#### **Spring**

**Annotations:**

**Sterio Type Annotation??**

**@Autowired:** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on a method, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

**@Configuration:** It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a **source of bean definitions**.

**@ComponentScan:** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

**@Bean:** It is a ***method-level*** annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

**@Component:** It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with **@Componen**t is found during the classpath. The Spring Framework pick it up and configure it in the application context as a **Spring Bean**.

**@Controller:** The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a **web request handler**. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

**@Service:** It is also used at class level. It tells the Spring that class contains the **business logic**.

**@Repository:** It is a class-level annotation. The repository is a **DAOs** (Data Access Object) that access the database directly. The repository does all the operations related to the database.

**@SpringBootApplication:**Itis a combination of three annotations @EnableAutoConfiguration, @ComponentScan, and @Configuration.

**@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.

**@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return object into JSON and XML format.

**@RequestParam** annotation used for accessing the **query parameter** values from the request. Look at the following request URL: [*http://localhost:8080/springmvc/hello/101?param1=10&param2=20*](http://localhost:8080/springmvc/hello/101?param1=10&param2=20)

In the above URL request, the values for param1 and param2 can be accessed as below:

public String getDetails(

@RequestParam(value="param1", required=true) String param1,

@RequestParam(value="param2", required=false) String param2){..}

**@PathVariable**

@PathVariable identifies the pattern that is used in the URI for the incoming request. Let’s look at the below request URL:

[*http://localhost:8080/springmvc/hello/101?param1=10&param2=20*](http://localhost:8080/springmvc/hello/101?param1=10&param2=20)

The above URL request can be written in your Spring MVC as below:

@RequestMapping("/hello/{id}")

public String getDetails(@PathVariable(value="id") String id,

@RequestParam(value="param1", required=true) String param1,

@RequestParam(value="param2", required=false) String param2){

.......

}

The @**PathVariable** annotation has only one attribute value for binding the request URI template. It is allowed to use the multiple @**PathVariable** annotation in the single method. But, ensure that no more than one method has the same pattern.

**@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.**For each detail in the header, we should specify separate annotations. We can use it multiple time in a method

* **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
* **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.
* The PATCH HTTP method is used when you want to apply a **partial** update to the resource. Consider we want to update the Employee resources partially (only firstName field) in a database. So here is the REST API that demonstrates the usage of @PatchMapping annotation:

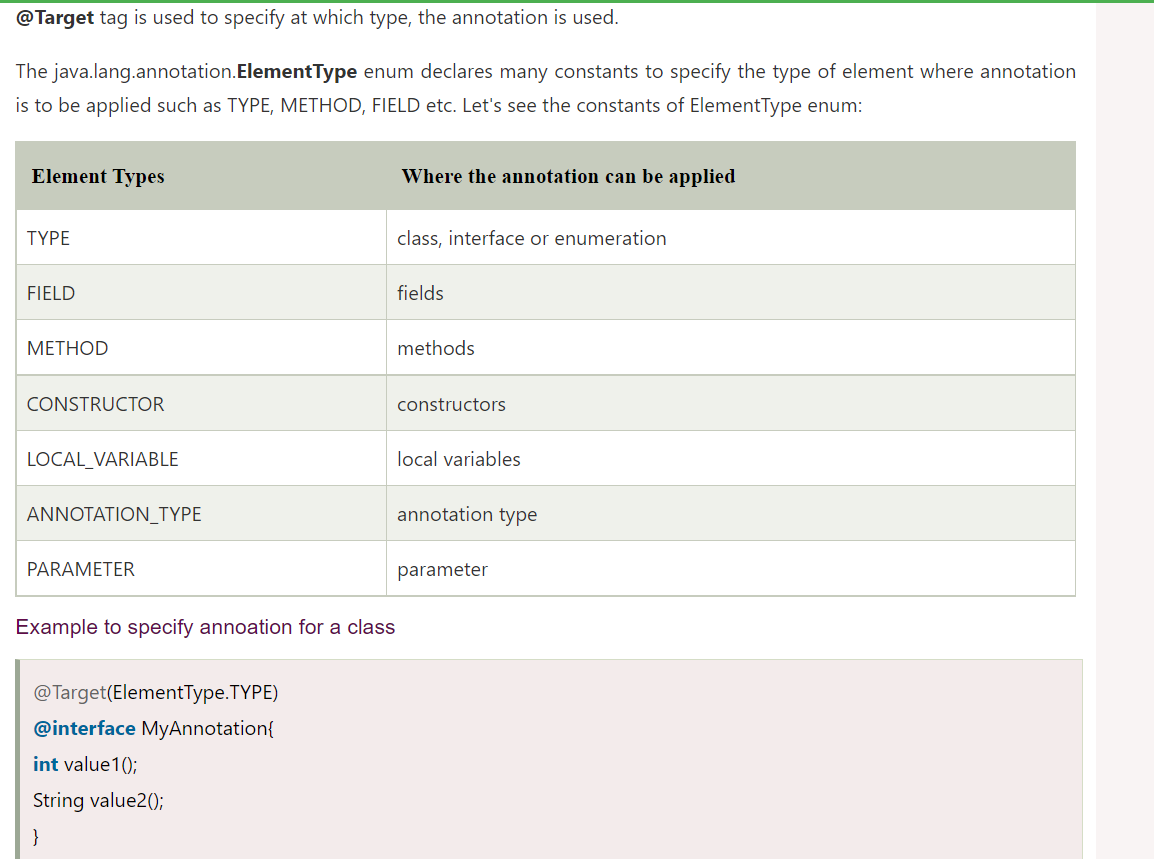
@PatchMapping("/employees/{id}/{firstName}")

public ResponseEntity<Employee> updateEmployeePartially(@PathVariable Long id, @PathVariable String firstName){……

}

* **@PutMapping**- Updation- @PutMapping("/employees/{id}")
* **@PostMapping** – Creation- @PostMapping("/employees")
* **@EnableDiscoveryClient** - @EnableDiscoveryClient on top of the Application.java class to treat this module as Eureka Client and
* **@Ignore** – It is used to skip particular testcase

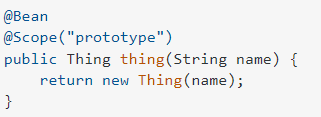
**Built-in Annotations used in custom annotations in java**

* @Target
* @Retention
* @Inherited
* @Documented
* 

**Spring Bean Scopes**

There are five types of [spring bean](https://www.journaldev.com/2461/spring-ioc-bean-example-tutorial) scopes:

1. **singleton** – only one instance of the spring bean will be created for the spring container. This is the **default** spring bean scope. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.
2. **prototype** – A new instance will be created every time the bean is requested from the spring container.



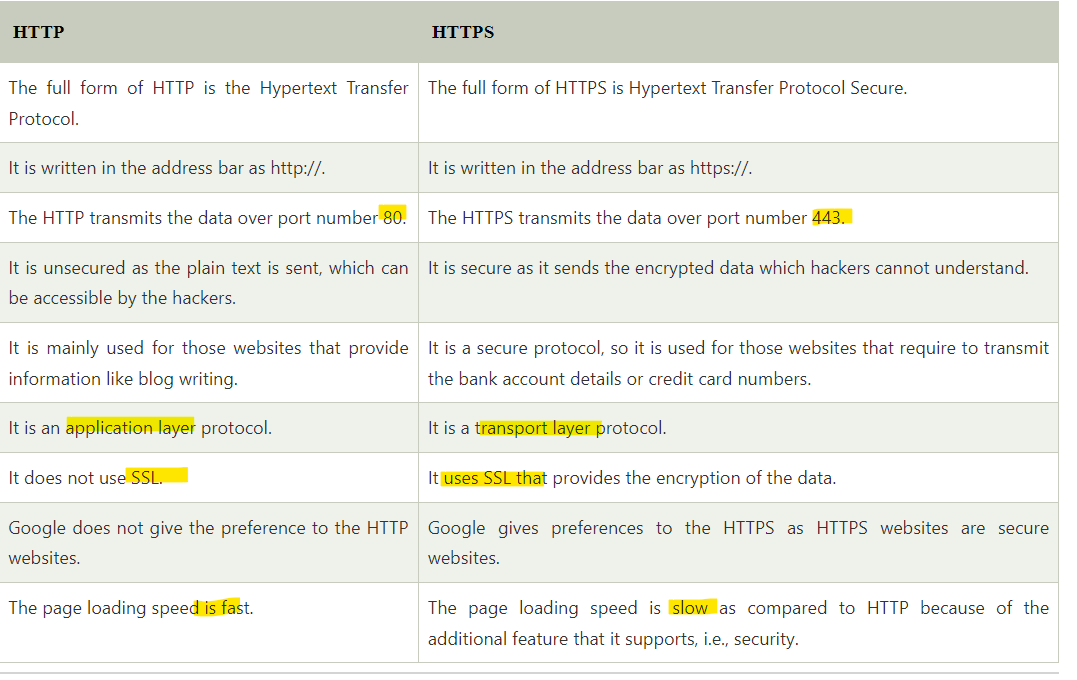
1. **request** – This is same as prototype scope. however, it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
2. **session** – A new bean will be created for each HTTP session by the container.
3. **global-session** – This is used to create global session beans for Portlet applications.

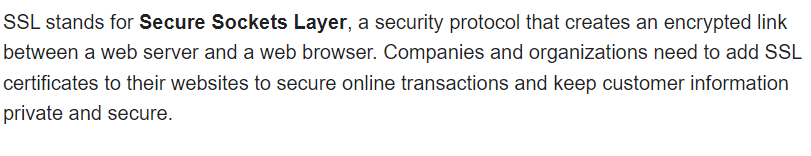
**request**, **session** and **global-session** scopes are not available for standalone applications.

**Idempotency**

The methods GET, PUT and DELETE – Idempotent method as duplicate calls won’t affect anything, whereas POST is non-idempotent as post is creating a new record with unique id every time. So, this is NI method.

**HTTP and HTTPS both are stateless protocols**.The S in HTTPS stands for Secure and it refers to use of ordinary HTTP over an encrypted SSL/TLS connection. Each request to server will consider as the new request since server doesn’t store the state of the previous request. SSL/TLS- **secure sockets layer and transport layer security**





#### Spring Boot:

**Spring Boot Actuator** is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live. If we want to get production-ready features in an application, we should use the **Spring Boot actuator.**

These actuators include auditing, health, CPU usage, HTTP hits, and metric gathering, and many more that are automatically applied to your application.

For example, the **/health** endpoint provides the basic health information of an application. The actuator, by default, mapped it to **/actuator/health**.

/beans-> It is used to display a complete list of all the Spring beans in your application.

/env-> It is used to expose properties from Spring's Configurable Environment.

/trace-> It is used to display trace information.

/info-> It is used to display arbitrary application info.

**What are the actuator-provided endpoints used for monitoring the Spring boot application?**

Actuators provide below pre-defined endpoints to monitor our application -

* Health
* Info
* Beans
* Mappings
* Configprops
* Httptrace
* Heapdump
* Threaddump
* Shutdown

**Features of Spring Boot that make it different?**

* Creates stand-alone spring application with minimal configuration needed.
* It has embedded tomcat, jetty which makes it just code and run the application.
* Provide production-ready features such as metrics, health checks, and externalized configuration.
* Absolutely no requirement for XML configuration.

