should be more than enough of PermGen space for most of the applications

However, It is not unusal to see the error “**java.lang.OutOfMemoryError: PermGen space“** if you are loading unusual number oThe key difference between PermGen and Metaspace is this: while PermGen is part of Java Heap (Maximum size configured by -Xmx option), **Metaspace is NOT part of Heap.** Rather Metaspace is part of **Native Memory (process memory)** which is only limited by the Host Operating System.f classes.

Gone are the days of OutOfMemory Errors due to PermGen space. **With Java 8, there is NO PermGen**. That’s right. So no more OutOfMemory Errors due to PermGen

**So, what is the significance of this change?**

While you will NOT run out of PermGen space anymore (since there is NO PermGen), you may consume excessive Native memory making the total process size large. The issue is, if your application loads lots of classes (and/or interned strings), **you may actually bring down the Entire Server** (not just your application). Why ? Because the native memory is only limited by the Operating System. This means you can literally take up all the memory on the Server. Not good.

It is critical that you add the new option **-XX:MaxMetaspaceSize**  which sets the Maximum Metaspace size for your application.

@Bean and @Autowired

@Bean and @Autowired do two very different things. The other answers here explain in a little more detail, but at a simpler level:

* @Bean tells Spring 'here is an instance of this class, please keep hold of it and give it back to me when I ask'.
* @Autowired says 'please give me an instance of this class, for example, one that I created with an @Bean annotation earlier'.

@Bean is just for the metadata definition to create the bean(equivalent to tag). @Autowired is to inject the dependancy into a bean(equivalent to ref XML tag/attribute).

## @Component vs @Bean [enter image description here](https://i.stack.imgur.com/zJpux.png)

1. @Component **auto detects** and configures the beans using classpath scanning whereas @Bean **explicitly declares** a single bean, rather than letting Spring do it automatically.
2. @Component **does not decouple** the declaration of the bean from the class definition where as @Bean **decouples** the declaration of the bean from the class definition.
3. @Component is a **class level annotation** whereas @Bean is a **method level annotation** and name of the method serves as the bean name.
4. @Component **need not to be used with the @Configuration** annotation where as @Bean annotation has to be **used within the class which is annotated with @Configuration**.
5. We **cannot create a bean** of a class using @Component, if the class is outside spring container whereas we **can create a bean** of a class using @Bean even if the class is present **outside the spring container**.
6. @Component has **different specializations** like @Controller, @Repository and @Service whereas @Bean has **no specializations**.

**Spring MVC flow**

low of an HTTP request in Java application created using the Spring MVC framework:  
  
1) The client sends an HTTP request to a specific URL  
  
2) DispatcherServlet of Spring MVC receives the request  
  
2) It passes the request to a specific controller depending on the URL requested using @Controller and @RequestMapping annotations.  
  
3) Spring MVC Controller then returns a logical view name and model to DispatcherServlet.  
  
4) DispatcherServlet consults view resolvers until actual View is determined to render the output  
  
5) DispatcherServlet contacts the chosen view (like Thymeleaf, Freemarker, JSP) with model data and it renders the output depending on the model data  
  
6) The rendered output is returned to the client as a response  
  
The flow of the [RESTful Web Service](http://javarevisited.blogspot.sg/2017/02/how-to-consume-json-from-restful-web-services-Spring-RESTTemplate-Example.html#axzz4jWEcwHFD) request is also not very different from this. It follows the same path but in the case of REST, the Controller methods are annotated with @ResponseBody which means it doesn't return a logical view name to DispatcherServlet, instead it write the output directly to the HTTP response body.

**Can we Override static methods in java?**   
We can declare static methods with the same signature in the subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘No’.   
If a derived class defines a static method with the same signature as a static method in the base class, the method in the derived class is hidden by the method in the base class.

**JVM Memory Management**

When an object is created, it is allocated on the **Eden(1)** space. Because the Eden space is not that big, it gets full quite fast. The garbage collector runs on the Eden space and marks objects as alive.

Once an object survives a garbage collecting process, it gets moved into a so-called survivor space **S0(2)**. The second time the garbage collector runs on the Eden space, it moves all surviving objects into the **S1(3)** space. Also, everything that is currently on **S0(2)** is moved into the **S1(3)** space.

If an object survives for X rounds of garbage collection (X depends on the JVM implementation, in my case it’s 8), it is most likely that it will survive forever, and it gets moved into the **Old(4)** space.

Taking everything said so far, if you look at the **garbage collector graph(6)**, each time it has run, you can see that the objects switch to the survivor space and that the Eden space gained space. And so on and so forth. The old generation can be also garbage collected, but since it is a bigger part of the memory compared to Eden space, it does not happen that often. The **Metaspace(5)** is used to store the metadata about your loaded classes in the JVM.

The presented picture is actually a Java 8 application. Prior to Java 8, the structure of the memory was a bit different. The metaspace is called actually the PermGen. space. For example, in Java 6, this space also stored the memory for the string pool. Therefore, if you have too many strings in your Java 6 application, it might crash.

**JVM Architecture**

**Class Loaders in Java**

The **Java ClassLoader** is a part of the [**Java Runtime Environment**](https://www.geeksforgeeks.org/differences-jdk-jre-jvm/) that dynamically loads Java classes into the [**Java Virtual Machine**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). The Java run time system does not need to know about files and file systems because of classloaders.

[Java classes](https://www.geeksforgeeks.org/classes-objects-java/) aren’t loaded into memory all at once, but when required by an application. At this point, the **Java ClassLoader** is called by the **JRE** and these ClassLoaders load classes into memory dynamically.

**Types of ClassLoaders in Java**

Not all classes are loaded by a single ClassLoader. Depending on the type of class and the path of class, the ClassLoader that loads that particular class is decided. To know the ClassLoader that loads a class the [*getClassLoader()*](https://www.geeksforgeeks.org/java-lang-class-class-java-set-1/) method is used. All classes are loaded based on their names and if any of these classes are not found then it returns a [**NoClassDefFoundError**](https://www.geeksforgeeks.org/classnotfoundexception-vs-noclassdeffounderror-java/) or [**ClassNotFoundException**](https://www.geeksforgeeks.org/classnotfoundexception-vs-noclassdeffounderror-java/).

A Java Classloader is of **three types**:

1. **BootStrap ClassLoader:** A Bootstrap Classloader is a Machine code which kickstarts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location ***rt.jar***. Bootstrap ClassLoader doesn’t have any parent ClassLoaders. It is also called as the **Primodial ClassLoader**.
2. **Extension ClassLoader:** The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from ***jre/lib/ext*** directory or any other directory pointed by the system property ***java.ext.dirs***.
3. **System ClassLoader:** An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable ***CLASSPATH, -classpath or -cp command line option***. The Application ClassLoader is a child class of Extension ClassLoader.

**Note**: The ClassLoader Delegation Hierarchy Model always functions in the order Application ClassLoader->Extension ClassLoader->Bootstrap ClassLoader. The Bootstrap ClassLoader is always given the higher priority, next is Extension ClassLoader and then Application ClassLoader.

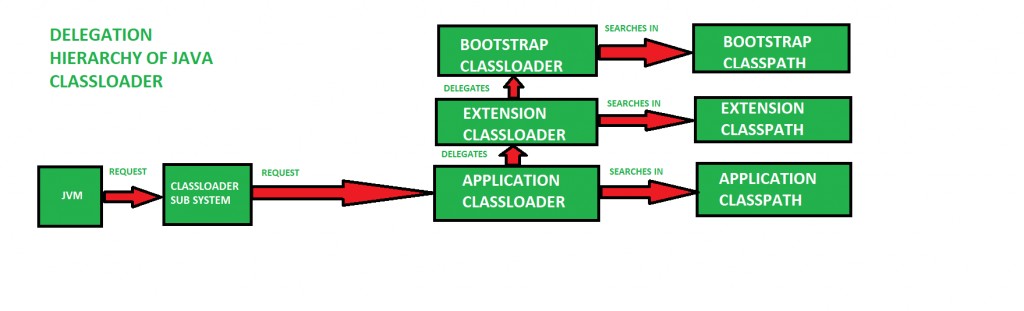
**Principles of functionality of a Java ClassLoader**

Principles of functionality are the **set of rules** or features on which a Java ClassLoader works. There are **three principles of functionality**, they are:

1. **Delegation Model**: The Java Virtual Machine and the Java ClassLoader use an algorithm called the **Delegation Hierarchy Algorithm** to Load the classes into the Java file.

The ClassLoader works based on a set of operations given by the delegation model. They are:

* + ClassLoader always follows the **Delegation Hierarchy Principle**.
  + Whenever JVM comes across a class, it checks whether that class is already loaded or not.
  + If the Class is already loaded in the method area then the JVM proceeds with execution.
  + If the class is not present in the method area then the JVM asks the Java ClassLoader Sub-System to load that particular class, then ClassLoader sub-system hands over the control to **Application ClassLoader**.
  + Application ClassLoader then delegates the request to Extension ClassLoader and the **Extension ClassLoader** in turn delegates the request to **Bootstrap ClassLoader**.
  + Bootstrap ClassLoader will search in the Bootstrap classpath(JDK/JRE/LIB). If the class is available then it is loaded, if not the request is delegated to Extension ClassLoader.
  + Extension ClassLoader searches for the class in the Extension Classpath(JDK/JRE/LIB/EXT). If the class is available then it is loaded, if not the request is delegated to the Application ClassLoader.
  + Application ClassLoader searches for the class in the Application Classpath. If the class is available then it is loaded, if not then a **ClassNotFoundException** exception is generated.

[](https://media.geeksforgeeks.org/wp-content/uploads/20190417144207/java_classloader.png)

1. **Visibility Principle**: The **Visibility Principle** states that a class loaded by a parent ClassLoader is visible to the child ClassLoaders but a class loaded by a child ClassLoader is not visible to the parent ClassLoaders. Suppose a class GEEKS.class has been loaded by the Extension ClassLoader, then that class is only visible to the Extension ClassLoader and Application ClassLoader but not to the Bootstrap ClassLoader. If that class is again tried to load using Bootstrap ClassLoader it gives an exception ***java.lang.ClassNotFoundException***.
2. **Uniqueness Property**: The **Uniquesness Property** ensures that the classes are unique and there is no repetition of classes. This also ensures that the classes loaded by parent classloaders are not loaded by the child classloaders. If the parent class loader isn’t able to find the class, only then the current instance would attempt to do so itself.

**Methods of Java.lang.ClassLoader**

After the JVM requests for the class, a few steps are to be followed in order to load a class. The Classes are loaded as per the delegation model but there are a few important Methods or Functions that play a vital role in loading a Class.

1. **loadClass(String name, boolean resolve)**: This method is used to load the classes which are referenced by the JVM. It takes the name of the class as a parameter. This is of type loadClass(String, boolean).
2. **defineClass()**: The defineClass() method is a *final* method and cannot be overriden. This method is used to define a array of bytes as an instance of class. If the class is invalid then it throws **ClassFormatError**.
3. **findClass(String name)**: This method is used to find a specified class. This method only finds but doesn’t load the class.
4. **findLoadedClass(String name)**: This method is used to verify whether the Class referenced by the JVM was previously loaded or not.
5. **Class.forName(String name, boolean initialize, ClassLoader loader)**: This method is used to load the class as well as initialize the class. This method also gives the option to choose any one of the ClassLoaders. If the ClassLoader parameter is NULL then Bootstrap ClassLoader is used.

**Example:** The following code is executed before a class is loaded:

|  |
| --- |
| protected synchronized Class<?>  loadClass(String name, boolean resolve)      throws ClassNotFoundException  {      Class c = findLoadedClass(name);      try {          if (c == NULL) {              if (parent != NULL) {                  c = parent.loadClass(name, false);              }              else {                  c = findBootstrapClass0(name);              }          }          catch (ClassNotFoundException e)          {              System.out.println(e);          }      }  }  How classes are loaded in java? |

When you compile this piece of code and run it, the JVM correctly determines the entry point into the program and starts running the main method of class A. However, the JVM doesn’t load all imported classes or even referred-to classes eagerly—that is, right away. In particular, this means that only when the JVM encounters the bytecode instructions for the new B() statement will it try to locate and load class B.

Besides calling a constructor of a class, there are other ways to initiate the process of loading a class, such as accessing a static member of the class or accessing it through the Reflection API.

In order to actually load a class, the JVM uses Classloader objects. Every already loaded class contains a reference to its class loader, and that class loader is used to load all the classes referenced from that class. In the preceding example, this means that loading class B can be approximately translated into the following Java statement: A.class.getClassLoader().loadClass("B").

Here comes a paradox: every class loader is itself an object of the java.lang.Classloader type that developers can use to locate and load the classes by name. If you’re confused by this chicken-and-egg problem and wonder how the first class loader that loads all the JDK classes (for example, java.lang String) is created, you’re thinking along the right lines.

Indeed, the primordial class loader, called the *bootstrap class loader*, comes from the core of the JVM and is written in native platform-dependent code. It loads the classes necessary for the JVM itself, such as those of the java.lang package, classes for Java primitives, and so forth. Application classes are loaded using the regular, user-defined class loaders written in Java—so, if needed, the developer can influence the processing of these loaders.

## The Class-Loader Hierarchy

The class loaders in the JVM are organized into a tree hierarchy, in which every class loader has a parent. Prior to locating and loading a class, a good practice for a class loader is to check whether the class’s parent can load—or already has loaded—the required class.

This helps avoid doing double work and loading classes repeatedly. As a rule, the classes of the parent class loader are visible to the children but are not visible otherwise. This structure, which is based on delegation and visibility of the classes, allows for separation of the responsibilities of the class loaders in the hierarchy and makes the class loaders responsible for loading classes from a specific location only.

Let’s look at this hierarchy of class loaders in a Java application and explore what classes they typically load. At the root of the hierarchy, Java is the bootstrap class loader. It loads the system classes required to run the JVM itself. You can expect all the classes that were provided with the JDK distribution to be loaded by this class loader. (A developer can expand the set of classes that the bootstrap class loader will be able to load by using the -Xbootclasspath JVM option.)

Note that even though the library might be put on the boot classpath, it won’t be automatically loaded and initialized. Classes are loaded into the JVM only on demand, so even though classes might be available for the bootstrap class loader, the application needs to access them to trigger their actual loading. (A curious aspect of this loading process is that you can override JDK classes if your JAR file is prepended to the boot classpath. While this is almost always a poor idea, it does open a door to potentially more-powerful tools.)

A sort of child of the bootstrap class loader is the extension class loader, which loads the classes from the extension directories (explained in a moment). These classes may be used to specify machine-specific configuration such as locales, security providers, and such. The locations of the extension directories are specified via the java.ext.dirs system property, which on my machine is set to the following:

/Users/shelajev/Library/Java/Extensions:/Library/

Java/JavaVirtualMachines/jdk1.8.0\_40.jdk/Contents/

Home/jre/lib/ext:/Library/Java/Extensions:/Network/

Library/Java/Extensions:/System/Library/Java/

Extensions:/usr/lib/java

By changing the value of this property, you can change which additional libraries are loaded into the JVM process.

Next comes the system class loader, which loads the application classes and the classes available on the classpath. Users can specify the classpath using the -cp property.

Both the extension class loader and the system class loader are of the URLClassloader type and behave in the same way: delegating to the parent first, and only then finding and resolving the required classes themselves, if need dictates.

The class-loader hierarchy of web applications is a bit more complicated. Because multiple applications can be deployed simultaneously to an application server, they need to be able to distinguish their classes from each other. So, every web application uses its own class loader, which is responsible for loading its libraries.