**Why Spring Boot over Spring?**

Below are some key points which spring boot offers but spring doesn’t:

* Starter POM.
* Version Management.
* Auto Configuration.
* Component Scanning.
* Embedded server.
* InMemory DB.
* Actuators

**@SpringBootApplication** internally has

@Configuration, @EnableAutoConfiguration, and @ComponentScan

**What are the most common Spring Boot CLI commands?**

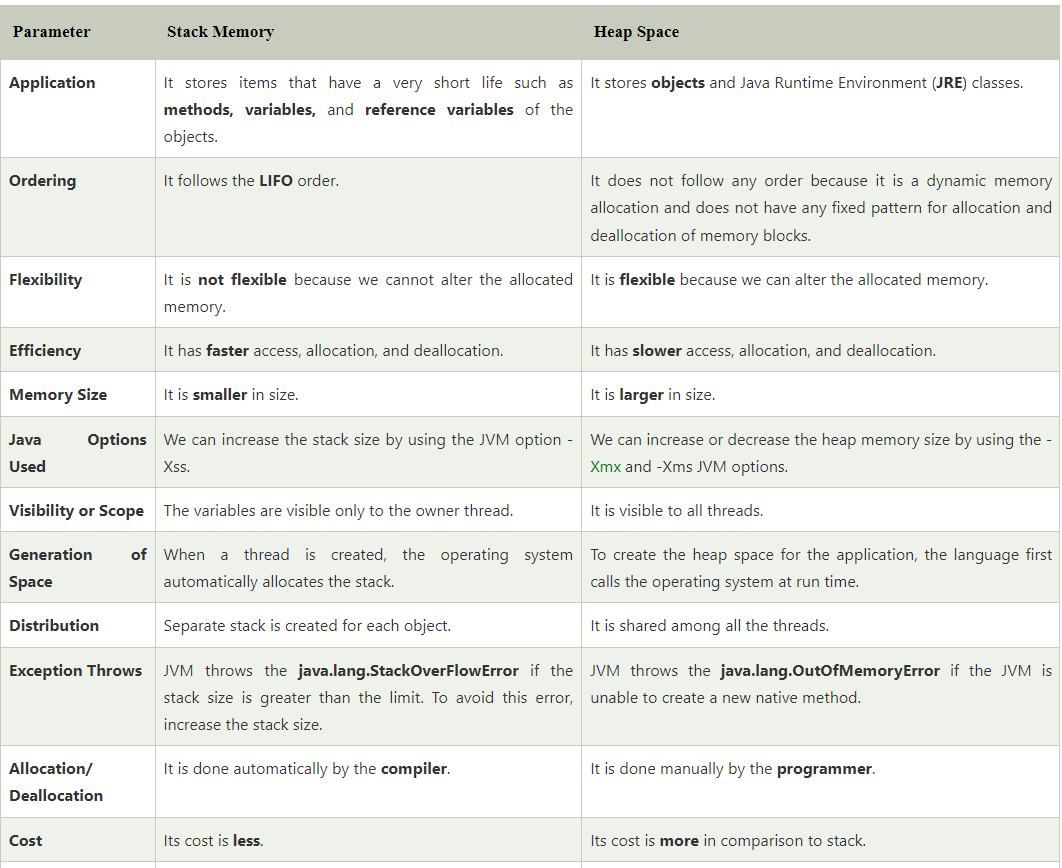
* -run, -test, -grap, -jar, -war, -install, -uninstall, --init, -shell, -help.

**What is Spring Boot dependency management?**

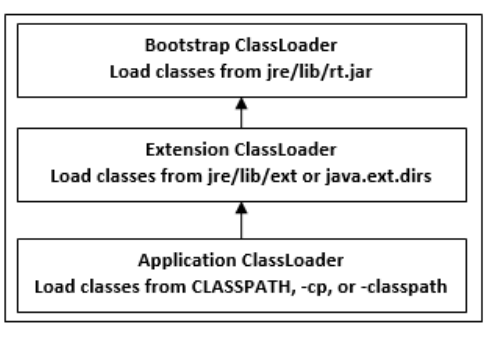
Spring Boot dependency management is used to manage dependencies and configuration automatically without you specifying the version for any of that’s dependencies.

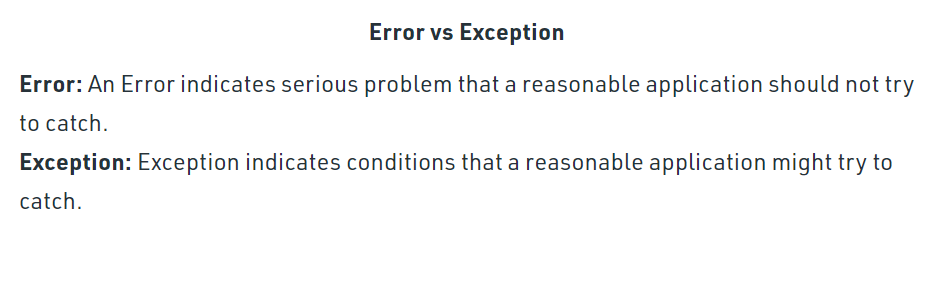
#### Java

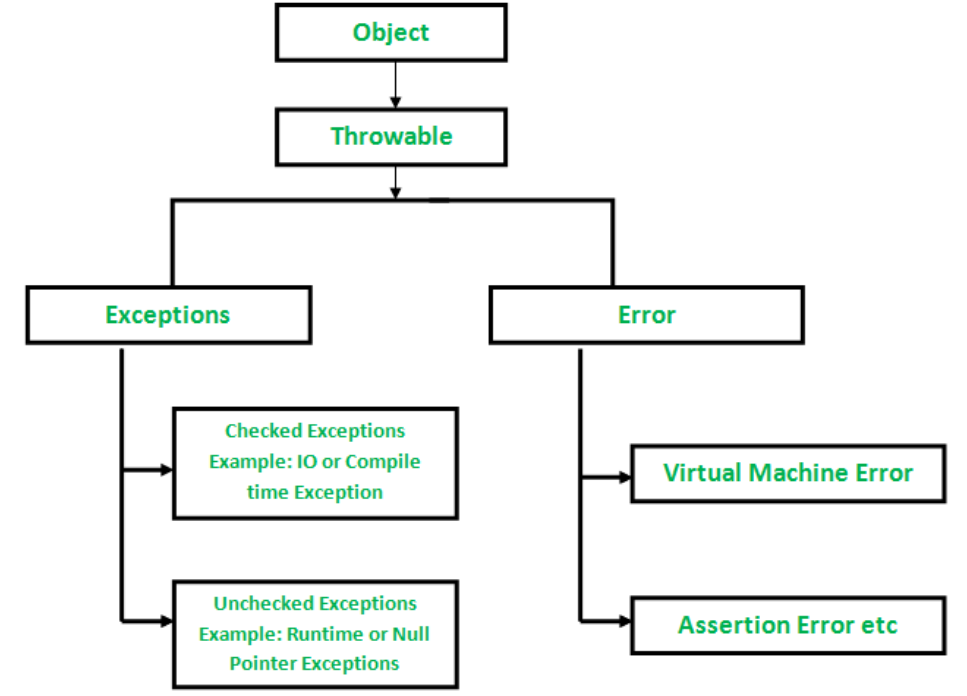
**Stack vs Heap memory:**



**Class Loader:**





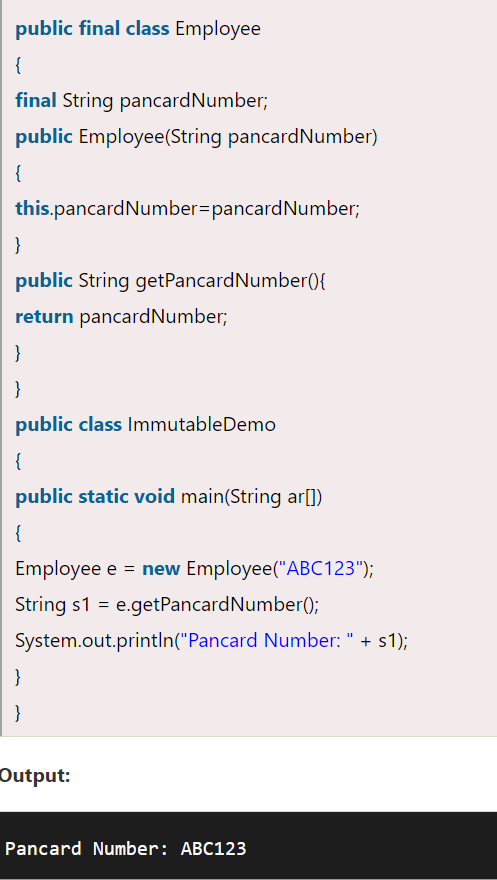


**Immutable class**

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.

* 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.*



**Serialization/DeSerialization**:   
Serialization in Java is the process of converting an object into a stream of bytes, which can be persisted to a file, sent over a network, or stored in a database. Java provides a built-in mechanism for serialization through the **java.io.Serializable** interface.

### 1. serialVersionUID Not Defined:

If you do not define a **serialVersionUID**, the JVM will generate one for you based on the class structure. If the class structure changes (adding or removing fields, changing their types, etc.), the automatically generated **serialVersionUID** may change, leading to potential compatibility issues during deserialization.

### 2. Same serialVersionUID in Different Classes:

If two different classes have the same **serialVersionUID**, they are considered by the JVM to be compatible in terms of serialization. This can lead to unexpected behavior during deserialization, as the JVM may attempt to cast objects of one class into the other.

**Serializable interface**. The Serializable interface is a **marker** interface; it declares **no methods** at all. It tells the serialization mechanism that the class can be serialized. ... That is **done by calling the writeObject()** method of the java

**What is the purpose of static methods and variables?**

Static variables are used with the class name and the dot operator, since they are associated with a class, not objects of a class. Static methods cannot access or change the values of instance variables, but they can access or change the values of static variables. Static methods cannot call non-static methods.

**What is the static variable?**

It is a variable which belongs to the class and not to object. Static variables are initialized only once, at the **start of the execution**.

**What is the static method?**

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

**Why does Java not support pointers?**

Java has a robust security model and disallows pointer arithmetic for the same reason. No pointer support makes Java more secure because they point to memory location or used for memory management that loses the security as we use them directly.

**HashMap**, In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable (uses the compareTo method of Comparable ), and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

If entries are removed from the map, the number of entries in the bucket might reduce such that this tree structure is no longer necessary. That's what the **UNTREEIFY\_THRESHOLD = 6** is for. If the number of elements in a bucket drops below six, we might as well go back to using a linked list.

Hash map is not synchronized. But we can make it synchronized externally.

Collections.synchronizedMap(hashmap object);

**Hashmap** inherits from **AbstractMap** class while **HashTable** inherits from **Dictionary** class.

1. **Fail-Fast Iterator:**
   * **Behavior:** If the underlying collection is structurally modified (e.g., elements are added, removed, or modified) after the iterator is created, a fail-fast iterator detects this modification and throws a **ConcurrentModificationException** immediately. This is a safety mechanism to prevent concurrent modification issues and to ensure that the iterator reflects the state of the collection at the time it was created.
   * **Example:** **ArrayList**, **HashMap**, and other standard Java collections typically use fail-fast iterators.
2. **Fail-Safe Iterator:**
   * **Behavior:** A fail-safe iterator, on the other hand, does not throw an exception if the underlying collection is modified during iteration. Instead, it continues to operate on the original copy of the collection that was present when the iterator was created. Modifications to the collection after iterator creation do not affect the iterator, and the iterator completes its iteration based on the snapshot of the collection taken during its creation.
   * **Example:** The **CopyOnWriteArrayList** class in Java provides a fail-safe iterator. It achieves this by creating a copy of the underlying array whenever a modification is made, allowing existing iterators to continue operating on the original data.

List<String> list = new ArrayList<>(Arrays.asList("one", "two", "three"));

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

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

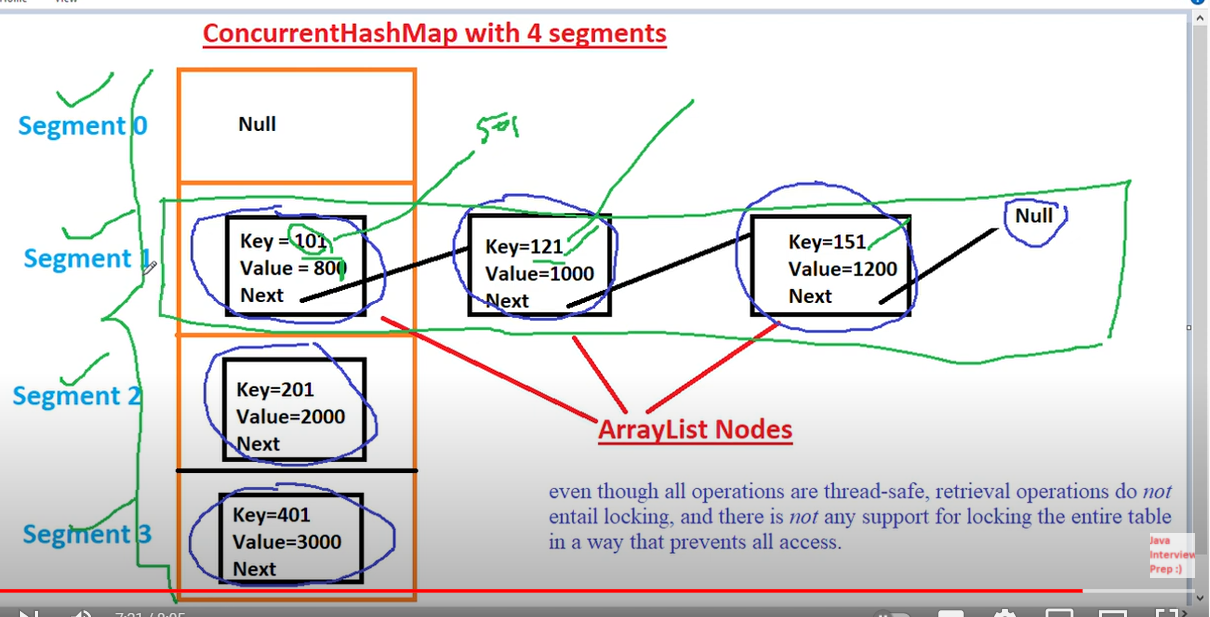
// ConcurrentModificationException will be thrown if the list is modified here

list.add("four");

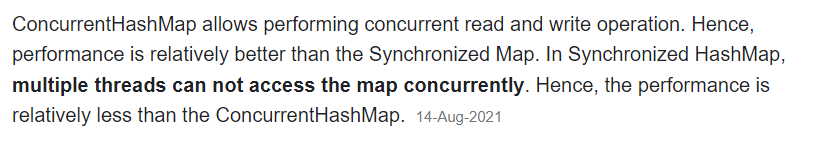
}

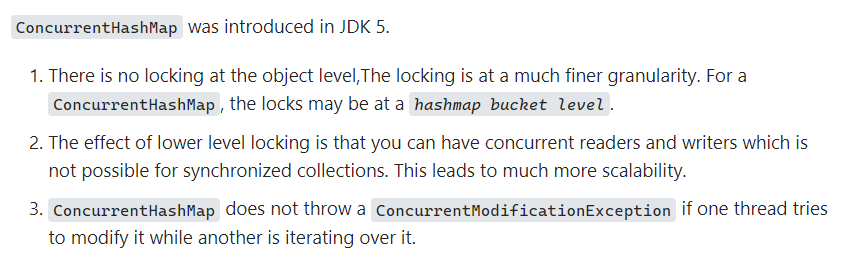
<https://www.geeksforgeeks.org/fail-fast-fail-safe-iterators-java/>

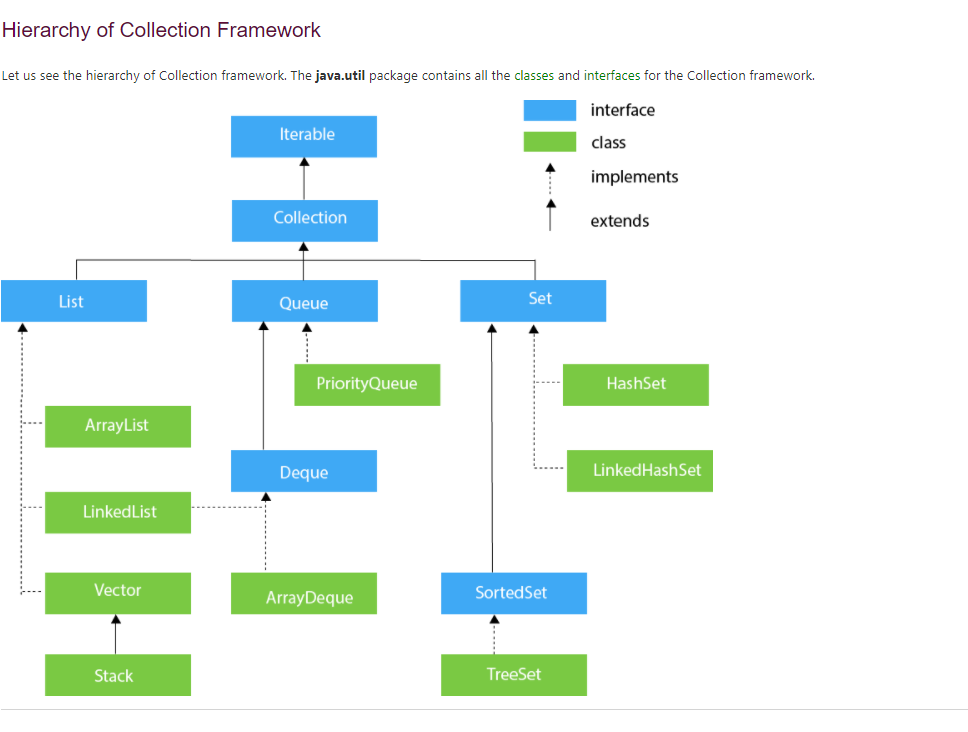
**Concurrent HashMap**: It is using **reentrant Lock** to lock particular portion of map. Only the segments get locked (which we are updating) not entire map. It’s thread safe. Though we are having Hashtable for synchronized access, it will slow. But CHM will only lock the particular segment.



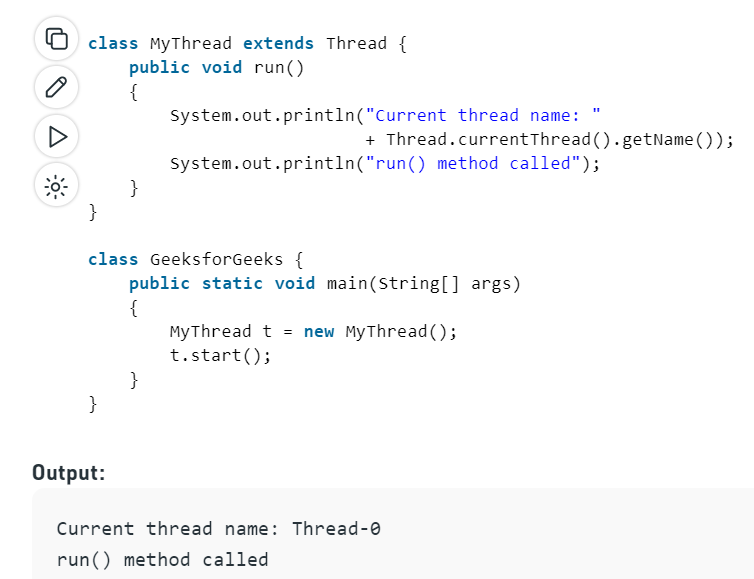
**Difference between synchronized HashMap and ConcurrentHashMap**







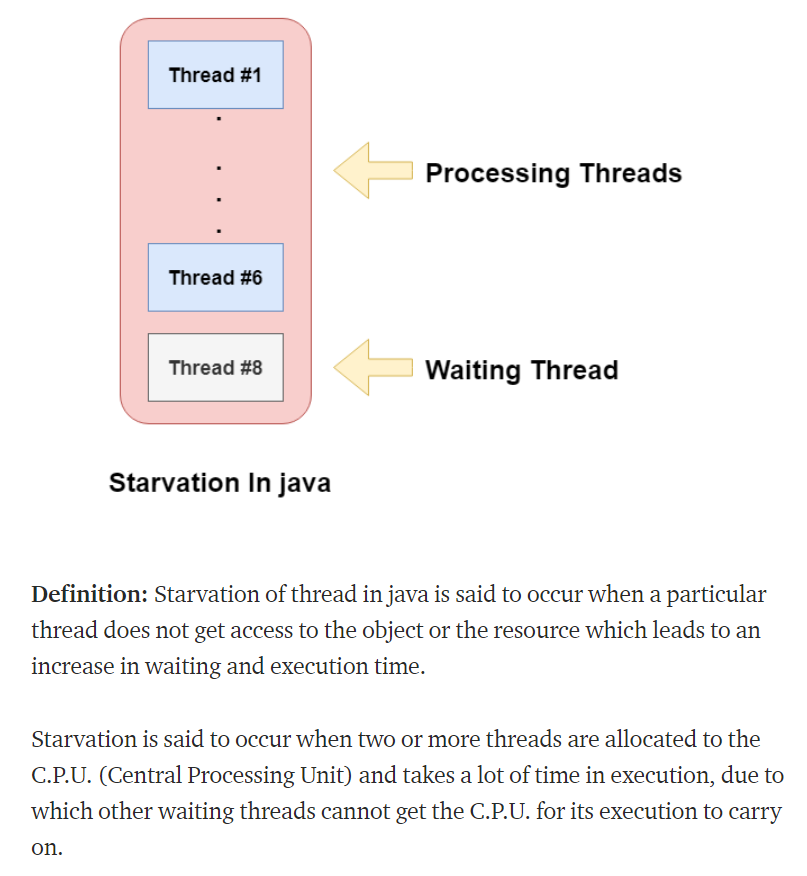
**New Thread creation:** When a program calls the start() method, a new thread is created and then the run() method is executed. But if we directly call the run() method then no new thread will be created and run() method will be executed as a normal method call on the current calling thread itself and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



**If one Thread throws Runtime Exception will complete application (other running threads) goes down?** No. Only thread from which the exception is occurred will terminate. Other thread will continue to run and progress if an exception is thrown from one thread.