Such isolation ensures that different web applications deployed to a single server can have different versions of the same library without conflicts. So the application server automatically provides every web application with its own class loader, which is responsible for loading the application’s libraries. This arrangement works because the web application class loader will try to locate the classes packaged in the application’s WAR file first, rather than first delegating the search to the parent class loader.

**Finding the Right Class**

In general, if multiple classes with the same fully qualified name are available to the JVM, the conflict resolution strategy is simple and straightforward: the first appropriate class wins. The URLClassloader, which most of the class loaders extend from, will traverse the directories in the order they are given on the classpath and load the first class it finds that has requested the class name.

The same goes for JAR files that share the same name. The JAR files will be scanned in the order in which they appear in the classpath, not according to their names. If the first JAR file contains an entry for the required class, the class will be loaded. If not, the classpath scan will continue and reach the second JAR file.

Naturally, if the class isn’t found anywhere on the classpath, the ClassNotFound exception will be thrown.

Usually, relying on the order of directories in the classpath is a fragile practice, so instead the developer can add the classes to -Xbootclasspath to ensure that they will be loaded first. There’s nothing in particular wrong with this approach, but maintaining a project that relies on a polluted boot classpath requires work. Intuition about where the classes are loaded from will be broken, and everyone will be confused.

A better practice is to resolve the confusion at its root and figure out why there are multiple classes with the same name on the classpath. Maybe upgrading some dependency version, cleaning the caches, or running a clean build will be enough to get rid of the duplicates.

**Resolution, Linking, and Verification**

After a class is located and its initial in-memory representation created in the JVM process, it is verified, prepared, resolved, and initialized.

* **Verification** makes sure that the class is not corrupted and is structurally correct: its runtime constant pool is valid, the types of variables are correct, and the variables are initialized prior to being accessed. Verification can be turned off by supplying the -noverify option. If the JVM process does not run potentially malicious code, strict verification might not be required. Turning off the verification can speed up the startup of the JVM. Another benefit is that some classes, especially those generated on the fly by various tools, can be valid and safe for the JVM but unable to pass the strict verification process. In order to use such tools, the developer should disable this verification, which is often acceptable to do in a development environment.
* **Preparation** of a class involves initializing its static fields to the default values for their respective types. (After preparation, fields of type int contain 0, references are null, and so forth.)
* **Resolution** of a class means checking that the symbolic references in the runtime constant pool actually point to valid classes of the required types. The resolution of a symbolic reference triggers loading of the referenced class. According to the JVM specification, this resolution process can be performed lazily, so it is deferred until the class is used.
* **Initialization** expects a prepared and verified class. It runs the class’s initializer. During initialization, the static fields are initialized to whatever values are specified in the code. The static initializer method that combines the code from all the static initialization blocks is also run. The initialization process should be run only once for every loaded class, so it is synchronized, especially because the initialization of the class can trigger the initialization of other classes and should be performed with care to avoid deadlocks.

### Runtime Data Area

The Runtime Data Area is divided into five major components:

1. **Method Area** – All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.
3. **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:
   1. **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here.
   2. **Operand stack** – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
   3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

### Execution Engine

The bytecode, which is assigned to the **Runtime Data Area,** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
   1. **Intermediate Code Generator** – Produces intermediate code
   2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
   3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
   4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

**Java Native Interface (JNI)**: JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

**Native Method Libraries**: This is a collection of the Native Libraries, which is required for the Execution Engine.

Singleton class

### **How to Design/Create a Singleton Class in Java?**

To create a singleton class, we must follow the steps, given below:

**1.**Ensure that only one instance of the class exists.

**2.**Provide global access to that instance by

* Declaring all constructors of the class to be private.
* Providing a static method that returns a reference to the instance. The lazy initialization concept is used to write the static methods.
* The instance is stored as a private static variable.

// Java program implementing Singleton class

// with using getInstance() method

// Class 1

// Helper class

class Singleton {

// Static variable reference of single\_instance

// of type Singleton

private static Singleton single\_instance = null;

// Declaring a variable of type String

public String s;

// Constructor

// Here we will be creating private constructor

// restricted to this class itself

private Singleton()

{

s = "Hello I am a string part of Singleton class";

}

// Static method

// Static method to create instance of Singleton class

public static Singleton getInstance()

{

if (single\_instance == null)

single\_instance = new Singleton();

return single\_instance;

}

}

// Class 2

// Main class

class GFG {

// Main driver method

public static void main(String args[])

{

// Instantiating Singleton class with variable x

Singleton x = Singleton.getInstance();

// Instantiating Singleton class with variable y

Singleton y = Singleton.getInstance();

// Instantiating Singleton class with variable z

Singleton z = Singleton.getInstance();

// Printing the hash code for above variable as

// declared

System.out.println("Hashcode of x is "

+ x.hashCode());

System.out.println("Hashcode of y is "

+ y.hashCode());

System.out.println("Hashcode of z is "

+ z.hashCode());

// Condition check

if (x == y && y == z) {

// Print statement

System.out.println(

"Three objects point to the same memory location on the heap i.e, to the same object");

}

else {

// Print statement

System.out.println(

"Three objects DO NOT point to the same memory location on the heap");

}

}

}

**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.

**What if parent class has already overriden clone() method, and it is not throwing CloneNotSupportedException?**

CloneNotSupportedException is a checked exception and as parent class method is not throwing CloneNotSupportedException , we can not throw it from our SingletonChild. Here is the code for your reference :

In such cases, you should throw RunTimeException

Can we create object of singleton class?

We declare it as private so that **no other classes can instantiate or make objects from it**. A private static variable of the same class that is the only instance of the class. Declare a static factory method with the return type as an object of this singleton class.

**Thread Safe Singleton:** A thread safe singleton in created so that singleton property is maintained even in multithreaded environment. To make a singleton class thread-safe, getInstance() method is made synchronized so that multiple threads can’t access it simultaneously.

synchronized public static GFG getInstance()

  {

    if (instance == null)

    {

      // if instance is null, initialize

      instance = new GFG();

    }

    return instance;

  }

# **Java Custom Exception**

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

**class** InvalidAgeException  **extends** Exception

{

**public** InvalidAgeException (String str)

    {

        // calling the constructor of parent Exception

**super**(str);

    }

}

**public** **class** TestCustomException1

{

    // method to check the age

**static** **void** validate (**int** age) **throws** InvalidAgeException{

**if**(age < 18){

        // throw an object of user defined exception

**throw** **new** InvalidAgeException("age is not valid to vote");

    }

**else** {

        System.out.println("welcome to vote");

        }

     }

 // main method

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

    {

**try**

        {

            // calling the method

            validate(13);

        }

**catch** (InvalidAgeException ex)

        {

            System.out.println("Caught the exception");

            // printing the message from InvalidAgeException object

            System.out.println("Exception occured: " + ex);

        }

        System.out.println("rest of the code...");

    }

}

**ArrayList maintains the insertion order** i.e order of the object in which they are inserted. HashSet is an unordered collection and doesn't maintain any order. ArrayList allows duplicate values in its collection. On other hand duplicate elements are not allowed in Hashset.

**Comparison between normal or regular class and static nested class**

| S.NO | Normal/Regular inner class | Static nested class |
| --- | --- | --- |
| 1. | Without an outer class object existing, there cannot be an inner class object. That is, the inner class object is always associated with the outer class object. | Without an outer class object existing, there may be a static nested class object. That is, static nested class object is not associated with the outer class object. |
| 2. | Inside normal/regular inner class, static members can’t be declared. | Inside static nested class, static members can be declared. |
| 3. | As main() method can’t be declared, regular inner class can’t be invoked directly from the command prompt. | As main() method can be declared, the static nested class can be invoked directly from the command prompt. |
| 4. | Both static and non static members of outer class can be accessed directly. | Only a static member of outer class can be accessed directly. |

**Main method**

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

**1. Can we overload the main in Java?**

Yes you can [overload](http://java67.blogspot.sg/2012/08/what-is-method-overloading-in-java-example.html) the main method in Java, nothing wrong with this but Java will only call your specific main method, i.e. main method with the following signature:

public static void main(String[] args) or public static void main(String args...) which is the main method as [variable argument method](http://javarevisited.blogspot.sg/2011/09/variable-argument-in-java5-varargs.html) and only supported post-Java 5 world.

**2. Can we override main in Java?**

No, you can not override the main method in Java, Why? because main is a [static method](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) and in Java static method is bonded during compile time and you can not [override](http://java67.blogspot.sg/2012/08/what-is-method-overriding-in-java-example-tutorial.html) the static method in Java. If you declare a method with the same name and signature it's called method hiding.

**3. Can we make the main final in Java?**

Of course, you can make the main method final in Java. JVM has no issue with that. Unlike any [final method](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html), you can not override main in Java.

**4. Can we make the main synchronized in Java?**

Yes, the main can be synchronized in Java, a [synchronized modifier](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) is allowed in the main signature and you can make your main method synchronized in Java.

**5. How to call a nonstatic method from main in Java?**

This question applies not only to main but all static methods in Java. Since [nonstatic methods can not be called from static context](http://javarevisited.blogspot.sg/2012/02/why-non-static-variable-cannot-be.html) directly, you need to first create an Object as a local variable and then you can call a non-static method using that object, as shown in the following example:

**import** java.util.Date;  
  
  
/\*\*  
 \* Java program to show how to call non-static method from the static method in Java  
 \*  
 \* @author http://java67.blogspot.com  
 \*/  
**public** **class** StaticTest {  
  
    **public** **static** **void** main(**String** args[]) {  
       
        *// calling non static method from main in Java*  
        *//printCurrentTime(); //compile time error - can not call non static method from main*  
       
        StaticTest test = **new** StaticTest();  
        test.printCurrentTime();  
       
    }  
   
   
    **public** **void** printCurrentTime(){  
       **System**.out.println(**new** **Date**());  
    }  
}

Why Main method is static?

The main() method is static **so that JVM can invoke it without instantiating the class**. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the main() method by the JVM

# [**Declaring the main method synchronized**](https://stackoverflow.com/questions/11261865/declaring-the-main-method-synchronized)

I saw a Java example that had a main method labeled as synchronized, calling another static synchronized method. The effect is that, basically, the other method runs on the separate thread only after the main method has returned.

What practical functionality would such a construct have?

public class SynchronisedMain {

public static synchronized void main(String[] args) throws InterruptedException {

new Thread(new Runnable() {

@Override

public void run() {

thingy();

}

}).start();

System.out.println("Kickstarted thingy thread.");

TimeUnit.MILLISECONDS.sleep(1000);

}

public static synchronized void thingy() {

System.out.println("Thingy!");

}

}

It's probably useful as a makeshift "application closed handler", doing some cleanup duty before the app finishes entirely. It's pretty contrived though...

In Java programs, the point from where the program starts its execution or simply the entry point of Java programs is the **main()** method. Hence, it is one of the most important methods of Java and having a proper understanding of it is very important.

The Java compiler or JVM looks for the main method when it starts executing a Java program. The signature of the main method needs to be in a specific way for the JVM to recognize that method as its entry point. If we change the signature of the method, the program compiles but does not execute.

The execution of Java program, the ***java.exe***is called. The Java.exe inturn makes Java Native Interface or JNI calls, and they load the JVM. The java.exe parses the command line, generates a new String array, and invokes the main() method. A daemon thread is attached to the main method, and this thread gets destroyed only when the Java program stops execution.

### 1. Public

It is an *Access modifier*, which specifies from where and who can access the method. Making the *main()* method public makes it globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class.

### 2. Static

It is a *keyword*that is when associated with a method, making it a class-related method. The *main()* method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the *main()* method by the JVM.

### 3. Void

It is a keyword and is used to specify that a method doesn’t return anything. As the *main()* method doesn’t return anything, its return type is *void*. As soon as the *main()* method terminates, the java program terminates too. Hence, it doesn’t make any sense to return from the *main()* method as JVM can’t do anything with the return value of it.

### 4. main

It is the name of the Java main method. It is the identifier that the JVM looks for as the starting point of the java program. It’s not a keyword.

Abandoned Object – 3 ways of GC

Going out of scope

Assigning new object

Assigning null

Mark and Sweep

Compact

Copy

Young Generation – eden and survivor(from and to)

Old(Tenured) Generation – after15 GC cycles

Full GC = Young GC + Old GC +PermaGen/Metasapace

# Difference Between Web server and Application server

* Last Updated : 12 Jul, 2019

A server is a central repository where information and computer programs are held and accessed by the programmer within the network. **Web server** and **Application server** are kinds of the server which employed to deliver sites and therefore the latter deals with application operations performed between users and back-end business applications of the organization.

**Web Server:**It is a computer program that accepts the request for data and sends the specified documents. Web server may be a computer where the online content is kept. Essentially internet server is employed to host sites however there exist different web servers conjointly like recreation, storage, FTP, email, etc.

**Example of Web Servers:**

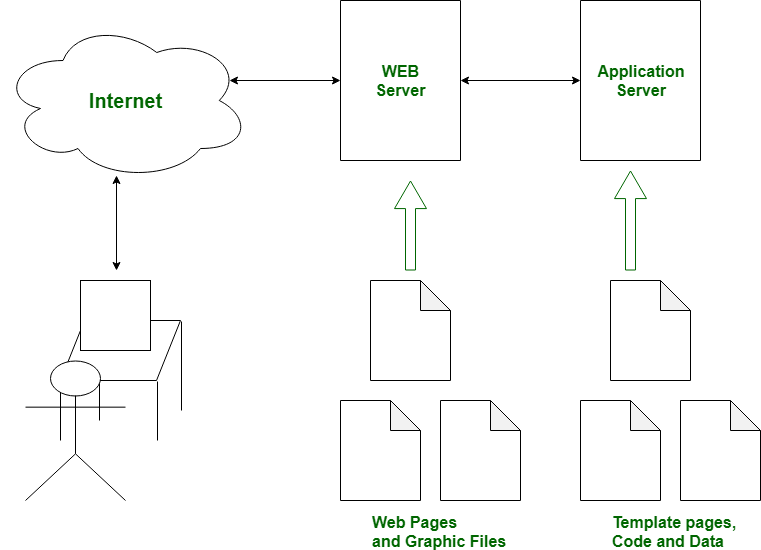
* Apache Tomcat
* Resin



**Application server:** It encompasses Web container as well as EJB container. Application servers organize the run atmosphere for enterprises applications. Application server may be a reasonably server that mean how to put operating system, hosting the applications and services for users, IT services and organizations. In this, user interface similarly as protocol and RPC/RMI protocols are used.