Whenever, we use wait(), the method should be synchronized one. Also notify() is used to notify the another thread to start

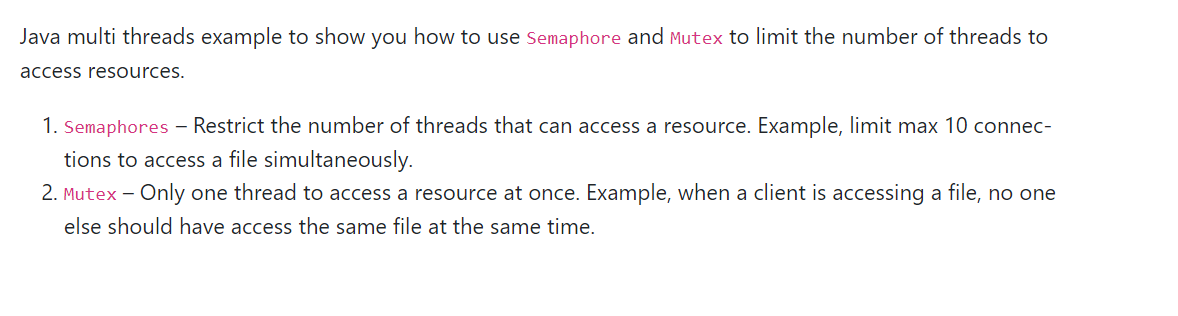
**Starvation:**



**Deadlock**: When the situation that Thread1 is waiting for thread2 gets released and Thread2 is waiting for thread1 to get released.

**Race Condition:** A sort-of-canonical definition is "*when two threads access the same location in memory at the same time, and at least one of the accesses is a write*." In the situation the "reader" thread may get the old value or the new value, depending on which thread "wins the race."

**Semaphores Mutual Exclusion:**



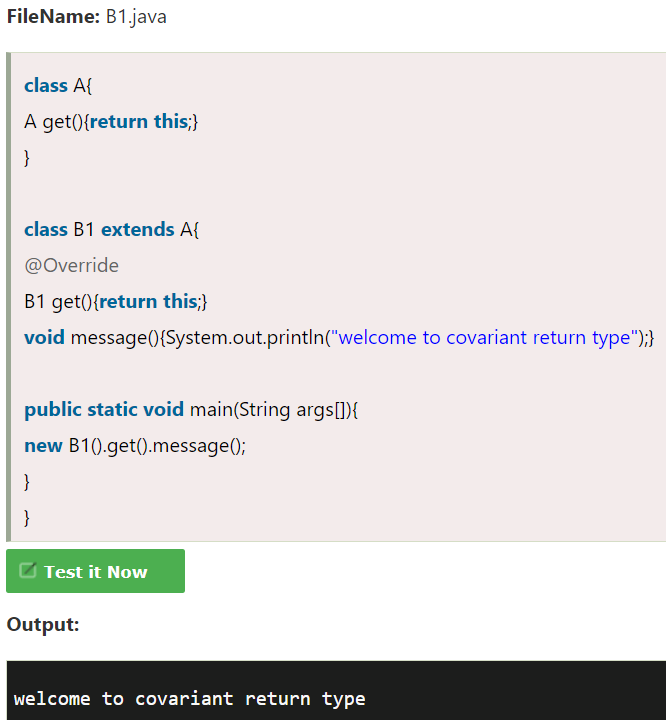
Semaphore Example: Consider an ATM cubicle with 4 ATMs, Semaphore can make sure only 4 people can access simultaneously.

Mutex Example: Mutex is the Semaphore with an access count of 1. Consider a situation of using lockers in the bank. Usually, the rule is that only one person is allowed to enter the locker room.

Generics(Template):

**Covariant Return Type**

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. Covariant return type helps in preventing the run-time ClassCastExceptions on returns.



**Call by Value and Call by Reference in Java**

|  |
| --- |
| There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method. |

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value.

**Volatile Keyword in Java**

Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

The volatile keyword does not cache the value of the variable and always read the variable from the **main memory**. The volatile keyword cannot be used with classes or methods. However, it is used with variables. It also guarantees visibility and ordering. It prevents the compiler from the reordering of code.

The contents of the particular device register could change at any time, so you need the volatile keyword to ensure that such accesses are not optimized away by the compiler.

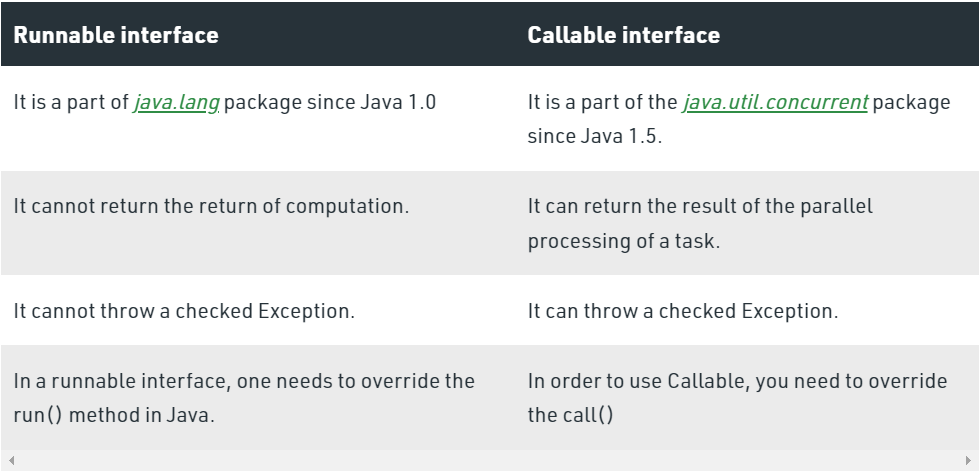
1. **class** Test
2. {
3. **static** **int** var=5;
4. }

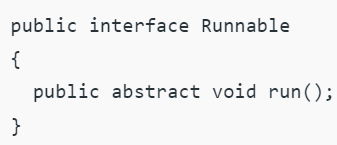
In the above example, assume that two threads are working on the same class. Both threads run on different processors where v thread has its local copy of var. If any thread modifies its value, the change will not reflect in the original one in the main memory. It leads to **data inconsistency because the other thread is not aware of the modified value.**

1. **class** Test
2. {
3. **static** **volatile** **int** var =5;
4. }

In the above example, static variables are class members that are shared among all objects. There is only one copy in the main memory. The value of a volatile variable will never be stored in the cache. All read and write will be done from and to the main memory.

**Callable and Runnable:**





**Autoboxing:**Converting a primitive value into an object of the corresponding [wrapper class](https://www.geeksforgeeks.org/wrapper-classes-java/) is called autoboxing. For example, converting int to [Integer class](https://www.geeksforgeeks.org/wrapper-classes-java/). The Java compiler applies autoboxing when a primitive value is:

* Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
* Assigned to a variable of the corresponding **wrapper class**.

**Eg**:  List<Integer> list = new ArrayList<Integer>();

list.add(2); //2 is the primitive

**Unboxing:** Converting an object of a wrapper type to its corresponding primitive value is called unboxing. For example conversion of [Integer](https://www.geeksforgeeks.org/wrapper-classes-java/) to int. The Java compiler applies unboxing when an object of a wrapper class is:

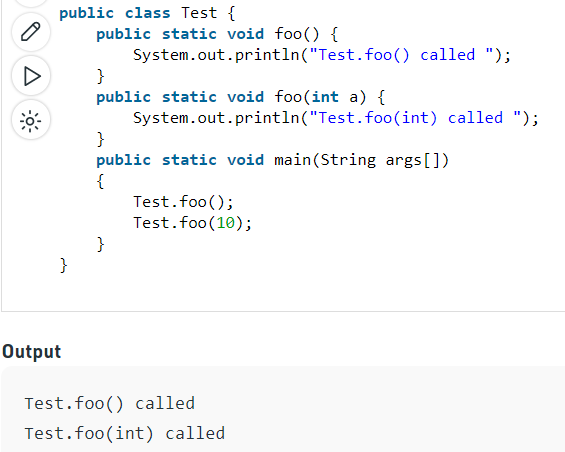
* Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
* Assigned to a variable of the corresponding **primitive type**.

**Eg**: Integer i = new Integer(10);

// unboxing the Object

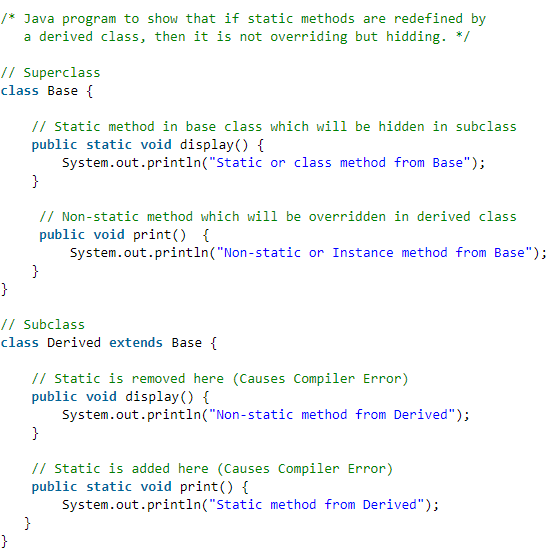
int i1 = i;

**Can we overload static methods?**   
The answer is ‘Yes’. We can have two or more static methods with the same name, but differences in input parameters. For example, consider the following Java program.



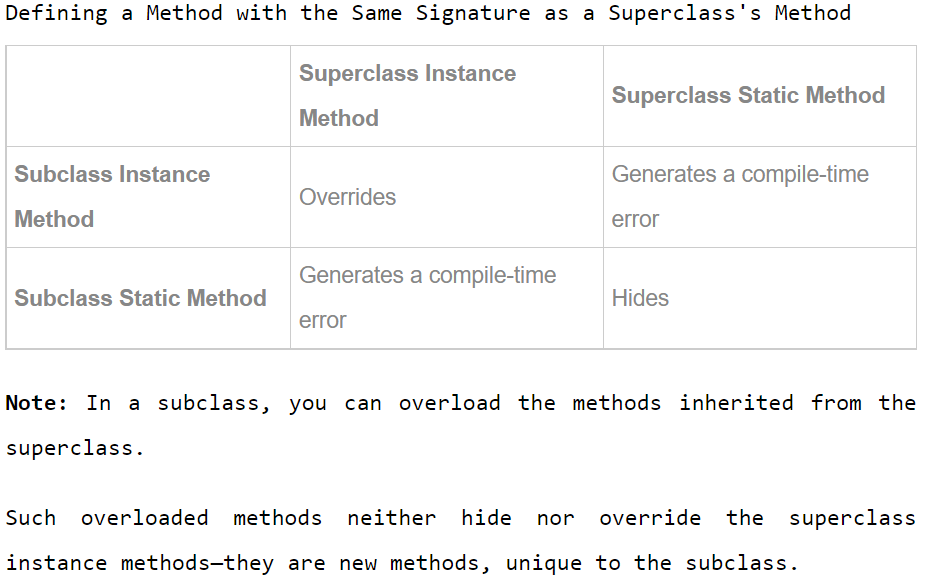
**Can we Override static methods in java?**   
We can declare static methods with the same signature in the subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘No’.   
If a derived class defines a static method with the same signature as a static method in the base class, the method in the derived class is hidden by the method in the base class.

The following are some important points for method overriding and static methods in Java.   
**1)** For class (or static) methods, the method according to the type of reference is called, not according to the object being referred, which means method call is decided at compile time.  
**2)** For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.  
**3)** An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.



**Output:** Static or class method from Base

Non-static or Instance method from Derived



**Field hiding**: <https://www.tutorialspoint.co0m/explain-about-field-hiding-in-java>

**A class member declared protected becomes member of subclass of which type**  A class member declared protected becomes private member of subclass.

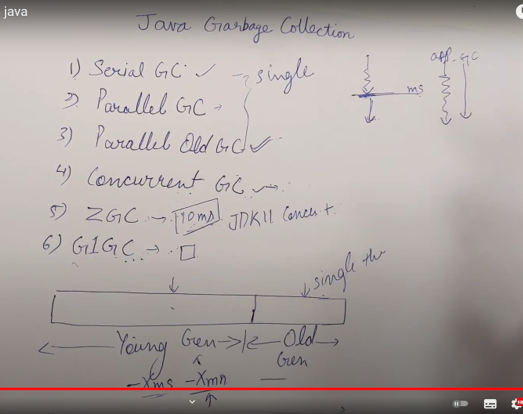
**Are constructors and initializers also inherited to sub classes in Java?**

No, Constructors and initializers (Static initializers and instance initializers) are not inherited to sub classes. But they are executed while instantiating a sub class.

**Are static members inherited to sub classes in Java?**

Yes, Static members are also inherited to sub classes.

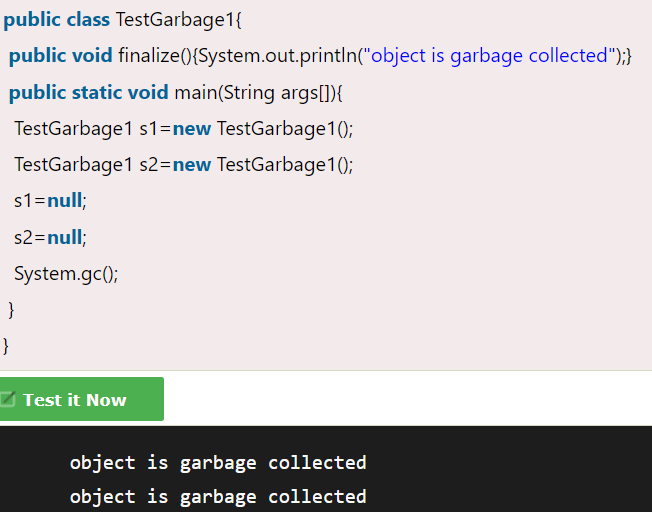
**Garbage collection:**



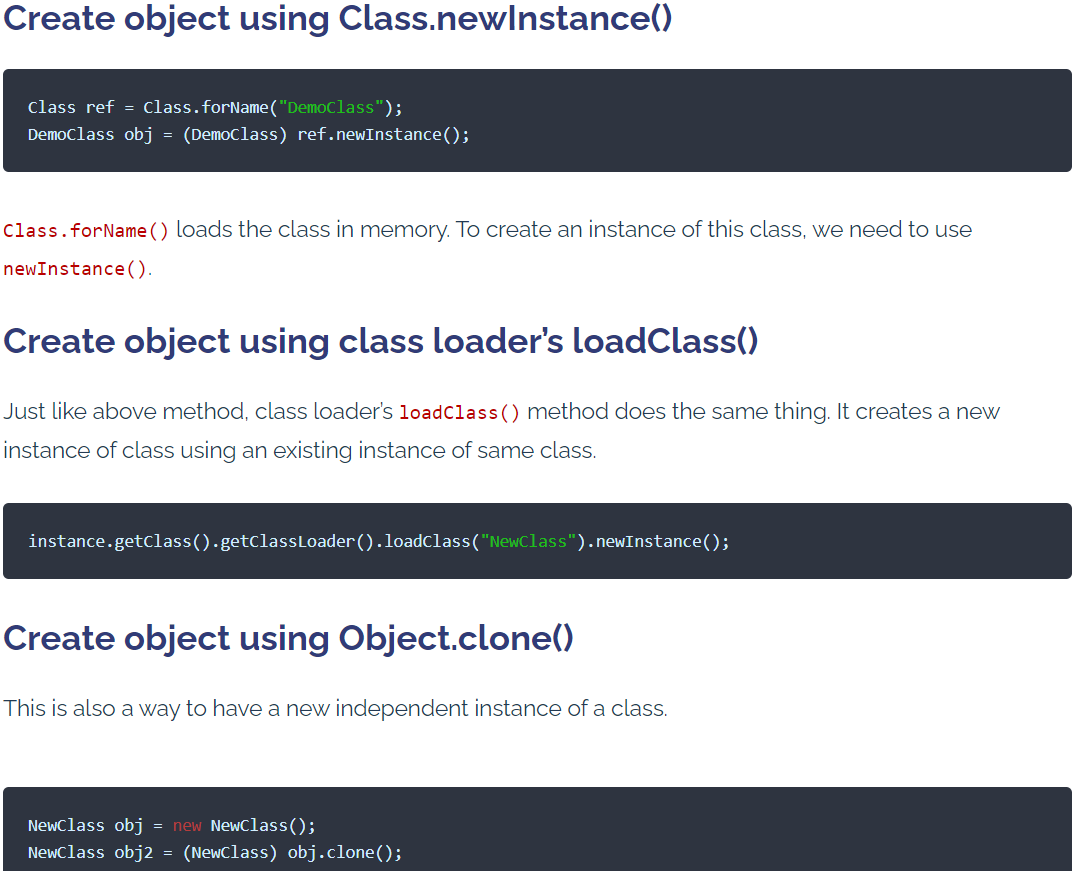
The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in **System** and **Runtime** classes.

The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

Garbage collection is performed by a daemon thread called Garbage Collector (GC). This thread calls the finalize() method before object is garbage collected.



**Create Object without new Keyword:**

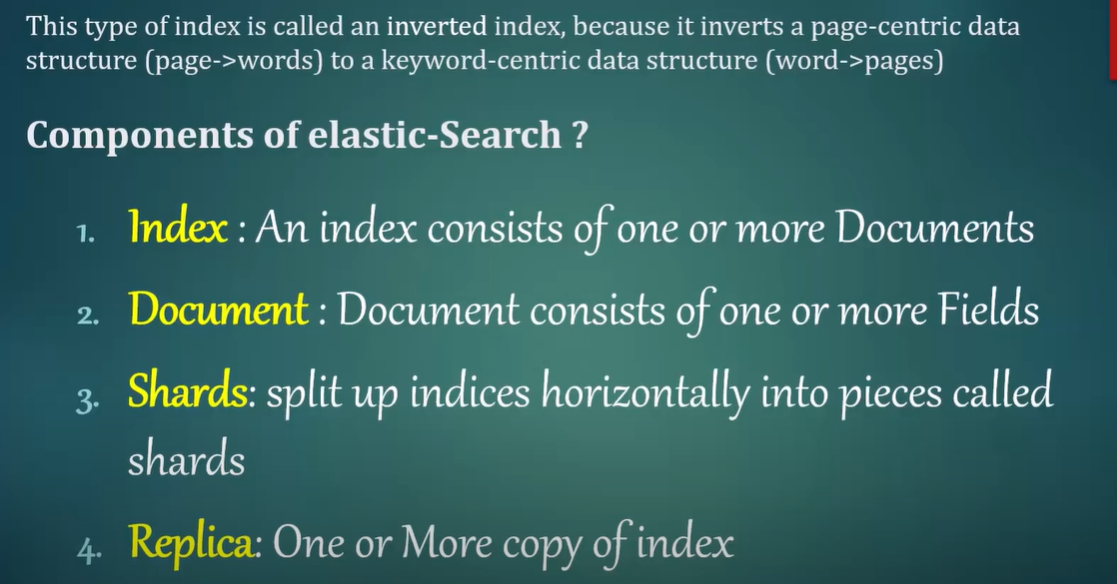


How to safe singleton class from reflections?

How to make singleton class as non -singleton?

https://www.geeksforgeeks.org/prevent-singleton-pattern-reflection-serialization-cloning/