**Examples of Application Server:**

* Weblogic
* JBoss
* Websphere



**Difference between web server and application server:**

|  |  |  |
| --- | --- | --- |
| S.NO | Web Server | Application Server |
| 1. | Web server encompasses web container only. | While application server encompasses Web container as well as EJB container. |
| 2. | Web server is useful or fitted for static content. | Whereas application server is fitted for dynamic content. |
| 3. | Web server consumes or utilizes less resources. | While application server utilize more resources. |
| 4. | Web servers arrange the run environment for web applications. | While application servers arrange the run environment for enterprises applications. |
| 5. | In web servers, multithreading is not supported. | While in application server, multithreading is supported. |
| 6. | Web server’s capacity is lower than application server. | While application server’s capacity is higher than web server. |
| 7. | In web server, HTML and HTTP protocols are used. | While in this, GUI as well as HTTP and RPC/RMI protocols are used. |

| List | Set |
| --- | --- |
| The List is an ordered sequence. | The Set is an unordered sequence. |
| List allows duplicate elements | Set doesn’t allow duplicate elements. |
| Elements by their position can be accessed. | Position access to elements is not allowed. |
| Multiple null elements can be stored. | The null element can store only once. |

The **LinkedHashMap** **Class** is just like [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

**ConcurrentHashMap:**

* The underlined data structure for ConcurrentHashMap is [Hashtable](https://www.geeksforgeeks.org/hashtable-in-java/).
* ConcurrentHashMap class is thread-safe i.e. multiple threads can operate on a single object without any complications.
* At a time any number of threads are applicable for a read operation without locking the ConcurrentHashMap object which is not there in HashMap.
* In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level.
* The default concurrency-level of ConcurrentHashMap is 16.
* In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updated in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as **Segment locking or bucket locking**. Hence at a time, 16 update operations can be performed by threads.
* Inserting null objects is not possible in ConcurrentHashMap as a key or value.

## PUT vs POST Dilemma

In a typical [REST architecture](https://www.baeldung.com/cs/rest-architecture), a client sends requests in the form of HTTP methods to the server to create, retrieve, modify, or destroy resources. While both PUT and POST can be used to create resources, there are significant differences between them in terms of their intended applications.

According to the [RFC 2616](https://tools.ietf.org/html/rfc2616) standard, the POST method should be used to request the server to accept the enclosed entity as a subordinate of the existing resource identified by the Request-URI. This means **the POST method call will create a child resource** under a collection of resources.

On the other hand, the PUT method should be used to request the server to store the enclosed entity under the provided Request-URI. If the Request-URI points to an existing resource on the server, the supplied entity will be considered a modified version of the existing resource. Therefore, **the PUT method call will either create a new resource or update an existing one**.

Another important difference between the methods is that **PUT is an idempotent method while POST is not**. For instance, calling the PUT method multiple times will either create or update the same resource. On the contrary, multiple POST requests will lead to the creation of the same resource multiple times.

# [**When is mandatory to have a default constructor along with parameterized constructor in Java?**](https://stackoverflow.com/questions/12798416/when-is-mandatory-to-have-a-default-constructor-along-with-parameterized-constru)

he compiler doesn't ever enforce the existence of a default constructor. You can have any kind of constructor as you wish.

For some *libraries* or *frameworks* it might be necessary for a class to have a default constructor, but that is not enforced by the compiler.

The problem you might be seeing is if you have a class with a custom constructor and you don't have an implicit super() call in your constructor body. In that case the compiler will introduce a call to the super classes default constructor. If the super class doesn't have a default constructor then *your class* will fail to compile.

**If there is any one parametrized Constructor present in a class, Default Constructor will not be added at Compile time**. So if your program has any constructor containing parameters and no default constructor is specified then you will not be able to create object of that class using Default constructor.

In JPA, we have two options to define the composite keys: the @IdClass and @EmbeddedId annotations.

***@IdClass* can be quite useful in places where we** **are using a composite key class that we can't modify.**

If we're going to access parts of the composite key individually, we can make use of *@IdClass,*but **in places where we frequently use the complete identifier as an object, *@EmbeddedId* is preferred.**

**Joins**

**INNER JOIN:** The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.



**LEFT JOIN**: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.



**RIGHT JOIN**: RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

**FULL JOIN:** FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL*



## What Is a Circular Dependency?

A circular dependency occurs when a bean A depends on another bean B, and the bean B depends on bean A as well:

Bean A → Bean B → Bean A

Let's say we don't have a circular dependency. We instead have something like this:

Bean A → Bean B → Bean C

Spring will create bean C, then create bean B (and inject bean C into it), then create bean A (and inject bean B into it).

But with a circular dependency, Spring cannot decide which of the beans should be created first since they depend on one another. In these cases, Spring will raise a *BeanCurrentlyInCreationException* while loading context.

It can happen in Spring when using **constructor injection.** If we use other types of injections, we shouldn't have this problem since the dependencies will be injected when they are needed and not on the context loading.

### **Use**@Lazy

A simple way to break the cycle is by telling Spring to initialize one of the beans lazily. So, instead of fully initializing the bean, it will create a proxy to inject it into the other bean. The injected bean will only be fully created when it’s first needed.

To try this with our code, we can change the CircularDependencyA:

@Component

**public** **class** **CircularDependencyA** {

**private** CircularDependencyB circB;

@Autowired

**public** **CircularDependencyA**(@Lazy CircularDependencyB circB) {

this.circB = circB;

}

}

### **Use Setter/Field Injection**

Simply put, we can address the problem by changing the ways our beans are wired — to use setter injection (or field injection) instead of constructor injection. This way, Spring creates the beans, but the dependencies are not injected until they are needed.

### **Use**@PostConstruct

Another way to break the cycle is by injecting a dependency using @Autowired on one of the beans and then using a method annotated with @PostConstruct to set the other dependency.

### **Implement**ApplicationContextAware**and**InitializingBean

If one of the beans implements ApplicationContextAware, the bean has access to Spring context and can extract the other bean from there.

By implementing InitializingBean, we indicate that this bean has to do some actions after all its properties have been set. In this case, we want to manually set our dependency.

# **Type Casting in Java**

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

# **Java static nested class**

A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

It can access static data members of the outer class, including private.

The static nested class cannot access non-static (instance) data members or

**class** TestOuter1{

**static** **int** data=30;

**static** **class** Inner{

**void** msg(){System.out.println("data is "+data);}

  }

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

  TestOuter1.Inner obj=**new** TestOuter1.Inner();

  obj.msg();

  }

}

# **Java Member Inner class**

A non-static class that is created inside a class but outside a method is called **member inner class**. It is also known as a **regular inner class**. It can be declared with access modifiers like public, default, private, and protected.

**class** TestMemberOuter1{

**private** **int** data=30;

**class** Inner{

**void** msg(){System.out.println("data is "+data);}

 }

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

  TestMemberOuter1 obj=**new** TestMemberOuter1();

  TestMemberOuter1.Inner in=obj.**new** Inner();

  in.msg();

 }

}

# **Java Anonymous inner class**

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

Class (may be abstract or concrete).

Interface

**abstract** **class** Person{

**abstract** **void** eat();

}

**class** TestAnonymousInner{

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

  Person p=**new** Person(){

**void** eat(){System.out.println("nice fruits");}

  };

  p.eat();

 }

}

# **Java Local inner class**

A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

**public** **class** localInner1{

**private** **int** data=30;//instance variable

**void** display(){

**class** Local{

**void** msg(){System.out.println(data);}

  }

  Local l=**new** Local();

  l.msg();

 }

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

  localInner1 obj=**new** localInner1();

  obj.display();

 }

}

# How Does Spring Boot Solve This Problem?

1. Spring Boot does all of those using AutoConfiguration and will take care of all the internal dependencies that your application needs — all you need to do is run your application. Spring Boot will auto-configure with the Dispatcher Servlet, if Spring jar is in the class path. It will auto-configue to the datasource, if Hibernate jar is in the class path. Spring Boot gives us a pre-configured set of Starter Projects to be added as a dependency in our project.
2. During web-application development, we would need the jars that we want to use, which versions of the jars to use, and how to connect them together. All web applications have similar needs, for example, Spring MVC, Jackson Databind, Hibernate core, and Log4j (for logging). So, we had to choose the compatible versions of all these jars. In order to decrease the complexity, Spring Boot has introduced what we call **Spring Boot Starters.**

| **PATH** | **CLASSPATH** |
| --- | --- |
| 1. | An environment variable is used by the operating system to find the executable files. | An environment variable is used by the Java compiler to find the path of classes. |
| 2. | PATH setting up an environment for the operating system. Operating System will look in this PATH for executables. | Classpath setting up the environment for Java. Java will use to find compiled classes. |
| 3. | Refers to the operating system. | Refers to the Developing Environment. |
| 4. | In path variable, we must place .\bin folder path | In classpath, we must place .\lib\jar file or directory path in which .java file is available. |
| 5. | PATH is used by CMD prompt to find binary files. | CLASSPATH is used by the compiler and JVM to find library files. |

### Path

PATH is an environment variable that is used to find and locate binary files like “java” and “javac” and to locate needed executables from the command line or Terminal window. To set the path, we’re supposed to include or mention JDK\_HOME/bin directory in a PATH environment variable. The PATH can not be overridden by providing command and PATH is only used by the operation system(OS) to find binary files.

### Classpath

Classpath is an environment variable that is used by the application ClassLoader or system to locate and load the compiled Java bytecodes stored in the .class file. To set CLASSPATH. the CLASSPATH can be overridden by adding classpath in the manifest file and by using a command like set -classpath. the CLASSPATH is only used by Java ClassLoaders to load class files.

**HTTP Methods**

**HTTP GET:** The Hypertext Transfer Protocol(HTTP) Get method is mainly used at the client (Browser) side to send a request to a specified server to get certain data or resources. Using this method the server should only let us receive the data and not change its state. Hence it is only used to view something and not to change it. Get method is one of the most used HTTP method. The request parameter of the get method is appended to the URL. Get request is better for the data which does not need to be secure (It means the data which does not contain images or word documents).

**HTTP POST:** The Hypertext Transfer Protocol(HTTP) Post method is mainly used at the client (Browser) side to send data to a Specified server in order to create or rewrite a particular resource/data. This data sent to the server is stored in the request body of the HTTP request. Post method eventually leads to the creation of a new resource or updating an existing one. Due to this dynamic use, it is one of the most used HTTP methods. It is not one of the most secure methods because the data that is been sent is included in the body of the request and not in the URL. Post request is better for the data which needs to be secure (It means the data which contains images or word documents).

1. **PUT:** The PUT method is used to replace the already existing state of the resource or create the resource if the resource doesn’t exist (decided by the origin server) with the request payload.
2. **PATCH:** The PATCH method is used to make a partial update to a resource. It can be considered as an instruction set that describes the modification of data already present on the server to produce a new version of the data and it is not considered as the complete resource.
3. **DELETE:** The DELETE method is used to delete a specified resource on the origin server. The client cannot be assured that the operation was performed, even though the status code which is returned from the server indicates it.
4. **HEAD:** The HEAD is identical to the GET request but the server must not send the response body but should send the same headers as it would have sent if URI was requested with the GET method. It can be used to get metadata without transferring the response body from the server and is often used for testing hypertext links for validity, accessibility, and recent modification.
5. **CONNECT:** The CONNECT method is used to establish an end-to-end HTTP tunnel through a proxy server.
6. **OPTIONS:**The OPTIONS method requests allowed communication options for a given resource. This request can be sent to find out the supported HTTP methods and other options that are supported without requesting the resource.
7. **TRACE:** The TRACE method requests the server to send the exact request, as received by it, in the response body. It is useful for diagnostic purposes.

**Difference between PUT and POST methods**

| **PUT** | **POST** |
| --- | --- |
| PUT request is made to a particular resource. If the Request-URI refers to an already existing resource, an update operation will happen, otherwise create operation should happen if Request-URI is a valid resource URI (assuming client is allowed to determine resource identifier).  **Example –**  PUT /article/{article-id} | POST method is used to request that the origin server accept the entity enclosed in the  request as a new subordinate of the resource identified by the Request-URI in the Request-Line. It essentially means that POST request-URI should be of a collection URI.  **Example –**  POST /articles |
| PUT method is idempotent. So if you send retry a request multiple times, that should be equivalent to single request modification. | POST is NOT idempotent. So if you retry the request N times, you will end up having N resources with N different URIs created on server. |
| Use PUT when you want to modify a single resource which is already a part of resources collection. PUT overwrites the resource in its entirety. Use PATCH if request updates part of the resource. | Use POST when you want to add a child resource under resources collection. |
| Generally, in practice, always use PUT for UPDATE operations. | Always use POST for CREATE operations. |

**Difference between PUT and PATCH  request:**

| PUT | PATCH |
| --- | --- |
| PUT is a method of modifying resource where the client sends data that updates the entire resource . | PATCH is a method of modifying resources where the client sends partial data that is to be updated without modifying the entire data. |
| In a PUT request, the enclosed entity is considered to be a modified version of the resource stored on the origin server, and the client is requesting that the stored version be replaced | With PATCH, however, the enclosed entity contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version. |
| HTTP PUT is said to be idempotent, So if you send retry a request multiple times, that should be equivalent to a single request modification | HTTP PATCH is basically said to be non-idempotent. So if you retry the request N times, you will end up having N resources with N different URIs created on the server. |
| It has High Bandwidth | Since Only data that need to be modified if send in the request body as a payload , It has Low Bandwidth |

# OPTIONS

The **HTTP OPTIONS method** requests permitted communication options for a given URL or server. A client can specify a URL with this method, or an asterisk (\*) to refer to the entire server.

The HTTP OPTIONS method is used to describe communication options for the target resource. Browsers send an HTTP OPTIONS request to find out the supported HTTP methods and other options supported for the target resource before sending the actual request. HTTP OPTIONS requests allow clients to obtain parameters and requirements for specific resources and server capabilities without taking action on the resource or requesting the resource.  
  
The server response can include an Allow header indicating the allowed HTTP methods for this resource or various CORS (Cross-Origin Resource Sharing) headers. The HTTP OPTIONS method is both secure and idempotent and is only intended to provide information on how to interact with a resource. If you want to change data on the server, use [POST](https://reqbin.com/Article/HttpPost), [PUT](https://reqbin.com/Article/HttpPut), [PATCH](https://reqbin.com/Article/HttpPatch), or [DELETE](https://reqbin.com/Article/HttpDelete) methods.  
  
For security reasons, when you send data to a different domain (cross-domain requests), browsers usually send a 'preflight' HTTP OPTIONS request to the target server before sending the data there.

**Upload a file from REST api**

Upload a new file using a multipart request to specify destination and file information.

**Note:**

File names are not case-sensitive; that is, File\_Name and file\_name are considered identical. Two files cannot have the same name in a given folder.

By default, if a file with the same name exists in the target destination, the uploaded file replaces the existing file as a new revision.

The request is a multipart HTTP request where one part is a JSON object (jsonInputParameters) with the parentID and the other part is the content of the file itself (primaryFile). It is also possible to set metadata values using another JSON object (metadataValues).

1. Base64 encode the file, at the expense of increasing the data size by around 33%, and add processing overhead in both the server and the client for encoding/decoding.
2. Send the file first in a multipart/form-data POST, and return an ID to the client. The client then sends the metadata with the ID, and the server re-associates the file and the metadata.
3. Send the metadata first, and return an ID to the client. The client then sends the file with the ID, and the server re-associates the file and the metadata.

| Class Level Lock | Object Level Lock |
| --- | --- |
| This lock can be used when we want to prevent multiple threads to enter the synchronized block of available instances on runtime. | This lock is used when we want a non-static method or non-static block of our code should be accessed by only one thread at a time. |
| This lock is used to make static data thread-safe. | This lock is used to make non-static data thread-safe. |
| Multiple objects of a particular class may exist but there is always one class’s class object lock available. | Every object the class have their own lock. |
| We can get a class level lock as follows:  public class GFG {    public void m1( ) {       synchronized (GFG.class) {      // some line of code    }  } | We can get object level lock as follows:  public class GFG {    public void m1( ) {       synchronized (this) {          // some line of code       }    }  } |

**Lombok**

[Lombok](https://projectlombok.org/) is a very handy tool for minimizing the boilerplate code and providing lots of other features such as lazy loading, [thread safety](https://howtodoinjava.com/java/multi-threading/what-is-thread-safety/) or immutability.

Automating the boilerplate code generation for any java class. So if Lombok is in the classpath, it can quickly get rid of all the getters & setters methods, [class constructors](https://howtodoinjava.com/java/oops/java-constructors/), [hashcode and equals](https://howtodoinjava.com/java/basics/java-hashcode-equals-methods/) methods and many more by adding a couple of annotations to the class.

Inner Join

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns.

Why setter injection is better than constructor injection?

Constructor-based DI fixes the order in which the dependencies need to be injected. Setter-based DI **helps us to inject the dependency only when it is required, as opposed to requiring it at construction time**. Spring code generation library doesn't support constructor injection so it will not be able to create proxy.

| **Sr. No.** | **Key** | **Constructor based Injection** | **Setter based Injection** |
| --- | --- | --- | --- |
| 1 | Circular | It doesn’t allow to create circular dependency | It doesn’t check the circular dependency |
| 2 | Ordering | Constructor-based DI fixes the order in which the dependencies need to be injected. | Setter-based DI helps us to inject the dependency only when it is required, as opposed to requiring it at construction time. |
| 3 | MutilThread Environment | Combining with final fields, constructor injection gives extra safety in multithreaded environment | No extra benefit in setter injection |
| 4 | Spring Code generation Library | Spring code generation library doesn’t support constructor injection so it will not be able to create proxy. It will force you to use no-argument constructor. | Spring framework level code uses setter injection |
| 5 | Use Case | It should be used for mandatory dependencies | It should be used for optional dependencies. |

Inversion of control is a design principle which helps to invert the control of object creation.

According to the paper written by Martin Fowler , inversion of control is the principle where the control flow of a program is inverted: instead of the programmer controlling the flow of a program, the external sources (framework, services, other components) take control of it. It's like we plug something into something else. He mentioned an example about EJB 2.0.

Dependency Injection is a design pattern which implements IOC principle. DI provides objects that an object needs. Let’s say, class X is dependent on Y. So rather than creating object of Y within the class “X”, we can inject the dependencies via a constructor or setter injection.

# **Why Wait(), Notify() and NotifyAll() Defined in Object Class**

n the Java language, you wait() on a particular instance of an Object – a monitor assigned to that object to be precise. If you want to send a signal to one thread that is waiting on that specific object instance then you call notify() on that object. If you want to send a signal to all threads that are waiting on that object instance, you use notifyAll() on that object.

If wait() and notify() were on the Thread instead then each thread would have to know the status of every other thread. How would thread1 know that thread2 was waiting for access to a particular resource? If thread1 needed to call thread2.notify() it would have to somehow find out that thread2 was waiting. There would need to be some mechanism for threads to register the resources or actions that they need so others could signal them when stuff was ready or available.

In Java, the object itself is the entity that is shared between threads which allows them to communicate with each other. The threads have no specific knowledge of each other and they can run asynchronously. They run and they lock, wait, and notify on the *object* that they want to get access to. They have no knowledge of other threads and don't need to know their status. They don't need to know that it is thread2 which is waiting for the resource – they just notify on the resource and whomever it is that is waiting (if anyone) will be notified.

# Serialization and Deserialization

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the **java.io.Serializable** interface.  
The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

The ObjectInputStream class contains **readObject()** method for deserializing an object.

**SerialVersionUID**  
The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name.  
It must be static, final and of type long.  
i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

It is also recommended to use private modifier for UID since it is not useful as inherited member.

**serialver**  
The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.  
You can run the following command to get serialVersionUID

serialver [-classpath classpath] [-show] [classname…]

# How to prevent Singleton Pattern from Reflection, Serialization and Cloning?

**1.Reflection:** [Reflection](https://www.geeksforgeeks.org/reflection-in-java/) can be caused to destroy singleton property of singleton class.

Constructor[] constructors =

                    Singleton.class.getDeclaredConstructors();

            for (Constructor constructor : constructors)

            {

                // Below code will destroy the singleton pattern

                constructor.setAccessible(true);

                instance2 = (Singleton) constructor.newInstance();

                break;

            }

**Overcome reflection issue:** To overcome issue raised by reflection, [enums](https://www.geeksforgeeks.org/enum-in-java/) are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

As enums don’t have any constructor so it is not possible for Reflection to utilize it. Enums have their by-default constructor, we can’t invoke them by ourself.**JVM handles the creation and invocation of enum constructors internally.** As enums don’t give their constructor definition to the program, it is not possible for us to access them by Reflection also. Hence, reflection can’t break singleton property in case of enums.

**2.Serialization:-** [Serialization](https://www.geeksforgeeks.org/serialization-in-java/) can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of byte stream and save in a file or send over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

**Overcome serialization issue:-** To overcome this issue, we have to implement method readResolve() method.

**3.Cloning:** [Cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/) is a concept to create duplicate objects. Using clone we can create copy of object. Suppose, we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

**Overcome Cloning issue:-** To overcome this issue, override clone() method and throw an exception from clone method that is CloneNotSupportedException. Now whenever user will try to create clone of singleton object, it will throw exception and hence our class remains singleton.

**Shallow Copy** 

* Whenever we use default implementation of clone method we get shallow copy of object means it creates new instance and copies all the field of object to that new instance and returns it as object type, we need to explicitly cast it back to our original object. This is shallow copy of the object.
* clone() method of the object class support shallow copy of the object. If the object contains primitive as well as non primitive or reference type variable in shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves.
* That’s why the name shallow copy or shallow cloning in Java. If only primitive type fields or Immutable objects are there then there is no difference between shallow and deep copy in Java.

public Ex(int[] values) {

        data = values;

    }

**Deep Copy** 

* Whenever we need own copy not to use default implementation we call it as deep copy, whenever we need deep copy of the object we need to implement according to our need.
* So for deep copy we need to ensure all the member class also implement the Cloneable interface and override the clone() method of the object class.

A deep copy means actually creating a new array and copying over the values.

 public Ex(int[] values) {

        data = new int[values.length];

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

            data[i] = values[i];

        }

Spring Aware

Spring Aware interfaces allow you to look into the inner workings of the Spring Framework. Through Spring Aware interfaces, you can access the Spring context, or Spring bean lifecycle events.

Your Spring beans might require access to framework objects, such as ApplicationContext, BeanFactory, and ResourceLoader. To gain access, a bean can implement one of the many Aware interfaces of the Spring Framework.

When a bean implements an Aware interface, the Spring Framework injects a particular framework object to the bean through a callback-style method. The object Spring injects depends on the interface which the bean implements. For example, if the bean implements the ApplicationContextAware interface, Spring will inject an ApplicationContext object into the bean.

In the bean lifecycle, the Spring Framework calls the aware interface methods after populating bean properties and just before pre initialization with BeanPostProcessor.

**BeanNameAware makes the object aware of the bean name defined in the container**.

**BeanFactoryAware is used to inject the BeanFactory object**. This way we get access to the BeanFactory which created the object.

 If your bean needs to look up some other beans. Similarly, if your bean needs access to some application file resource in your bean or even publish some application events, you need access to the ApplicationContext.

Spring provides an ApplicationContextAware interface that allows beans access to the ApplicationContext. This interface provides a single setApplicationContext method.

## Taking an Initial Approach to Scaling

Deciding when and how to approach scaling your service can be a daunting task, but it doesn't have to be. In order to effectively scale, engineers must have a solid understanding of the behavior of the system, data to support their hypothesis, and a method to test the results. We'll talk more about measuring the performance later in this series, but for now it's important to understand that most effective scaling strategies involve "Just In Time" optimizations.

It's critical to have an environment and culture which allows for production level testing via canary type deployments or synthetic testing infrastructure in place to allow for test engineers to develop load tests which closely emulate production workloads. If neither of those are available in your organization, you're effectively shooting in the dark when it comes to scaling and optimization.

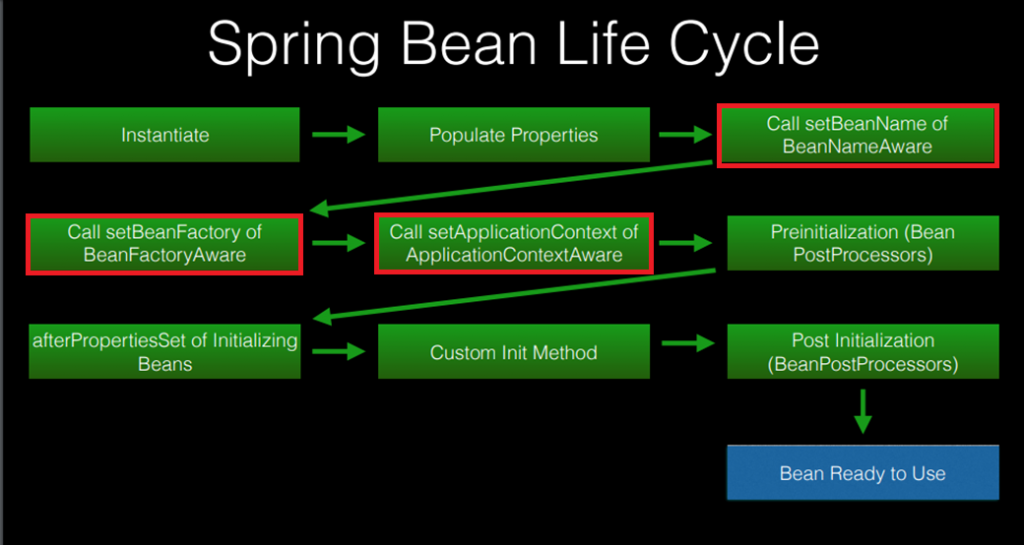
In the example below we traverse through 4 distinct phases of scaling, and the problems associated with each one. Later in the series we'll discuss each one in detail, but it's a good overview of the types of problems you'll encounter when scaling distributed systems.

1. In Service B we've experienced thread pool exhaustion because the combined traffic from A and C have overwhelmed the service. The natural response is to increase the thread count but this quickly causes issue.
2. Once we've added more threads this creates additional load and connection exhaustion on the database. The initial response might be to utilize connection pooling but eventually even this will not be enough. We would then scale the database size vertically to accommodate the load.
3. Eventually, we need to scale Service B more by adding more replicas, this creates even more load on the database, potentially to the point where a single instance can no longer sustain the load.
4. We may implement sharding, or utilize a distributed database solution such as Cassandra or Mongo. Either way, we've now implemented a solution which should enable significant horizontal scaling solutions.

Bean Lifecycle

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.

The following image shows the process flow of the bean life cycle.





### Static Binding

The binding which can be resolved at compile time by the compiler is known as static or early binding. The binding of all the static, private, and final methods is done at compile-time.

### Dynamic Binding

In Dynamic binding compiler doesn’t decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have the same method.

| Static Binding | Dynamic Binding |
| --- | --- |
| It takes place at compile time for which is referred to as early binding | It takes place at runtime so do it is referred to as late binding. |
| It uses overloading more precisely operator overloading method | It uses overriding methods. |
| It takes place using normal functions | It takes place using virtual functions |
| Real objects never use static binding | Real objects use dynamic binding. |

## ****Reference Types****

### 1. Strong Reference

These are the most popular reference types that we all are used to. In the example above with the StringBuilder, we actually hold a strong reference to an object from the heap. The object on the heap it is not garbage collected while there is a strong reference pointing to it, or if it is strongly reachable through a chain of strong references.

### **2. Weak Reference**

In simple terms, a weak reference to an object from the heap is most likely to not survive after the next garbage collection process. A weak reference is created as follows:

1

WeakReference<StringBuilder> reference = new WeakReference<>(new StringBuilder());

A nice use case for weak references are caching scenarios. Imagine that you retrieve some data, and you want it to be stored in memory as well — the same data could be requested again. On the other hand, you are not sure when, or if, this data will be requested again. So you can keep a weak reference to it, and in case the garbage collector runs, it could be that it destroys your object on the heap. Therefore, after a while, if you want to retrieve the object you refer to, you might suddenly get back a null value. A nice implementation for caching scenarios is the collection **WeakHashMap<K,V>**. If we open the WeakHashMap class in the Java API, we see that its entries actually extend the WeakReference class and uses its **ref**field as the map’s key:

1

/\*\*

2

\* The entries in this hash table extend WeakReference, using its main ref

3

\* field as the key.

4

\*/

5

​

6

private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V> {

7

​

8

V value;

Once a key from the WeakHashMap is garbage collected, the entire entry is removed from the map.

### **3. Soft Reference**

These types of references are used for more memory-sensitive scenarios, since those are going to be garbage collected only when your application is running low on memory. Therefore, as long as there is no critical need to free up some space, the garbage collector will not touch softly reachable objects. Java guarantees that all soft referenced objects are cleaned up before it throws an OutOfMemoryError. The Javadocs state, “all soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an OutOfMemoryError.”

Similar to weak references, a soft reference is created as follows:

1

SoftReference<StringBuilder> reference = new SoftReference<>(new StringBuilder());

### **4. Phantom Reference**

Used to schedule post-mortem cleanup actions, since we know for sure that objects are no longer alive. Used only with a reference queue, since the .get() method of such references will always return null**.** These types of references are considered preferable to **finalizers.**

### SQL Dialects

**Dialect**is a class that acts as a bridge between **Java JDBC types** and **SQL types**, which contains the mapping between java language data type and database datatype. Dialect allows Hibernate to generate SQL optimized for a particular relational database. Hibernate generates queries for the specific database based on the **Dialect**class. A hibernate dialect gives information to the framework of how to convert **hibernate queries(HQL)** into native **SQL queries**.

Since **Hibernate**is database agnostic. It can work with different databases. However, databases have proprietary extensions/native SQL variations, and set/sub-set of SQL standard implementations. Due to this hibernate has to use database-specific SQL. Dialect specifies the type of database used in hibernate so that hibernate can switch to the database-specific SQL generator code. Dialects can be used in the following ways:

* To generate Optimized SQL queries
* To interact with a particular Database if the application works with the help of more than one Database.
* To set default values for hibernate configuration file properties based on the database software even though they are not specified in the configuration file.

Get() and Load() in Hibernate

get() returns the object by fetching it from database or from [hibernate cache](https://www.journaldev.com/2969/hibernate-caching-first-level-cache) whereas load() just returns the reference of an object that might not actually exists, it loads the data from database or cache only when you access other properties of the object.

1. get() loads the data as soon as it’s called whereas load() returns a proxy object and loads data only when it’s actually required, so load() is better because it support lazy loading.
2. Since load() throws exception when data is not found, we should use it only when we know data exists.
3. We should use get()

**Log4Shell**

The Log4j 2 library controls how applications log strings of code and information. The vulnerability enables an attacker to gain control over a string and trick the application into requesting and executing malicious code under the attacker’s control. As a result, attackers can remotely take over any internet-connected service that uses certain versions of the Log4j library anywhere in the software stack.

Because the Log4j 2 library can communicate with other sources and internal directory services, attackers can easily feed Log4j 2 with malicious commands from the outside and make it download and execute dangerous code from malicious sources.

**JWT – OAuth**

**JWT**, or JSON Web Token, is an open standard used to share security information between two parties — a client and a server. Each JWT contains encoded JSON objects, including a set of claims. JWTs are signed using a cryptographic algorithm to ensure that the claims cannot be altered after the token is issued.

### What Is JSON?

For beginning developers, JSON stands for JavaScript Object Notation and is a text-based format for transmitting data across web applications. It stores information in an easy-to-access manner, both for developers and computers. It can be used as a data format by any programming language and is quickly becoming the preferred syntax for APIs, surpassing XML.