#### Elastic search:

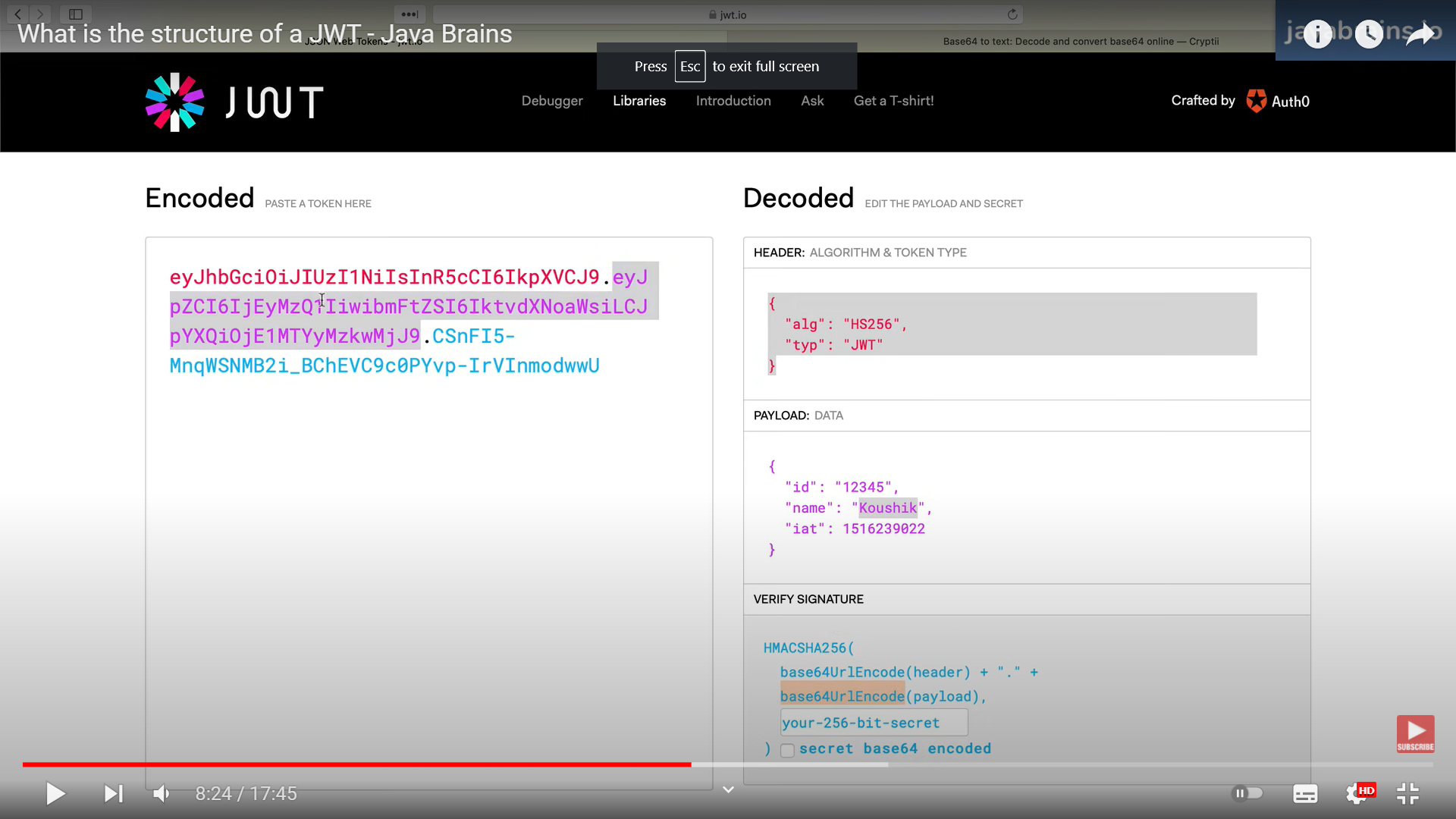


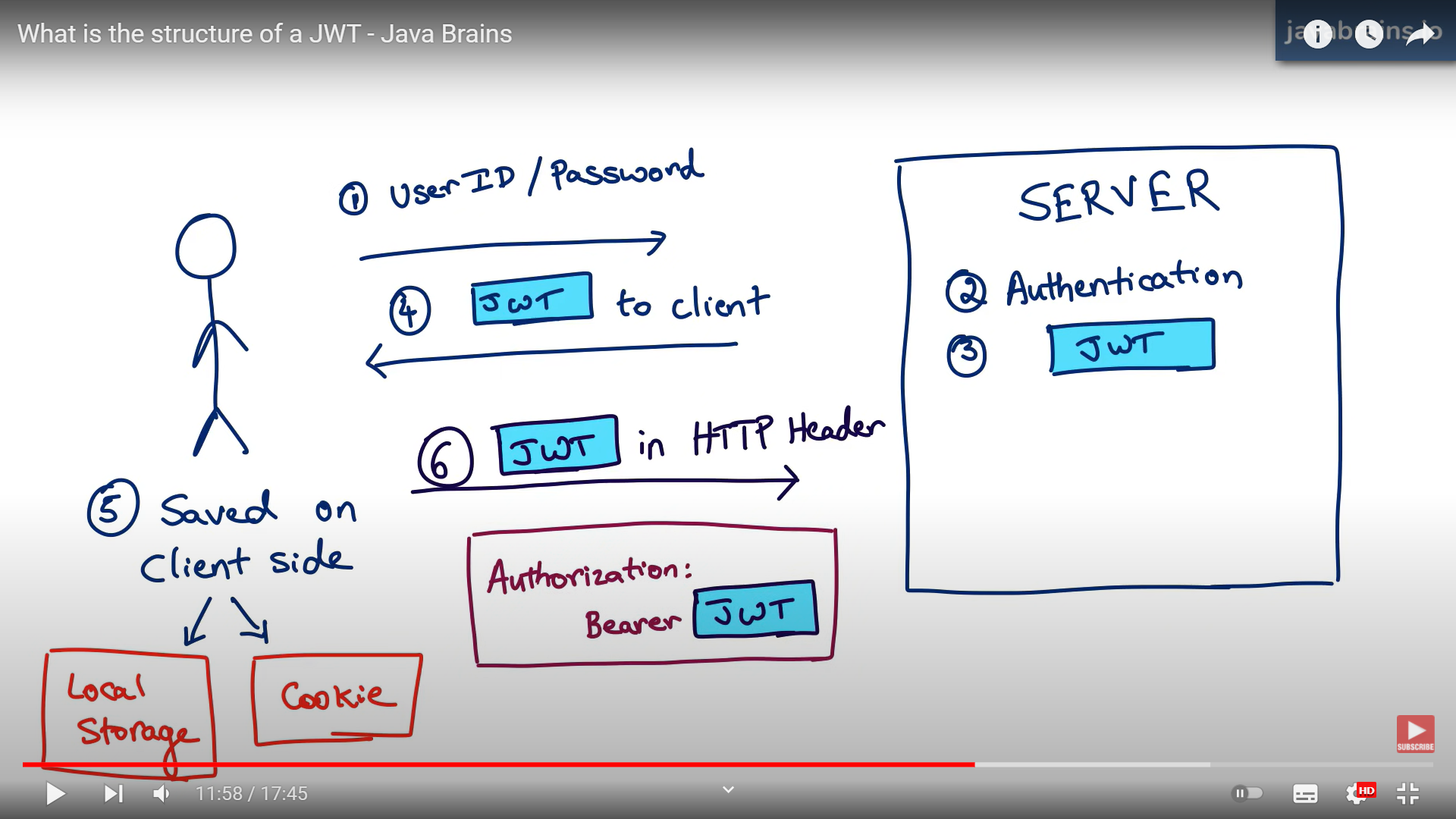
##### JWT:

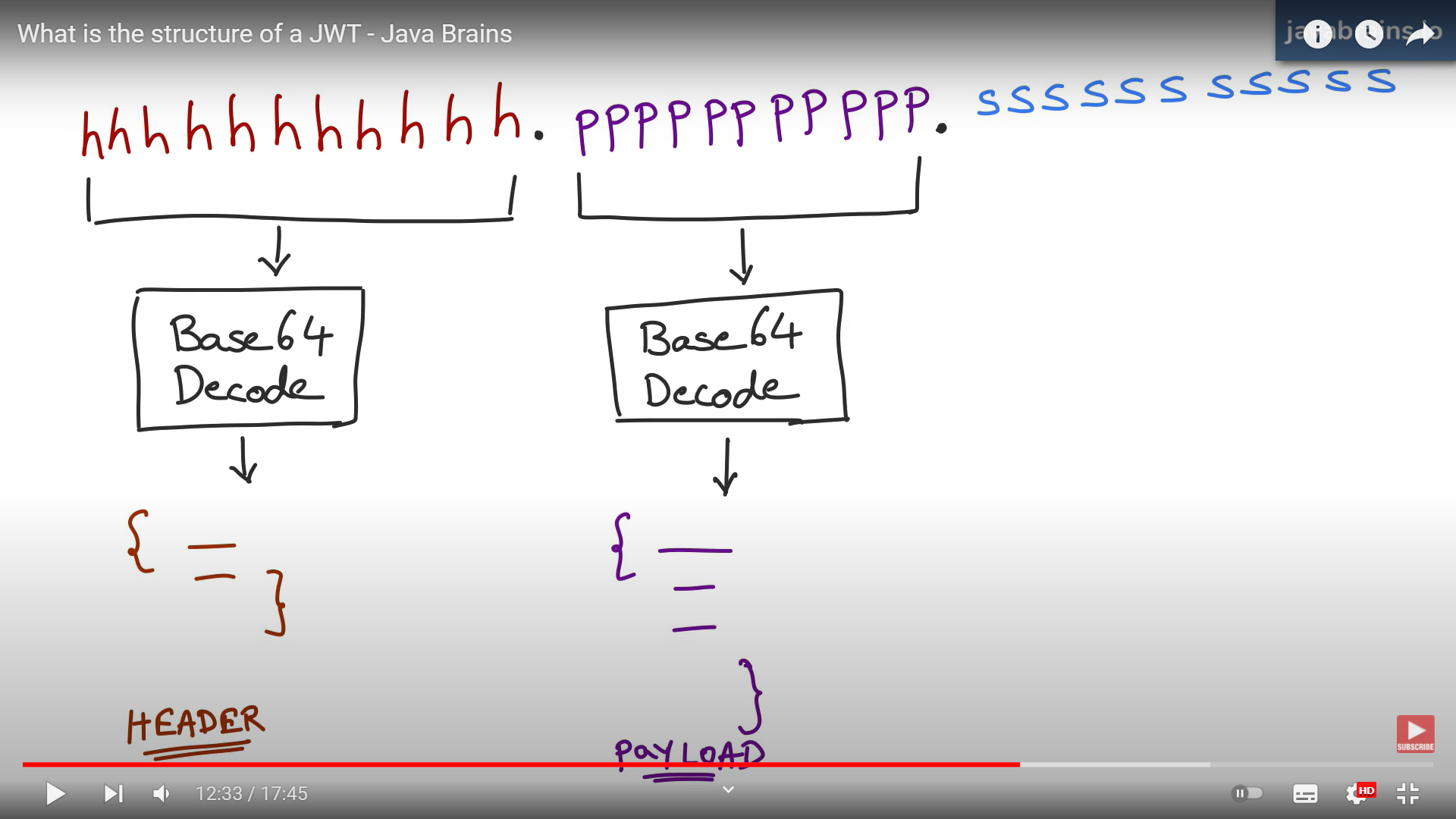
The jwt token string has 3 parts(header.payload.signature). As shown below with different colours. We can find header and payload value by passing encoded section for decoding; however, signature part is not decoded as its tied with that specific header and payload pairs. If you are changing the payload/header, signature gets changed and vice versa.