JWTs differ from other web tokens in that they contain a set of claims. Claims are used to transmit information between two parties. What these claims are depends on the use case at hand. For example, a claim may assert who issued the token, how long it is valid for, or what permissions the client has been granted.

A JWT is a string made up of three parts, separated by dots (.), and serialized using base64. In the most common serialization format, compact serialization, the JWT looks something like this: xxxxx.yyyyy.zzzzz.

Once decoded, you will get two JSON strings:

1. The **header**and the **payload.**
2. The **signature.**

The **JOSE (JSON Object Signing and Encryption) header**contains the type of token — JWT in this case — and the signing algorithm.

The **payload**contains the claims. This is displayed as a JSON string, usually containing no more than a dozen fields to keep the JWT compact. This information is typically used by the server to verify that the user has permission to perform the action they are requesting.

There are no mandatory claims for a JWT, but overlaying standards may make claims mandatory. For example, when using JWT as bearer access token under OAuth2.0, iss, sub, aud, and exp must be present. some are more common than others.

The **signature**ensures that the token hasn’t been altered. The party that creates the JWT signs the header and payload with a secret that is known to both the issuer and receiver, or with a private key known only to the sender. When the token is used, the receiving party verifies that the header and payload match the signature.

Common way to use JWTs is as [OAuth](https://www.akana.com/blog/what-is-oauth) bearer tokens. In this example, an authorization server creates a JWT at the request of a client and signs it so that it cannot be altered by any other party. The client will then send this JWT with its request to a [REST API](https://www.akana.com/blog/what-is-rest-api). The REST API will verify that the JWT’s signature matches its payload and header to determine that the JWT is valid. When the REST API has verified the JWT, it can use the claims to either grant or deny the client’s request.

In simpler terms, you can think of a JWT bearer token as an identity badge to get into a secured building. The badge comes with special permissions (the claims); that is, it may grant access to only select areas of the building. The authorization server in this analogy is the reception desk — or the issuer of the badge. And to verify that the badge is valid, the company logo is printed on it, similar to the signature of the JWT. If the badge holder attempts to access a restricted area, the permissions on the badge determine whether or not they can access the area, similar to the claims in a JWT.

JWTs are used as a secure way to authenticate users and share information.

Typically, a private key, or secret, is used by the issuer to sign the JWT. The receiver of the JWT will verify the signature to ensure that the token hasn’t been altered after it was signed by the issuer. It is difficult for unauthenticated sources to guess the signing key and attempt to change the claims within the JWT.

Not all signing algorithms are created equal though. For example, some signing algorithms use a secret value that is shared between the issuer and the party that verifies the JWT. Other algorithms use a public and private key. The private key is known only to the issuer, while the public key can be widely distributed. The public key can be used to verify the signature, but only the private key can be used to create the signature. This is more secure than a shared secret because the private key only needs to exist in one place.

Because of this, the server does not need to keep a database with the information needed to identify the user. For developers, this is great news — the server that issues the JWT and the server that validates it do not have to be the same.

**Sonar rules examples**

NullpointerException could be thrown. Filed is nullable here.

Define and throw dedicated exception instead of generic one.

Iterate over a entrySet instead of keyset.

Replace this synchronized hashtable with unsynchronized hashmap.

**JMS**

JMS is a specification. JMS has three main parts to it. The first is the **producer**, which is nothing more than a bean that submits a "message" to a JMS **broker** (#2) (the system that manages messages between **producers** and **consumers**). In this case, ActiveMQ is the broker. Once the broker receives a message, the **consumer** (#3), or Message-Driven Bean (MDB), processes the message.

If you want to work with JMS, you'll just write both your producer/consumer code using the JMS API, but behind the scenes there is a "resource adapter" that is a special ActiveMQ driver that will connect to an ActiveMQ instance and do the management for you.

Message queue and Message broker

A message queue is a data structure, or a container - a way to hold messages for eventual consumption. A message broker is a separate component that manages queues.

Message broker

An intermediary program that translates a system’s language from one internationally suitable language to another via a telecommunication medium is termed as Message Broker. It is primarily used for message validation, transformation and routing. A message broker is also known as middleware or integration broker. There are three types of message brokers – *point-to-point, publish-subscribe,*and*a hybrid of both*. The purpose of a broker is to –

* Route or transfer messages to one or more destinations with an alternative representation.
* Perform message aggregation, and to decompose them to multiple messages
* Sending messages to their destination
* Recomposing the responses into one message
* Respond to the messages received / errors
* Communicate with external repository to augment a message

**Connection Pooling**

Connection pooling is a well-known data access pattern, whose main purpose is to reduce the overhead involved in performing database connections and read/write database operations.

**A connection pool is, at the most basic level, a database connection cache implementation**, which can be configured to suit specific requirements.

Typical database connection life cycle

1. Opening a connection to the database using the database driver
2. Opening a [TCP socket](https://en.wikipedia.org/wiki/Network_socket) for reading/writing data
3. Reading / writing data over the socket
4. Closing the connection
5. Closing the socket

It becomes evident that **database connections are fairly expensive operations**, and as such, should be reduced to a minimum in every possible use case (in edge cases, just avoided).

Here's where connection pooling implementations come into play.

By just simply implementing a database connection container, which allows us to reuse a number of existing connections, we can effectively save the cost of performing a huge number of expensive database trips, hence boosting the overall performance of our database-driven applications.

**HQL**

**The execute() method:** This method is used to execute SQL DDL statements, it returns a boolean value specifying weather the ResultSet object can be retrieved.

**executeUpdate():** This method is used to execute statements such as insert, update, delete. It returns an integer value representing the number of rows affected.

**executeQuery():** This method is used to execute statements that returns tabular data (example select). It returns an object of the class ResultSet.

**Producer – Consumer Problem**

In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

* The producer’s job is to generate data, put it into the buffer, and start again.
* At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

**Problem**   
To make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer.

**Solution**  
The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.   
An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

## Topics

In JMS a Topic implements publish and subscribe semantics. When you publish a message it goes to all the subscribers who are interested - so zero to many subscribers will receive a copy of the message. Only subscribers who had an active subscription at the time the broker receives the message will get a copy of the message.

## Queues

A JMS Queue implements load balancer semantics. A single message will be received by exactly one consumer. If there are no consumers available at the time the message is sent it will be kept until a consumer is available that can process the message. If a consumer receives a message and does not acknowledge it before closing then the message will be redelivered to another consumer. A queue can have many consumers with messages load balanced across the available consumers.

Queues and Topics are similar when a sender sends messages, but messages are processed differently by a receiver. **A queue can have only one consumer, whereas a topic can have multiple subscribers**.

**JPA – Relational Mapping**

**@Entity**

**public class Teacher {**

**private String firstName;**

**private String lastName;**

**}**

**@Entity**

**public class Course {**

**private String title;**

**}**

## @Entity

## public class Teacher {

## // ...

## @OneToMany(mappedBy = "teacher")

## private List<Course> courses;

## }

## @Entity

## public class Course {

## // ...

## 

## @ManyToOne

## @JoinColumn(name = "TEACHER\_ID", referencedColumnName = "ID")

## private Teacher teacher;

## }

## @OneToMany(mappedBy = "teacher", fetch = FetchType.EAGER)

## private List<Course> courses;

## @ManyToOne(fetch = FetchType.LAZY)

## private Teacher teacher;

## @ManyToOne(optional = false, cascade = CascadeType.PERSIST)

## @JoinColumn(name = "TEACHER\_ID", referencedColumnName = "ID")

## private Teacher teacher;

## @ManyToMany

## @JoinTable(

## name = "STUDENTS\_COURSES",

## joinColumns = @JoinColumn(name = "COURSE\_ID", referencedColumnName = "ID"),

## inverseJoinColumns = @JoinColumn(name = "STUDENT\_ID", referencedColumnName = "ID")

## )

## private List<Student> students;

## **JPQL**

@Query(value = "SELECT e FROM Employee e")

List<Employee> **findAllEmployees**(Sort sort);

@Query("SELECT e FROM Employee e WHERE e.name = ?1 and e.salary = ?2")

Employee **findEmployeeByNameAndSalary**(String name, Long salary);

@Query( value = "SELECT \* FROM Employee e WHERE e.salary = ?1", nativeQuery = true)

Employee **findEmployeeBySalaryNative**(Long salary);

HNQ

@NamedQuery(name = "Employee\_FindByEmployeeId", query = "from Employee where id = :id")

JPA Queries

**public** UserEntity **getUserByIdWithPlainQuery**(Long id) { **Query** jpqlQuery = getEntityManager().createQuery("SELECT u FROM UserEntity u WHERE u.id=:id"); jpqlQuery.setParameter("id", id);

**return** (UserEntity) jpqlQuery.getSingleResult();

}

**Typed Query**

**public** UserEntity **getUserByIdWithTypedQuery**(Long id) { TypedQuery<UserEntity> typedQuery = getEntityManager().createQuery("SELECT u FROM UserEntity u WHERE u.id=:id", UserEntity.class); typedQuery.setParameter("id", id);

**return** typedQuery.getSingleResult();

}

**Named Query**

@NamedQuery(name = "UserEntity.findByUserId", query = "SELECT u FROM UserEntity u WHERE u.id=:userId")

**public** UserEntity **getUserByIdWithNamedQuery**(Long id) { **Query** namedQuery = getEntityManager().createNamedQuery("UserEntity.findByUserId"); namedQuery.setParameter("userId", id);

**return** (UserEntity) namedQuery.getSingleResult();

}

**Native Query**

**public** UserEntity **getUserByIdWithNativeQuery**(Long id) { **Query** nativeQuery = getEntityManager().createNativeQuery("SELECT \* FROM users WHERE id=:userId", UserEntity.class); nativeQuery.setParameter("userId", id);

**return** (UserEntity) nativeQuery.getSingleResult()

}

**Criteria Query**

**public** UserEntity **getUserByIdWithCriteriaQuery**(Long id) { **CriteriaBuilder** criteriaBuilder = getEntityManager().getCriteriaBuilder(); CriteriaQuery<UserEntity> criteriaQuery = criteriaBuilder.createQuery(UserEntity.class); Root<UserEntity> userRoot = criteriaQuery.from(UserEntity.class);

**UserEntity** queryResult = getEntityManager().createQuery(criteriaQuery.select(userRoot) .where(criteriaBuilder.equal(userRoot.get("id"), id))) .getSingleResult();

**return** queryResult;

}

**Pagination and Sorting**

public ResponseEntity<List<EmployeeEntity>> getAllEmployees(

                        @RequestParam(defaultValue = "0") Integer pageNo,

                        @RequestParam(defaultValue = "10") Integer pageSize,

                        @RequestParam(defaultValue = "id") String sortBy)

public List<EmployeeEntity> getAllEmployees(Integer pageNo, Integer pageSize, String sortBy)

    {

        Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by(sortBy));

        Page<EmployeeEntity> pagedResult = repository.findAll(paging);

        if(pagedResult.hasContent()) {

            return pagedResult.getContent();

        } else {

            return new ArrayList<EmployeeEntity>();

        }

    }

Blocking Queue

A [Queue](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Queue.html) that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

We can distinguish two types of *BlockingQueue*:

* unbounded queue – can grow almost indefinitely
* bounded queue – with maximal capacity defined

unbounded queue

The Capacity of blockingQueue will be set to Integer.MAX\_VALUE. All operations that add an element to the unbounded queue will never block, thus it could grow to a very large size.

bounded queue

The second type of queues is the bounded queue. We can create such queues by passing the capacity as an argument to a constructor:

# **SynchronousQueue**

The *SynchronousQueue*only has **two supported operations: *take()*and *put(),*and both of them are blocking**.

For example, when we want to add an element to the queue, we need to call the *put()*method. That method will block until some other thread calls the *take()*method, signaling that it is ready to take an element.

Although the *SynchronousQueue* has an interface of a queue, we should think about it as an exchange point for a single element between two threads, in which one thread is handing off an element, and another thread is taking that element.

# **DelayQueue**

This is a blocking queue that could be used in producer-consumer programs.

It has a very useful characteristic – **when the consumer wants to take an element from the queue, they can take it only when the delay for that particular element has expired.**

this queue allows us to create programs according to the producer-consumer pattern, and coordinate messages passing from producers to consumers.

# **TransferQueue**

The implementation is actually similar to the [*BlockingQueue*](https://www.baeldung.com/java-blocking-queue) – but gives us the new ability to implement a form of backpressure. This means that, when the producer sends a message to the consumer using the transfer() method, the producer will stay blocked until the message is consumed.

**ForkJoin Pool**

**The framework first “forks,”** recursively breaking the task into smaller independent subtasks until they are simple enough to run asynchronously.

After that, **the “join” part begins.** The results of all subtasks are recursively joined into a single result. In the case of a task that returns void, the program simply waits until every subtask runs.

To provide effective parallel execution, the fork/join framework uses a pool of threads called the ForkJoinPool. This pool manages worker threads of type ForkJoinWorkerThread.

**ForkJoinPool** commonPool = ForkJoinPool.commonPool();

ForkJoinTask is the base type for tasks executed inside ForkJoinPool. In practice, one of its two subclasses should be extended: the RecursiveAction for void tasks and the RecursiveTask<V> for tasks that return a value.They both have an abstract method compute() in which the task’s logic is defined.

createSubtasks() method returns a List<CustomRecursiveAction>.

The list is submitted to the ForkJoinPool using the invokeAll() method:

Submit tasks -

**submit()**or ***execute()*** method (their use cases are the same):

forkJoinPool.execute(customRecursiveTask);

**int** result = customRecursiveTask.join();

The ***invoke()*** method forks the task and waits for the result, and doesn’t need any manual joining:

**int** result = forkJoinPool.invoke(customRecursiveTask);

What happens if we use @Repository and @Service vice-versa?

According to [documentaion](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-stereotype-annotations) @Repository,@Service,@Controller are all synonyms. They all are just specializations of @Component annotation. So, generally, they can be used one instead of other. But ... you should not do this.

First reason: any of these annotations make clear the role of your component in the application. Shows - is this component belongs to the controller, service, or data layer.

Second reason: some of these annotations processed differently by different Spring modules. For example, Spring Data JPA will process @Repository and will try to replace with implementation any interface marked by this annotation. Spring also will apply automatic exception translation to such classes. Another example: Spring Web MVC processes @Controller, and uses classes marked with it in URL mappings.

Bean post-processor that automatically applies persistence exception translation to any bean marked with Spring's @[Repository](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html) annotation, adding a corresponding [PersistenceExceptionTranslationAdvisor](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/annotation/PersistenceExceptionTranslationAdvisor.html) to the exposed proxy (either an existing AOP proxy or a newly generated proxy that implements all of the target's interfaces).

Translates native resource exceptions to Spring's [DataAccessException](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/DataAccessException.html) hierarchy. Autodetects beans that implement the [PersistenceExceptionTranslator](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/support/PersistenceExceptionTranslator.html) interface, which are subsequently asked to translate candidate exceptions.

All of Spring's applicable resource factories (e.g. [LocalContainerEntityManagerFactoryBean](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/orm/jpa/LocalContainerEntityManagerFactoryBean.html)) implement the PersistenceExceptionTranslator interface out of the box. As a consequence, all that is usually needed to enable automatic exception translation is marking all affected beans (such as Repositories or DAOs) with the @Repository annotation, along with defining this post-processor as a bean in the application context.