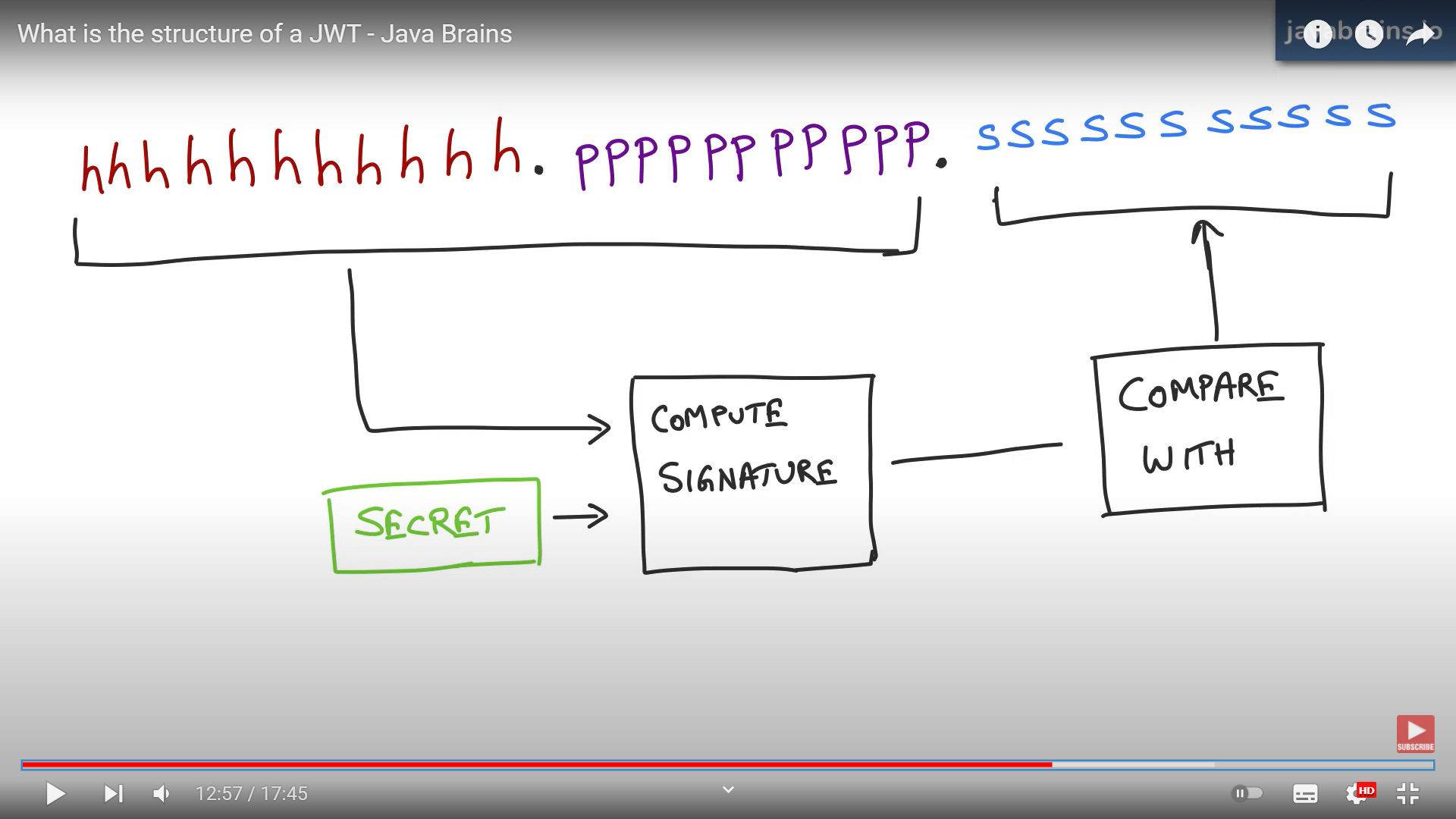
AlgHS256 is Base-64 Algorithm

**JWT will come into picture when authentication is completed and passes the token to every subsequent request to make sure that request has called from authenticated side.**









It’s possible to steal jwt token by calling the api which we developed for generating jwt such as /generateJWT. OAuth is stopping the steal of jwt. Visibility is not private in JWT.

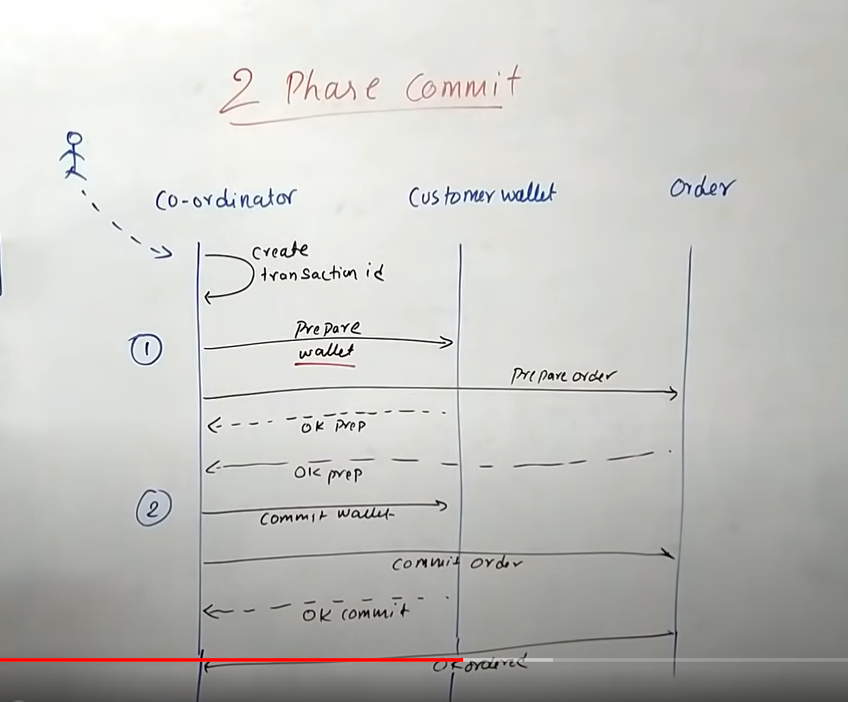
**OAuth:**

It’s not for authentication. It’s for authorization between services.

**Patterns for distributed transactions within a microservices architecture (Micro service A to B to C how to handle commits and rollback here)**

**Use Saga pattern,**

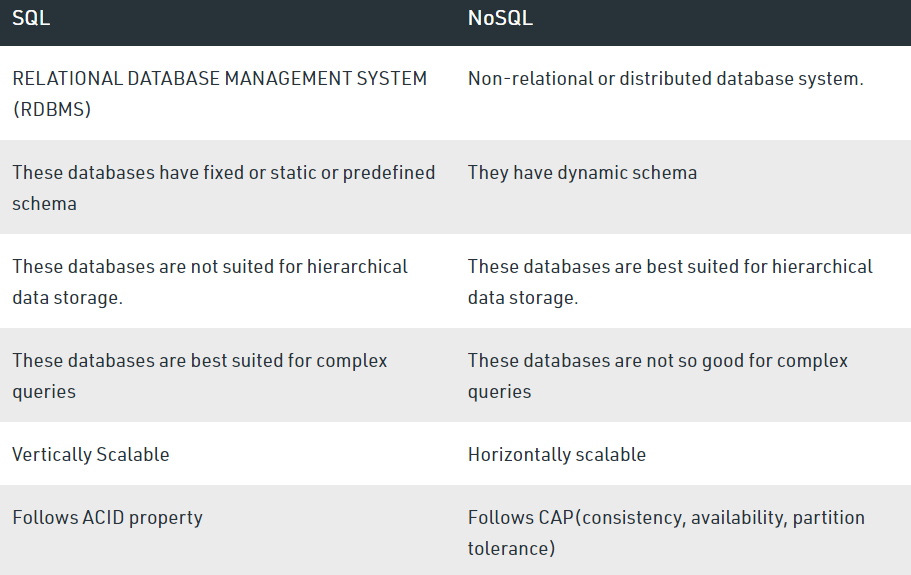
* Choreography-Based Saga. In this approach, there is no central orchestrator.
* Orchestration-Based Saga In this approach, there is a Saga orchestrator that manages all the transactions and directs the participant services to execute local transactions based on events.



**It will lock the payment and inventory table for that particular id. None of the transaction won’t go and update for that id until that transaction completes.**

Asynchronous meaning

##### SQL and NoSQL difference:



**Java to database connections (Explore connection class). Each time connection will establish or how??**

\_\_\_\_\_\_\_\_\_Need to be answered\_\_\_\_\_\_\_\_

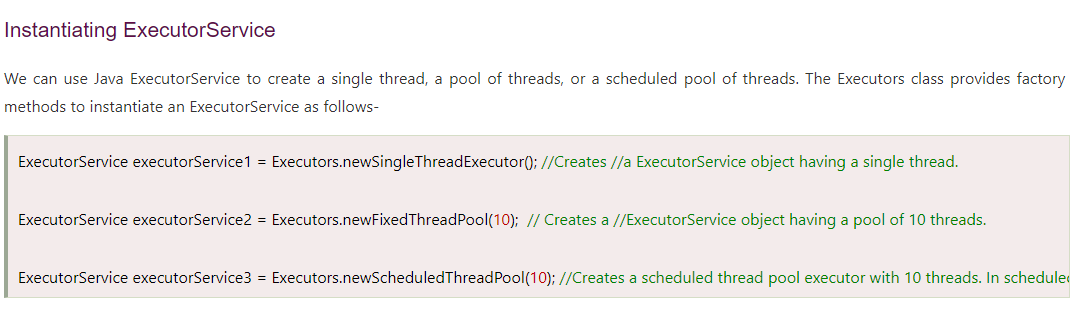
##### CompletableFuture

A CompletableFuture is **an extension to Java's Future API** which was introduced in Java 8. A Future is used for asynchronous Programming. It provides two methods, isDone() and get(). The methods retrieve the result of the computation when it completes.

##### Executable service:

The Java ExecutorService is the interface which allows us **to execute tasks on threads asynchronously**. The Java ExecutorService interface is present in the java.util.concurrent package. The ExecutorService helps in **maintaining a pool of threads and assigns them tasks (Separate thread pool for each task-Bulkhead pattern)**. It also provides the facility to queue up tasks until there is a free thread available if the number of tasks is more than the threads available.

<https://www.javatpoint.com/java-executorservice>





**JDBC and SpringJDBCTemplate Difference**

**Hibernate:**

**https://github.com/in28minutes/jpa-with-hibernate/blob/master/notes.md**

JdbcTemplate.query(select statement) -> Get without where condition

JdbcTemplate.queryForObject (select statement with where id=?, new Object[]{id})->Get with where condition -> two arguments method

JdbcTemplate.update(..sqlquery) ->insert, update, delete

**Graphical user interface, text, application

Description automatically generated**

**Custom JDBCRowMapper:**

**Text

Description automatically generated**

**JPA:** JPA uses merge() function for update and insert. Its running select statement and if record exist, it will update else insert. @PersistenceContext- EntityManager

Text

Description automatically generated

**Named Query: (to retrieve all the records without where condition)**

**Graphical user interface, text, application

Description automatically generated**

**Multiple Named query for a class**

Text, letter

Description automatically generated

@Autowired

EntityManager em;

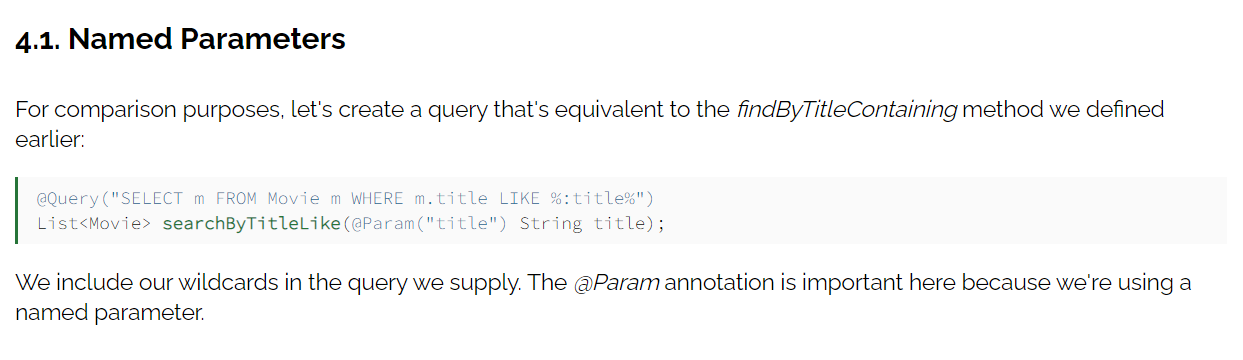
em.createNamedQuery(..,..); ->Query mentioned above in course class with @NamedQuery

em.createNativeQuery(“Select \* from Cource, Course.class); ->Actual query

@Transient -> Its used in Entity(Table pojo) to ignore the field which is not to be saved

Eg: @Transient

Private double percentage;



#### Microservice Communication:

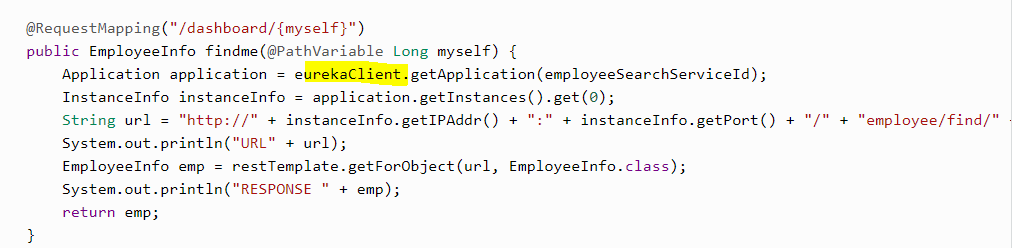
**Registering the Service:** Each microservice should be registered into the service registry with a unique name {service-id}, so it can be identified. Please note that it is an important step, as one of the main benefits of microservices is autoscaling; we can’t rely on the hostname/IP address, so a unique name is important in a distributed environment.