## Why Semantic Monitoring?

Typically, microservices-based applications are deployed in a [container format](https://www.webopedia.com/definitions/container/) and monitored using the basic container monitoring architecture, which monitors CPU and memory usage metrics. Container infrastructures often lack a sophisticated monitoring system. Even if there is one, it would hardly have the capability to monitor business process semantics. As a result, we must consider two monitoring vectors for microservices: service layer monitoring and semantic monitoring.

## Semantic Monitoring for Microservices

Albeit a painful task, microservices monitoring is important for the success of microservices architecture. However, there are many constraints and challenges involved. As an example, with simultaneous execution of many different services, it often becomes difficult to ascertain the root cause of a failure. Here’s where Semantic monitoring comes for the rescue.

Ideally, a monitoring solution should be able to collect data and send out alerts if there is an anomaly. Synthetic transaction monitoring or semantic monitoring does just that and much more. It is an altogether new approach towards monitoring an application using synthetic transactions by emulating end user flows. This is achieved by running a subset of an application’s automated tests against a production system at regular intervals of time.

These tests mimic end user flows and check for availability, functionality, performance, and response times. The results are then fed to a monitoring solution. This process of verifying the semantic correctness of a system is known as semantic monitoring. Semantic monitoring is particularly useful for monitoring e-commerce and other high traffic sites as well as microservices applications.

### **Types of Semantic Monitoring**

Basically, there are three types of semantic monitoring. These include the following:

* **Availability Monitoring** – availability monitoring is also known as uptime monitoring and is used to check if the site or the service is up and running and is accessible.
* **Performance Monitoring** – performance monitoring is used to check for page load speed, backend response times and slow performance.
* **Transaction Monitoring** – transaction monitoring is used to check if a transaction is working as expected.

Spring Started dependencies

Before Spring Boot was introduced, Spring Developers used to spend a lot of time on Dependency management. Spring Boot Starters were introduced to solve this problem so that the developers can spend more time on actual code than Dependencies. Spring Boot Starters are dependency descriptors that can be added under the **<dependencies>** section in pom.xml. There are around 50+ Spring Boot Starters for different Spring and related technologies. These starters give all the dependencies under a single name. For example, if you want to use Spring Data JPA for database access, you can include **spring-boot-starter-data-jpa**dependency.

The advantages of using Starters are as follows:

* Increase productivity by decreasing the Configuration time for developers.
* Managing the POM is easier since the number of dependencies to be added is decreased.
* Tested, Production-ready**,** and supported dependency configurations.
* No need to remember the name and version of the dependencies.

Why new Date/Time 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/LocatTime**and **LocalDateTime API :**Use it when time zones are NOT required.
* **Zoned date-time API**: Use it when time zones are to be considered

**OutOfMemory Error**

In Java the objects that occupy memory are all linked to some other objects, forming a giant tree. The idea is to find the largest branches of the tree, which will usually point to a memory leak situation (in Java, you leak memory not when you forget to delete an object, but when you forget to forget the object, i.e. you keep a reference to it somewhere).

### **Step 1. Enable heap dumps at run time**

Run your process with -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp

(It is safe to have these options always enabled. Adjust the path as needed, it must be writable by the java user)

### **Step 2. Reproduce the error**

Let the application run until the OutOfMemoryError occurs.

The JVM will automatically write a file like java\_pid12345.hprof.

### **Step 3. Fetch the dump**

Copy java\_pid12345.hprof to your PC (it will be at least as big as your maximum heap size, so can get quite big - gzip it if necessary).

### **Step 4. Open the dump file with IBM's**[**Heap Analyzer**](https://www.ibm.com/support/pages/ibm-heapanalyzer)**or Eclipse's**[**Memory Analyzer**](https://www.eclipse.org/mat/)

The Heap Analyzer will present you with a tree of all objects that were alive at the time of the error. Chances are it will point you **directly** at the problem when it opens.

### **Step 5. Identify areas of largest heap use**

Browse through the tree of objects and identify objects that are kept around unnecessarily.

Note it can also happen that all of the objects are necessary, which would mean you need a larger heap. Size and [tune](https://docs.oracle.com/javase/10/gctuning/factors-affecting-garbage-collection-performance.htm#JSGCT-GUID-5508674B-F32D-4B02-9002-D0D8C7CDDC75) the heap [appropriately](https://backstage.forgerock.com/knowledge/kb/article/a35746010).

### **Step 6. Fix your code**

Make sure to only keep objects around that you actually need. Remove items from collections in a timely manner. Make sure to not keep references to objects that are no longer needed, only then can they be garbage-collected.

**XX:+HeapDumpOnOutOfMemoryError**.

**OutOfMemoryError** usually means that you’re doing something wrong, either holding onto objects too long or trying to process too much data at a time. Sometimes, it indicates a problem that’s out of your control, such as a third-party library that caches strings or an application server that doesn’t clean up after deploys. And sometimes, it has nothing to do with objects on the heap.

**Error 1 – Java heap space:**

This error arises due to the applications that make excessive use of finalizers. If a class has a finalize method, objects of that type do not have their space reclaimed at [garbage collection](https://www.geeksforgeeks.org/garbage-collection-java/) time. Instead, after garbage collection, the objects are queued for finalization, which occurs later.

**Error 2 – GC Overhead limit exceeded:**

This error indicates that the garbage collector is running all the time and Java program is making very slow progress. After a garbage collection, if the Java process is spending more than approximately 98% of its time doing garbage collection and if it is recovering less than 2% of the heap and has been doing so far the last 5 (compile-time constant) consecutive garbage collections, then a **java.lang.OutOfMemoryError** is thrown.

This exception is typically thrown because the **amount of live data barely fits into the Java heap** having little free space for new allocations.

**Error 3 – Permgen space is thrown:**

Java memory is separated into different regions. The size of all those regions, including the permgen area, is set during the JVM launch. If you do not set the sizes yourself, platform-specific defaults will be used.

The **java.lang.OutOfMemoryError**: PermGen space error indicates that the Permanent Generation’s area in memory is exhausted.

**Error 4 – Metaspace:**

Java class metadata is allocated in native memory. Suppose metaspace for class metadata is exhausted, a **java.lang.OutOfMemoryError** exception with a detail MetaSpace is thrown.

The amount of metaspace used for class metadata is limited by the parameter MaxMetaSpaceSize, which is specified on the command line. When the amount of native memory needed for a class metadata exceeds MaxMetaSpaceSize, a java.lang.OutOfMemoryError exception with a detail MetaSpace is thrown.

**Error 5 – Requested array size exceeds VM limit:**

This error indicates that the application attempted to allocate an array that is larger than the heap size. For example, if an application attempts to allocate an array of 1024 MB but the maximum heap size is 512 MB then **OutOfMemoryError** will be thrown with “Requested array size exceeds VM limit”.

**Error 6 – Request size bytes for a reason. Out of swap space?:**

This apparent exception occurred when an allocation from the native heap failed and the native heap might be close to exhaustion. The error indicates the size (in bytes) of the request that failed and the reason for the memory request. Usually, the reason is the name of the source module reporting the allocation failure, although sometimes it is the actual reason.

The **java.lang.OutOfMemoryError: Out of swap space** error is often caused by operating-system-level issues, such as:

* The operating system is configured with insufficient swap space.
* Another process on the system is consuming all memory resources.

**Error 7 – reason stack\_trace\_with\_native\_method:**

Whenever this error message(reason stack\_trace\_with\_native\_method) is thrown then a stack trace is printed in which the top frame is a native method, then this is an indication that a native method has encountered an allocation failure. The difference between this and the previous message is that the allocation failure was detected in a Java Native Interface (JNI) or native method rather than the JVM code.

Paging and Sorting

Page<Entity> result = repository.findAll(PageRequest.of(pageNo, pageSize, Sort.by(sortBy));

result.getContent();

**Externalizable interface**

Externalization serves the purpose of custom Serialization, where we can decide what to store in stream.  
Externalizable interface present in java.io, is used for Externalization which extends Serializable interface. It consist of two methods.

**Key differences between Serializable and Externalizable** 

* **Implementation :** Unlike [Serializable interface](https://www.geeksforgeeks.org/serialization-in-java/)which will serialize the variables in object with just by implementing interface, here we have to explicitly mention what fields or variables you want to serialize.
* **Methods :** Serializable is marker interface without any methods. Externalizable interface contains two methods: writeExternal() and readExternal().
* **Process:** Default Serialization process will take place for classes implementing Serializable interface. Programmer defined Serialization process for classes implementing Externalizable interface.
* **Backward Compatibility and Control:** If you have to support multiple versions, you can have full control with Externalizable interface. You can support different versions of your object. If you implement Externalizable, it’s your responsibility to serialize super class.
* **public No-arg constructor:**Serializable uses reflection to construct object and does not require no arg constructor. But Externalizable requires public no-arg constructor.

**Closeble interface**

A**Closeable** is a source or destination of the data that needs to be closed. The **close()** method is invoked when we need to release resources that are being held by objects such as open files. It is one of the important interfaces to stream classes.

Since Closeable inherits the properties of AutoCloseable interface, therefore the class implementing Closeable can also use try-with-resources block. Multiple resources can be used inside a try-with-resources block and have them all automatically closed. In this case, the resources will be closed in the reverse order in which they were created inside the brackets.

Levels of cache in Hibernate

**There are mainly two types of caching:**

* First level cache
* Second-level cache

**1). First level cache**

The First level cache is ***by default enabled*** by Hibernate itself. The session object maintains the first-level cache.

An application can have many sessions. Data hold by one session object is not accessible to the entire application — means the data of a particular session is not shared with other sessions of the application. So you can use the first-level cache to store local data i.e. required by the session itself.

So when you query an entity or object, for the very first time it is retrieved from the database and stored into the first-level cache (associated with the hibernate session). If we query for the same entity or object again with the same session object, it will be loaded from cache and no SQL query will be executed.

## 2) Second level cache

The second-level cache is **by default disabled,**the developer needs to enable it explicitly, and the SessionFactory object is responsible to maintain it. The second-level cache is accessible by the entire application means data hold by SessionFactory can be accessible to all the sessions. Keep in mind that, once the session factory is closed all the cache associated with that is also removed from the memory.

Let’s take an example: Suppose your application has 2 active sessions session1 and session2 respectively. Now, session1 has requested data having id=101 so that will be fetched from a database as it is the first call, and then it is stored into the second-level (SessionFactory) as well as in the first-level (session) cache also. Now, session2 requires the same data so it has also been queried with the same id=101. So this time session2 will get data from the SessionFactory, it will not going to hit the database. Take a look at the below code snippet.

### Sequential Stream

Sequential Streams are non-parallel streams that use a single thread to process the pipelining. Any stream operation without explicitly specified as parallel is treated as a sequential stream. Sequential stream’s objects are pipelined in a single stream on the same processing system hence it never takes the advantage of the multi-core system even though the underlying system supports parallel execution. Sequential stream performs operation one by one.

### Parallel Stream

It is a very useful feature of Java to use parallel processing, even if the whole program may not be parallelized. Parallel stream leverage multi-core processors, which increases its performance. Using parallel streams, our code gets divide into multiple streams which can be executed parallelly on separate cores of the system and the final result is shown as the combination of all the individual core’s outcomes. It is always not necessary that the whole program be parallelized, but at least some parts should be parallelized which handles the stream. The order of execution is not under our control and can give us unpredictably unordered results and like any other parallel programming, they are complex and error-prone.

ConcurrentSkipListMap

The most important thing to note is that **we can take the snapshot of data while other threads are still adding new values** to the ConcurrentSkipListMap.

ConcurrentHashMap guarantees us data coherency in a multi-threaded environment.

**Interceptors in Spring boot**

You can use the Interceptor in Spring Boot to perform operations under the following situations −

* Before sending the request to the controller
* Before sending the response to the client

For example, you can use an interceptor to add the request header before sending the request to the controller and add the response header before sending the response to the client.

To work with interceptor, you need to create **@Component** class that supports it and it should implement the **HandlerInterceptor** interface.

The following are the three methods you should know about while working on Interceptors −

* **preHandle()** method − This is used to perform operations before sending the request to the controller. This method should return true to return the response to the client.
* **postHandle()** method − This is used to perform operations before sending the response to the client.
* **afterCompletion()** method − This is used to perform operations after completing the request and response.

## Filters

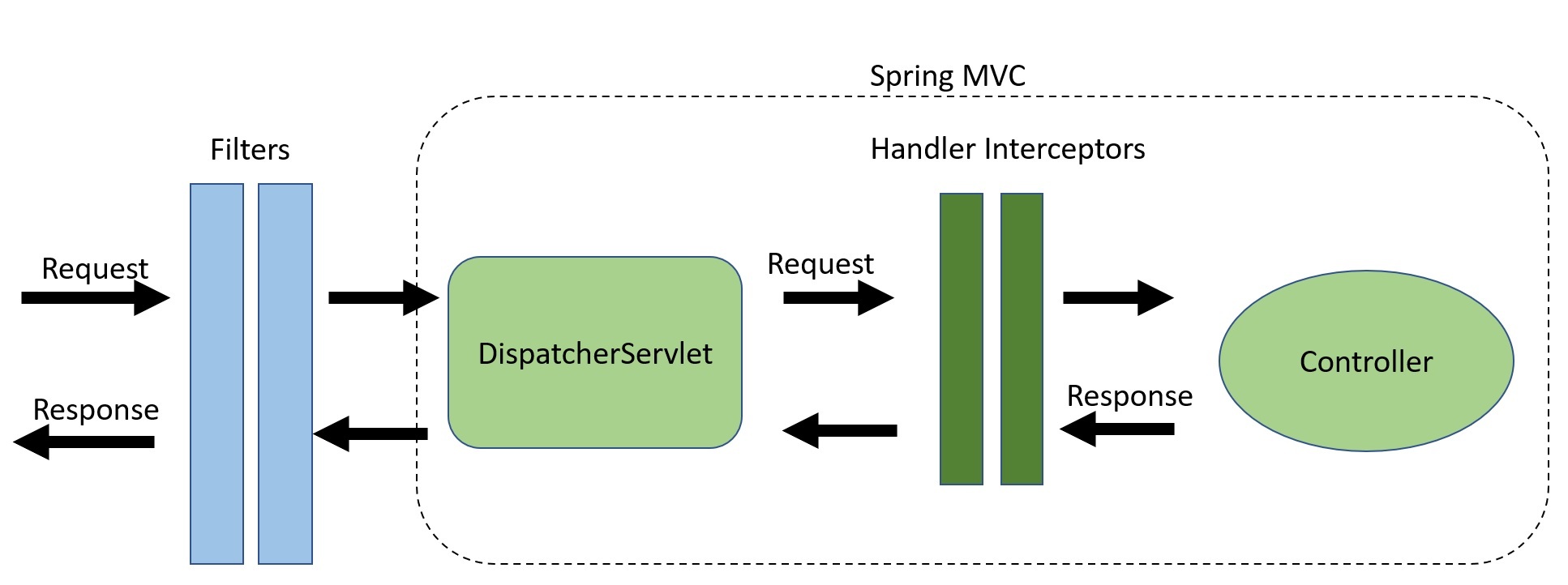
**Filters are part of the webserver and not the Spring framework.** For incoming requests, **we can use filters to manipulate and even block requests from reaching any**[**servlet**](https://www.baeldung.com/java-servlets-containers-intro). Vice versa, we can also block responses from reaching the client.

[Spring Security](https://www.baeldung.com/security-spring) is a great example of using filters for authentication and authorization. To configure Spring Security, we simply need to add a single filter, the [DelegatingFilterProxy](https://www.baeldung.com/spring-delegating-filter-proxy). Spring Security can then intercept all incoming and outgoing traffic. This is why Spring Security can be used outside of [Spring MVC](https://www.baeldung.com/spring-mvc).

To create a filter, first, we create a class that implements the [*javax.servlet.Filter*](https://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html) interface.

**Key Differences and Use Cases**

Let's look at a diagram showing where *Filter*s and *HandlerInterceptor*s fit in the request/response flow:

[](https://www.baeldung.com/wp-content/uploads/2021/05/filters_vs_interceptors.jpg)

**Filters intercept requests before they reach the *DispatcherServlet*, making them ideal for coarse-grained tasks** such as:

* Authentication
* Logging and auditing
* Image and data compression
* Any functionality we want to be decoupled from Spring MVC

***HandlerIntercepor*s, on the other hand, intercepts requests between the *DispatcherServlet* and our *Controller*s.** This is done within the Spring MVC framework, providing access to the *Handler* and *ModelAndView* objects. This reduces duplication and allows for more fine-grained functionality such as:

* Handling cross-cutting concerns such as application logging
* Detailed authorization checks
* Manipulating the Spring context or model

Hashing

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Why SLF4J is used?

This is the main purpose of SLF4J (Simple Logging Facade for Java) – **a logging abstraction which helps to decouple your application from the underlying logger by allowing it to be plugged in – at runtime.**

One of the nice features that provide SLF4J is the possibility to generate parametrized log messages easily and in a performant manner.

It’s fairly similar to the common String.format approach, but it differs in a critical aspect. **While string formatting or concatenation will happen whether the message needs to be logged or not, when you use SLF4J’s parametrized methods you’ll avoid incurring in the cost of parameter construction in case the log statement is disabled.**

### **SLF4J Supports Mapped Diagnostic Context Mechanisms**

Any server application you develop will typically attend several clients using a pool of threads. Each thread will be dedicated to one client, and will, therefore, have a specific context.

**The**[**Mapped Diagnostic Context**](https://www.baeldung.com/mdc-in-log4j-2-logback)**, or MDC for short, is simply a map managed by the logging framework on a per-thread basis. This map will contain relevant information that might be useful when logging messages.**

# [**Function vs. Stored Procedure**](https://stackoverflow.com/questions/1179758/function-vs-stored-procedure-in-sql-server)

| **Functions** | **Procedures** |
| --- | --- |
| A function has a return type and returns a value. | A procedure does not have a return type. But it returns values using the OUT parameters. |
| You cannot use a function with Data Manipulation queries. Only Select queries are allowed in functions. | You can use DML queries such as insert, update, select etc… with procedures. |
| A function does not allow output parameters | A procedure allows both input and output parameters. |
| You cannot manage transactions inside a function. | You can manage transactions inside a procedure. |
| You cannot call stored procedures from a function | You can call a function from a stored procedure. |
| You can call a function using a select statement. | You cannot call a procedure using select statements. |

| DROP | TRUNCATE |
| --- | --- |
| 1. | The DROP command is used to remove table definition and its contents. | Whereas the TRUNCATE command is used to delete all the rows from the table. |
| 2. | In the DROP command, table space is freed from memory. | While the TRUNCATE command does not free the table space from memory. |
| 3. | DROP is a DDL(Data Definition Language) command. | Whereas the TRUNCATE is also a DDL(Data Definition Language) command. |
| 4. | In the DROP command, view of table does not exist. | While in this command, view of table exist. |
| 5. | In the DROP command, integrity constraints will be removed. | While in this command, integrity constraints will not be removed. |
| 6. | In the DROP command, undo space is not used. | While in this command, undo space is used but less than DELETE. |
| 7. | The DROP command is quick to perform but gives rise to complications. | While this command is faster than DROP. |

Partition By clause

A **PARTITION BY**clause is used to partition rows of table into groups. It is useful when we have to perform a calculation on individual rows of a group using other rows of that group.

* It is always used inside OVER() clause.
* The partition formed by partition clause are also known as **Window**.
* This clause works on windows functions only. Like- RANK(), LEAD(), LAG() etc.
* If this clause is omitted in OVER() clause, then whole table is considered as a single partition.

### @RequestParam to get Query parameters

### PathVariable annotation to extract values from URI

Spring security is implemented in filter chains before dispatcher servlet

Request comes to Delegating filter proxy, then to filter chain proxy, then to chain of filters

Principal is an object which represents logged in user or device or app. Once user is authenticated, principle is created. Principle can have many authorities. Using these 2, object is populated called authentication, then it is populated to secutory context, then it will be populated to HTT session

### Synchronous programming

[**Synchronous**](https://whatis.techtarget.com/definition/synchronous) is known as a blocking architecture and is ideal for programming reactive systems. As a single-thread model, it follows a strict set of sequences, which means that operations are performed one at a time, in perfect order. While one operation is being performed, other operations’ instructions are blocked. The completion of the first task triggers the next, and so on.

### Asynchronous programming

[**Asynchronous programming**](https://www.techtarget.com/searchnetworking/definition/asynchronous), conversely, is a multithreaded model that’s most applicable to networking and communications. Asynchronous is a non-blocking architecture, which means it doesn’t block further execution while one or more operations are in progress.

With asynchronous programming, multiple related operations can run concurrently without waiting for other tasks to complete. During asynchronous communication, parties receive and process messages when it’s convenient or possible to do so, rather than responding immediately upon receipt.

**ActiveMQ**

Configure ActiveMQ, connectionFactory and JMSTemplate.

Autowire the queue and template in the producer

Use JmsListner in Consumer component.

Zookeeper

Zokeeper is used to track cluster state, membership and leadership.

Zookeeper is used for metadata management in the kafka world.

Zookeeper keeps track of which brokers are part of the Kafka cluster

Zookeeper is used by kafka brokers to determine which broker is the leader of a given partition and topic and perform leader elections.

Zookeeper keeps configurations for topics and permissions

Zookeeper sends notifications to kafka in case of changes.

**Kafka**

Kafka allows us to decouple data streams and systems. With kafka as adata integration layer, data sources will publish their data to kafka and the targer system will source their data from the kafka.

Topics

Topics organize related events. It is not queryable. Instead, we must create producers and consumers to utilize data.

Producer

to send data to the topic.

Consumers

To take data from kafka

Consumer groups

Consumers that are part of same application and therefore performing the same job

Partitions

Topics are broken down into number of partitions

Offset

Position of the message within a partition

Message Serializers

The process of transforming the producers programmatic representation of the object to binary

Schema registry

It helps register data schemes in kafka and ensure that producers and consumers will be compatible with each other.

c3p0

**c3p0 is** **a Java library that provides a convenient way for managing database connections**.

In short, it achieves this by creating a pool of connections. It also effectively handles the cleanup of *Statement*s and *ResultSet*s after use. This cleanup is necessary to ensure that resource usage is optimized and avoidable deadlocks do not occur.

**This library integrates seamlessly with various traditional JDBC drivers.** Additionally, it provides a layer for adapting DriverManager-based JDBC drivers to the newer *javax.sql.DataSource*scheme.

Kibana

Kibana is used for searching, viewing and visualizing data indexed in elasticsearch and analysing data through creation of bar charts, pie charts etc.

Logging and analytics

Infrastructure metrics and container monitoring

Application performance monitoring

How to resolve merge conflicts

We will get merge conflicts when we try to make code changes to same line in a file where another person already made some changes.

In this case we need to pull the new changes to our local and coordinate with the other developer in terms where the changes should be

Docker

Docker packages software into standardized units called [containers](https://aws.amazon.com/containers/) that have everything the software needs to run including libraries, system tools, code, and runtime. Using Docker, you can quickly deploy and scale applications into any environment and know your code will run.

Docker is an operating system for containers. Similar to how a [virtual machine](https://aws.amazon.com/ec2/) virtualizes (removes the need to directly manage) server hardware, containers virtualize the operating system of a server. Docker is installed on each server and provides simple commands you can use to build, start, or stop containers.

Kubernetes

Kubernetes (also known as k8s or “kube”) is an [open source](https://www.redhat.com/en/topics/open-source/what-is-open-source) container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications.

## Continuous Delivery Explained

Continuous delivery is a software development practice where code changes are automatically prepared for a release to production. A pillar of [modern application development](https://aws.amazon.com/modern-apps/), continuous delivery expands upon [continuous integration](https://aws.amazon.com/devops/continuous-integration/) by deploying all code changes to a testing environment and/or a production environment after the build stage. When properly implemented, developers will always have a deployment-ready build artifact that has passed through a standardized test process.

Continuous delivery lets developers automate testing beyond just unit tests so they can verify application updates across multiple dimensions before deploying to customers. These tests may include UI testing, load testing, integration testing, API reliability testing, etc. This helps developers more thoroughly validate updates and pre-emptively discover issues. With the cloud, it is easy and cost-effective to automate the creation and replication of multiple environments for testing, which was previously difficult to do on-premises.

## Continuous Delivery vs. Continuous Deployment

With continuous delivery, every code change is built, tested, and then pushed to a non-production testing or staging environment. There can be multiple, parallel test stages before a production deployment. The difference between continuous delivery and continuous deployment is the presence of a manual approval to update to production. With continuous deployment, production happens automatically without explicit approval.

Kubernetes

Node

# Nodes

Kubernetes runs your workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) and contains the services necessary to run [Pods](https://kubernetes.io/docs/concepts/workloads/pods/).

Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have only one node.

The [components](https://kubernetes.io/docs/concepts/overview/components/#node-components) on a node include the [kubelet](https://kubernetes.io/docs/reference/generated/kubelet), a [container runtime](https://kubernetes.io/docs/setup/production-environment/container-runtimes), and the [kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/).

# Pods

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more [containers](https://kubernetes.io/docs/concepts/containers/), with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

# ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

# NodePort

A NodePort service is the most primitive way to get external traffic directly to your service. NodePort, as the name implies, opens a specific port on all the Nodes (the VMs), and any traffic that is sent to this port is forwarded to the service.

Parallel Streams

Java Parallel Streams is a feature of Java 8 and higher, meant for utilizing multiple cores of the processor. Normally any java code has one stream of processing, where it is executed sequentially. Whereas by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes. The order of execution, however, is not under our control.

### Why Parallel Streams?

Parallel Streams were introduced to increase the performance of a program, but opting for parallel streams isn’t always the best choice. There are certain instances in which we need the code to be executed in a certain order and in these cases, we better use sequential streams to perform our task at the cost of performance. The performance difference between the two kinds of streams is only of concern for large-scale programs or complex projects. For small-scale programs, it may not even be noticeable. Basically, you should consider using Parallel Streams when the sequential stream behaves poorly.

Factory design pattern in spring

The bean initialization in Spring is based on Factory Design Pattern. The BeanFactory ([org.springframework.beans.factory.BeanFactory](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html)) interface with bunch of abstract methods allow us to initialize different beans providing fundament for Dependency Injection. Basically each getBean() method is an implementation of Factory Design Pattern. Although BeanFactory is basics for bean initialization, more modern ,method is using ApplicationContext. On the other hand, BeanFactory supports only xml-based configuration. Since xml-based configuration is a bit outdated, we will take a look at more modern approaches.

Spring uses this technique at the root of its [Dependency Injection (DI) framework](https://www.baeldung.com/spring-dependency-injection).

Fundamentally, **Spring treats** **a bean container as a factory that produces beans.**

Thus, Spring defines the BeanFactory interface as an abstraction of a bean container:

**Each of the *getBean* methods is considered a factory method**, which returns a bean matching the criteria supplied to the method, like the bean's type and name.

Spring then extends *BeanFactory* with the *ApplicationContext* interface, which introduces additional application configuration. Spring uses this configuration to start-up a bean container based on some external configuration, such as an XML file or Java annotations.

Using the *ApplicationContext* class implementations like *AnnotationConfigApplicationContext*, we can then create beans through the various factory methods inherited from the *BeanFactory* interface.

**Applications of Singleton classes**

There is a lot of applications of singleton pattern like cache-memory, database connection, drivers, logging. Some major of them are :-

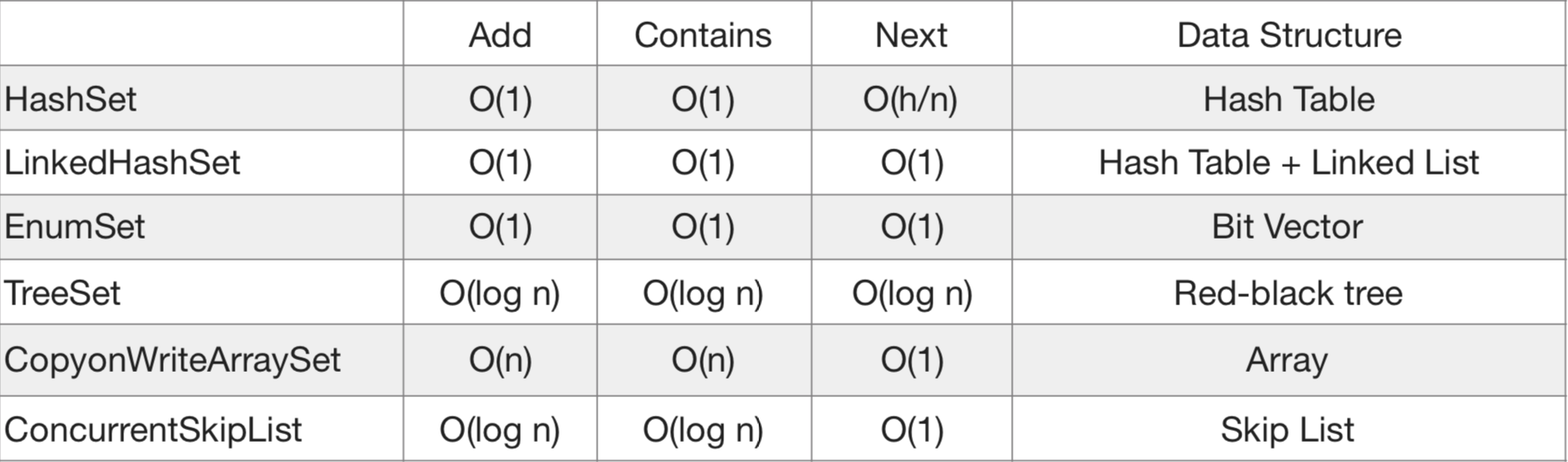
1. **Hardware interface access:** The use of singleton depends on the requirements. Singleton classes are also used to prevent concurrent access of class. Practically singleton can be used in case external hardware resource usage limitation required e.g. Hardware printers where the print spooler can be made a singleton to avoid multiple concurrent accesses and creating deadlock.
2. **Logger :** Singleton classes are used in log file generations. Log files are created by the logger class object. Suppose an application where the logging utility has to produce one log file based on the messages received from the users. If there is multiple client application using this logging utility class they might create multiple instances of this class and it can potentially cause issues during concurrent access to the same logger file. We can use the logger utility class as a singleton and provide a global point of reference so that each user can use this utility and no 2 users access it at the same time.
3. **Configuration File:** This is another potential candidate for Singleton pattern because this has a performance benefit as it prevents multiple users to repeatedly access and read the configuration file or properties file. It creates a single instance of the configuration file which can be accessed by multiple calls concurrently as it will provide static config data loaded into in-memory objects. The application only reads from the configuration file for the first time and thereafter from second call onwards the client applications read the data from in-memory objects.
4. **Cache:** We can use the cache as a singleton object as it can have a global point of reference and for all future calls to the cache object the client application will use the in-memory object.

# **Java Collections**

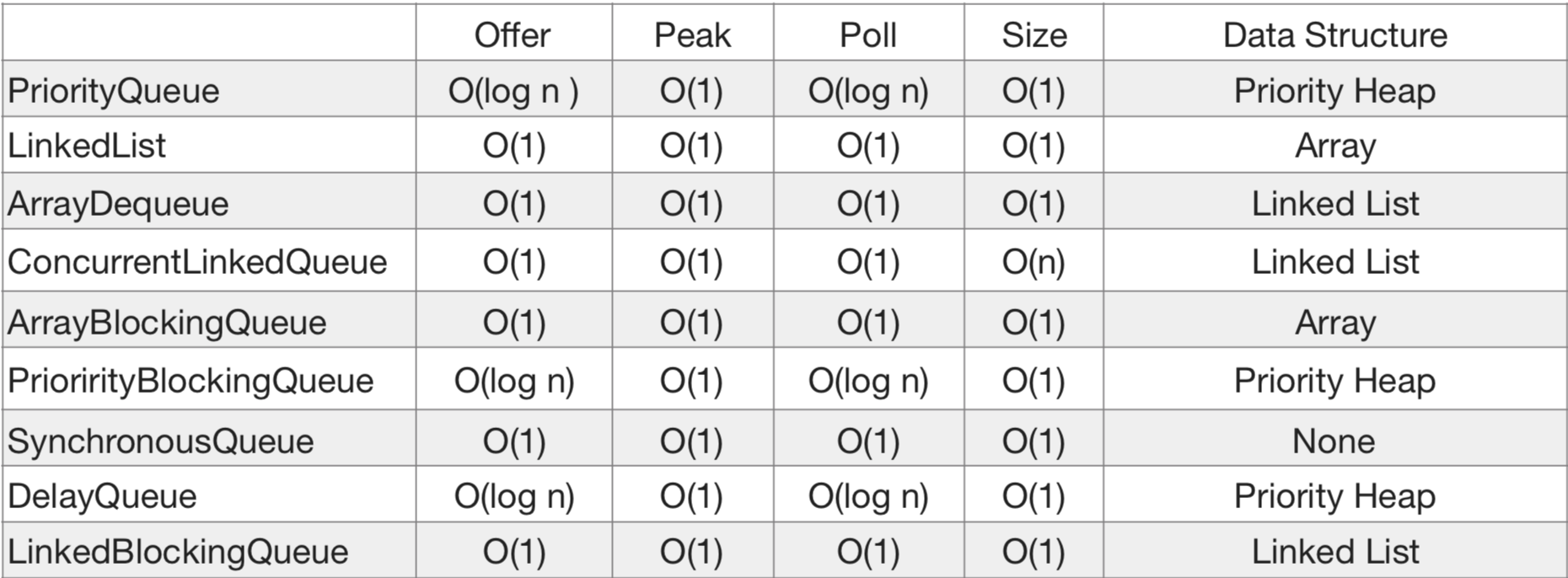
List: A list is an ordered collection of elements.



Set: A collection that contains no duplicate elements.



Queue: A collection designed for holding elements prior to processing.



Map: An object that maps keys to values.  
A map cannot duplicate keys; each key can map to at most one value



Swagger

Swagger is the standard way of documenting the Standard APIs. Swagger is helpful when deploying APIs in azure. Swagger is primarily used for documenting API; now the question arises that why document APIs?. The building APIs that are internal in the enterprise or for the public consumption, the theme is the same that the developers usually use in the apps that they are building. For the other developers to be able to use our API, the API must be properly documented; otherwise, how would they know that what are the endpoints exposed by the api and what are the operations supported on those endpoints? What parameters should they pass, and what will they get back? What authentication methods to use?. To answer these questions, it is very important to document the APIs; if you want APIs to be consumed and properly used.

SonarQube

SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs and code smells on 29 programming languages.

What happens when multiple instances of a service access the database at the same time?

If one thread performs a database call, calls made by other threads will be blocked until the first call completes, even if the subsequent calls access database objects that are unrelated to the first call. In addition, all threads within a process share a commit scope. True concurrent access to a database can only be achieved through separate processes, or by using the APIs that are described in this topic.

Why Static is used in Java?

The **static keyword** in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes. The static keyword belongs to the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.

Basically, static is used for a constant variable or a method that is same for every instance of a class.

## What Does the Static Keyword Do?

Every Java programmer knows that if we need to define some behavior (method) or state (field) that will be common to all objects, we define it as static. Because static content (behavior or state) does not belong to any particular instance or object, it will be common to all objects and all objects are free to change any static field — and every change will be visible to every object.

**Why an Outer Java Class Can’t Be Static**

From above, we can conclude that we should define members as static that should:

1. Be common to all objects of the class.
2. Belong to the class and accessible by class name.
3. Not need an object of class to access them.

Now, suppose we are defining an outer class as static, and suppose we are allowed to do so. Will this serve any purpose or provide any advantage to a developer, or it will create ambiguity and complications for both developers and language creators?

Let’s check: Will defining an outer class as static serve the purposes that we have defined above or not?

1. Every class is already common to all of its objects, and there is no need to make it static to become available to all of its objects.
2. We need a class name to access its static members because these members are part of a class while an outer class is part of a package. We can directly access the class by just writing package\_name.class\_name (similar to class\_name.static\_field\_name). So again, there is no need to do something that's already there by default.
3. We do not need any object to access a class if it is visible. We can simply write package\_name.class\_name to access it. And by definition, a class is a blueprint for its objects, and we create a class to create objects from it (though an exception will always be there, like java.lang.Math). Again, there is no need to define an outer class as static.

Why final is used in Java?

The final keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override). The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...). The final keyword is called a "modifier".

The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

The final keyword is called a "modifier".

| **Final Access Modifier** | **Static Access Modifier** |
| --- | --- |
| This modifier is applicable to both outer and inner classes, variables, methods, and blocks. | This modifier is only applicable to inner classes, methods, and variables. |
| It is not necessary to initialize the final variable at the time of its declaration. | It is necessary to initialize the static variable at the time of its declaration. |
| Final variable cannot be reinitialized. | Static variables can be reinitialized. |
| Final method can’t be inherited. | Static methods can only access the static members of the class and can only be called by other static methods. |
| Final class can’t be inherited by any class. | The static class object can’t be created and it only contains static members only. |
| Final keyword doesn’t support any block for initialization of final variables. | Static block is used to initialize the static variables. |
| Final local variables are allowed. | Unlike C/C++, static local variables are not allowed in Java. |

**Compile-time constants**

A Java variable is a compile-time constant if it's **of a primitive type or String, declared final, initialized within its declaration, and with a constant expression**.

Why objects are stored in Heap?

| **Parameter** | **Stack Memory** | **Heap Space** |
| --- | --- | --- |
| Application | Stack is used in parts, one at a time during execution of a thread | The entire application uses Heap space during runtime |
| Size | Stack has size limits depending upon OS, and is usually smaller than Heap | There is no size limit on Heap |
| Storage | Stores only primitive variables and references to objects that are created in Heap Space | All the newly created objects are stored here |
| Order | It's accessed using Last-in First-out (LIFO) memory allocation system | This memory is accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation. |
| Life | Stack memory only exists as long as the current method is running | Heap space exists as long as the application runs |
| Efficiency | Much faster to allocate when compared to heap | Slower to allocate when compared to stack |
| Allocation/Deallocation | This Memory is automatically allocated and deallocated when a method is called and returned, respectively | Heap space is allocated when new objects are created and deallocated by Gargabe Collector when they're no longer referenced |

Copy and clone

Copy – will use the same reference

Clone – will have different reference

Benefits of Java

One of the most significant advantages of Java is its ability to move easily from one computer system to another. The ability to run the same program on many different systems is crucial to World Wide Web software, and Java succeeds at this by being platform-independent at both the source and binary levels.

Java is a secured programming language because it doesn't use Explicit pointers. Also, Java programs run inside the virtual machine sandbox. JRE also provides a classloader, which is used to load the class into JVM dynamically. It separates the class packages of the local file system from the ones that are being imported from the network.

Java is a robust programming language since it uses strong memory management. We can also handle exceptions through the Java code. Also, we can use type checking to make our code more secure. It doesn't provide explicit pointers so that the programmer cannot access the memory directly from the code.

Benefits of Object-oriented-programming

## Modularity for easier troubleshooting

## Reuse of code through inheritance

## Flexibility through polymorphism

## Limitation of OOP

**1. Performance**

Java needs to be interpreted during runtime, which allows it to run on every operating system, but it also makes it perform slower than the languages like [C](https://www.javatpoint.com/c-programming-language-tutorial) and [C++](https://www.javatpoint.com/cpp-tutorial). On the other hand, the C++ program needs to be compiled on each operating system, directly to binary and therefore runs faster.

|  |
| --- |
| **Advantages and disadvantages of Java** Java is a general-purpose, robust, secure, and object-oriented programming language. It is a high-level language, I.e., its syntax uses English like language. It was developed by Sun Microsystems in the year 1995. It is now maintained and distributed by Oracle. Java has its runtime environment and API; therefore, it is also called a platform.  [Java](https://www.javatpoint.com/java-tutorial) is used in a large number of applications over the years. However, it has various advantages and disadvantages given below. **Advantages:** **1. Simple**  Java is a simple programming language since it is easy to learn and easy to understand. Its syntax is based on C++, and it uses automatic garbage collection; therefore, we don't need to remove the unreferenced objects from memory. Java has also removed the features like explicit pointers, operator overloading, etc., making it easy to read and write.  **2. Object-Oriented**  Java uses an object-oriented paradigm, which makes it more practical. Everything in Java is an object which takes care of both data and behavior. Java uses [object-oriented concepts](https://www.javatpoint.com/java-oops-concepts) like [object](https://www.javatpoint.com/object-and-class-in-java#object), [class](https://www.javatpoint.com/object-and-class-in-java#class), [inheritance](https://www.javatpoint.com/inheritance-in-java), [encapsulation](https://www.javatpoint.com/encapsulation), [polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java), and abstraction.  **3. Secured**  Java is a secured programming language because it doesn't use Explicit pointers. Also, Java programs run inside the virtual machine sandbox. [JRE](https://www.javatpoint.com/java-jre) also provides a [classloader](https://www.javatpoint.com/classloader-in-java), which is used to load the class into [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) dynamically. It separates the class packages of the local file system from the ones that are being imported from the network.  **4. Robust**  Java is a robust programming language since it uses strong memory management. We can also handle exceptions through the Java code. Also, we can use type checking to make our code more secure. It doesn't provide explicit pointers so that the programmer cannot access the memory directly from the code.  **5. Platform independent**  Java code can run on multiple platforms directly, I.e., we need not compile it every time. It is right once, runs anywhere language (WORA) which can be converted into byte code at the compile time. The byte code is a platform-independent code that can run on multiple platforms.  **6. Multi-Threaded**  Java uses a multi-threaded environment in which a bigger task can be converted into various threads and run separately. The main advantage of multi-threading is that we need not provide memory to every running thread. **Disadvantages** **1. Performance**  Java needs to be interpreted during runtime, which allows it to run on every operating system, but it also makes it perform slower than the languages like [C](https://www.javatpoint.com/c-programming-language-tutorial) and [C++](https://www.javatpoint.com/cpp-tutorial). On the other hand, the C++ program needs to be compiled on each operating system, directly to binary and therefore runs faster.  **2. Memory consumption** |

## Java program consumes more memory since it runs on top of Java virtual machine.

**3. Cost**

Java programming language is a bit costly due to its higher processing and memory requirements. We need better hardware to run the Java program.

**4. Less machine interactive**

Java lacks when it comes to interacting directly with machines, making it less viable for the software that needs to run quickly and run directly with the machine, as explicit pointers are also missing in Java.

**5. Garbage collection**

Java provides automatic garbage collection that cannot be controlled by the programmer. It doesn't provide the methods like delete() and free() to free the memory.

However, due to the various disadvantages, Java is one of the most used programming languages due to its advantages, making it platform-independent, secure and a robust programming language.

## Spring boot admin

## Spring Boot Admin is a web application, used for managing and monitoring Spring Boot applications. Each application is considered as a client and registers to the admin server. Behind the scenes, the magic is given by the Spring Boot Actuator endpoints.

# Deploy Spring Boot Apps From Jar to War

**1. Update your maven pom.xml with the following changes**

Set your packaging tag to war

Set all Tomcat Jar files scope to provided

**Update your Main class to**extends SpringBootServletInitializer

Locking on database

Locking on external resource basically means locking on a network/socket connection. Because if some object is locked in the database (e.g. SELECT ... FOR UPDATE) and someone else than the lock's holder wants to obtain the lock, the database will not respond (write to client's socket) anything until the lock can be obtained.  
Therefore your app will wait for something to appear on the socket and the thread blocks (blocking way) or there will be no event fired until then (non-blocking way).  
Locking in the same process (in which all threads run) is easier. All those threads share the same memory space, therefore you can create a shared memory structure (such as critical section - LPCRITICAL\_SECTION in Windows, something similar in POSIX) and access it from all threads (in Windows, functions EnterCriticalSection and LeaveCriticalSection, which get this critical section structure as an argument)

This is much faster than locking on some network resource, because everything happens within a single process and the locking is done directly by the OS kernel.

Jenkins

Jenkins is an open source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat.

CI/CD

Continuous integration is a [DevOps](https://aws.amazon.com/devops/) software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration most often refers to the build or integration stage of the software release process and entails both an automation component (e.g. a CI or build service) and a cultural component (e.g. learning to integrate frequently). The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

How to resolve merge conflicts

We will get merge conflicts when we try to make code changes to same line in a file where another person already made some changes.

In this case we need to pull the new changes to our local and coordinate with the other developer in terms where the changes should be

**Sequences in SQL**

Sequence is a set of integers 1, 2, 3, … that are generated and supported by some database systems to produce unique values on demand.

* A sequence is a user defined schema bound object that generates a sequence of numeric values.
* Sequences are frequently used in many databases because many applications require each row in a table to contain a unique value and sequences provides an easy way to generate them.
* The sequence of numeric values is generated in an a**scending or descending order** at defined intervals and can be configured to restart when exceeds max\_value.

**SQL Triggers**

A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

How to bind model and view in spring?

In Spring, the Model-View-Controller (MVC) pattern is used to separate the concerns of the web application into different layers. The Model represents the data and business logic, the View represents the presentation layer, and the Controller acts as an intermediary between the Model and the View. The following steps outline how to bind the Model and View in Spring:

1. Define a Model class: Create a POJO (Plain Old Java Object) that represents the data and business logic of the application. The Model class should be annotated with the **@Component** annotation to indicate that it is a Spring-managed bean.
2. Define a View: Create a web page or other type of presentation layer that will be used to display the data. In Spring, the View is typically implemented using a templating engine such as Thymeleaf or JSP.
3. Define a Controller: Create a controller class that acts as an intermediary between the Model and the View. The Controller should be annotated with the **@Controller** annotation to indicate that it is a Spring-managed bean.
4. Inject the Model into the Controller: Use the **@Autowired** annotation to inject the Model into the Controller. This allows the Controller to access the data and business logic in the Model.
5. Populate the Model: In the Controller, populate the Model with the data that will be displayed in the View. This is typically done using a service class that interacts with a database or other data source.
6. Return the View: In the Controller, return the name of the View that will be used to display the data. Spring will then render the View and populate it with the data from the Model.