**Fetching the Registry:** Before calling the downstream/dependent service, the caller service fetches the registry from Eureka server. The registry contains all the active services registered into the service registry.

**Finding the Downstream Service:** Now, using the unique service Id, the caller service gets the instance of the downstream service.

**Resolving Underlying IP Address:** Please note the Unique service id act as a Key in service registry but network does not know about it, network expects Hostname to call the desired Rest Endpoint on the dependent service like (localhost:8080/employee/{id} or employee.cognizant,com/2 etc) so it is required to resolve the actual hostname of the dependent service Eureka API provides a method for that we just invoke that method to get the Ip address, For a distributed system it is the public IP of Load balancer.

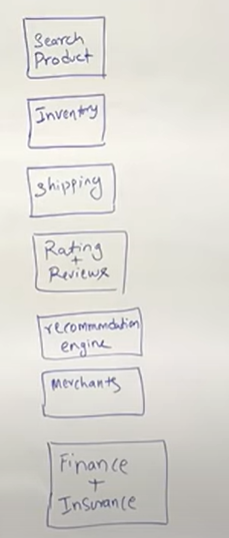
**Call the Rest Endpoint:** After resolving the IP address using Spring Resttemplate, we call the actual Rest endpoint and get the data.



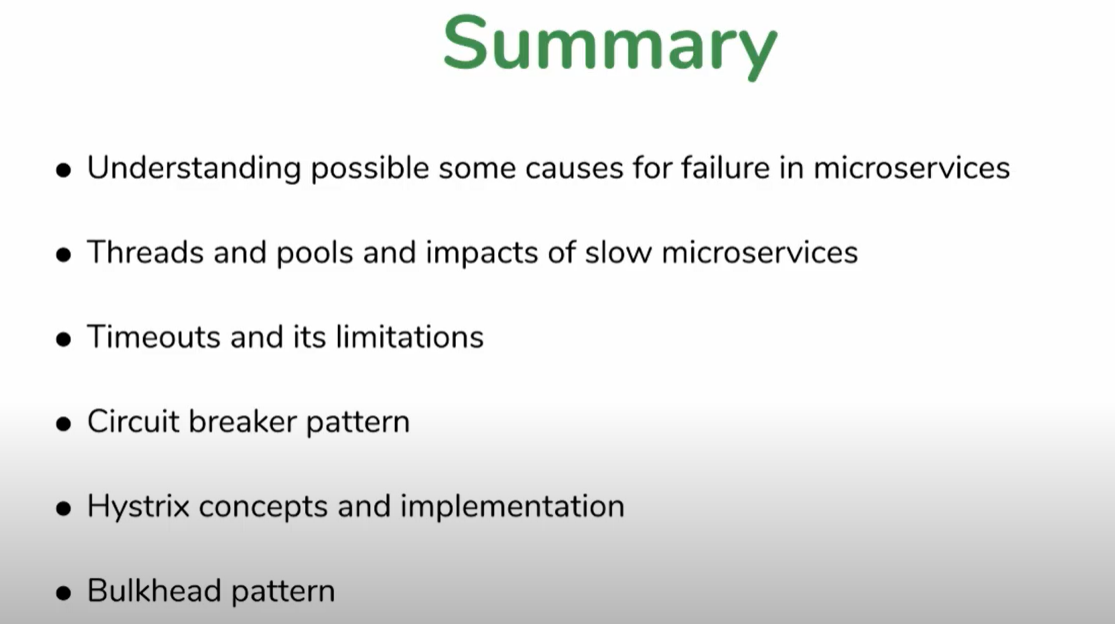
**How to scale microservices?**

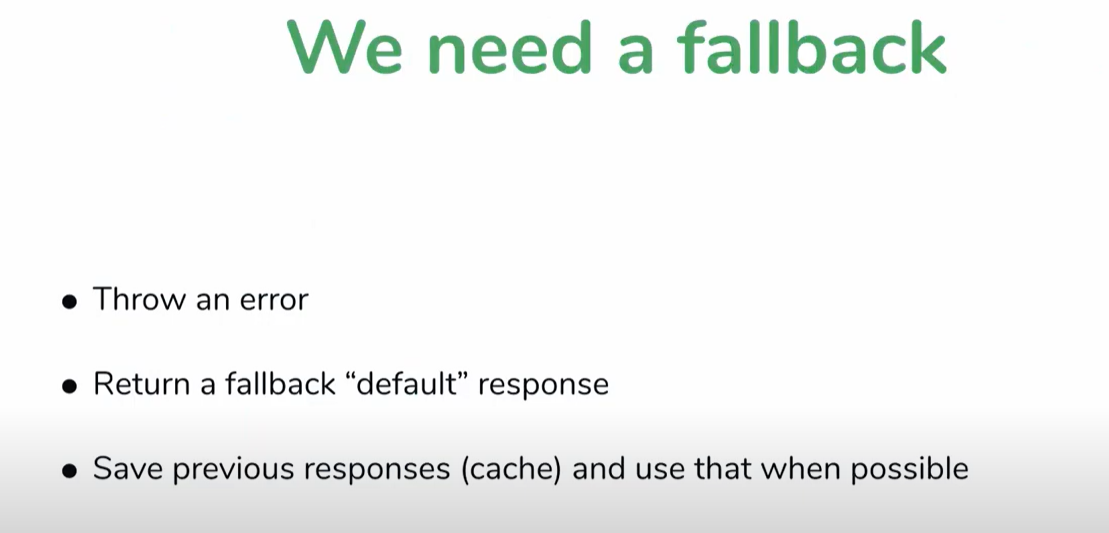
1.Horizontal scaling 2. Functional decomposition 3. Data partitioning 4. Vertical Scaling (ignore it)

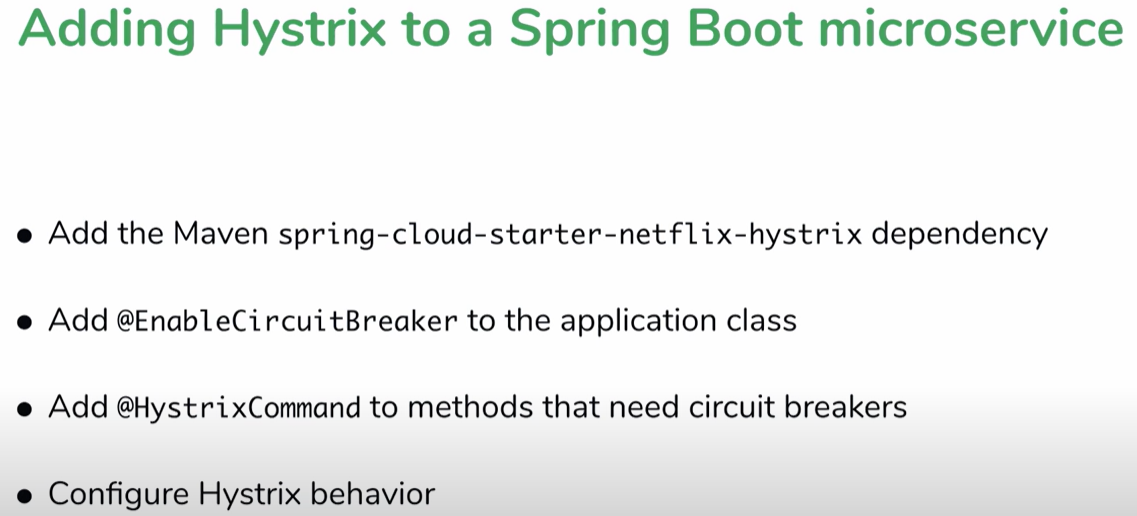
**API Gateway- eCommerce**

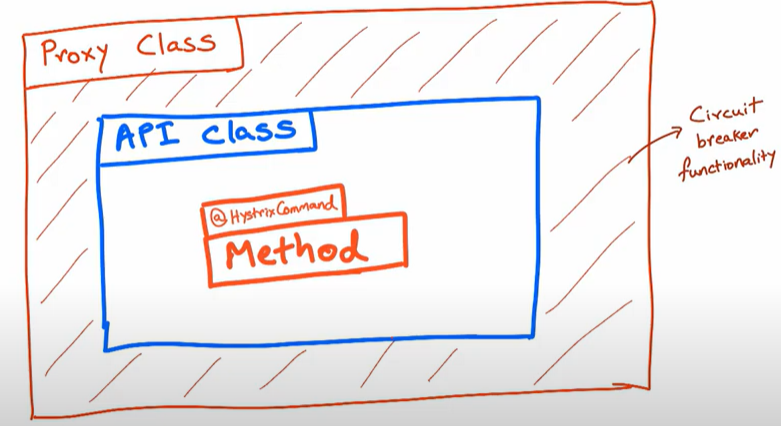


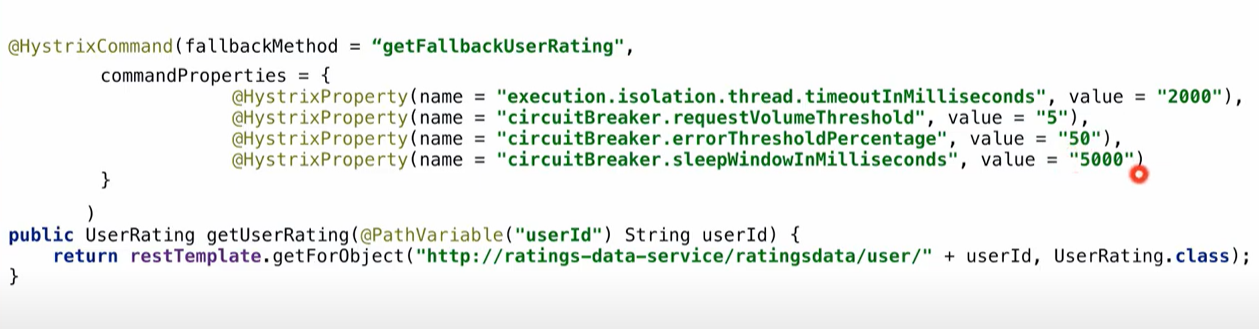
##### **Fault tolerance and resilience**







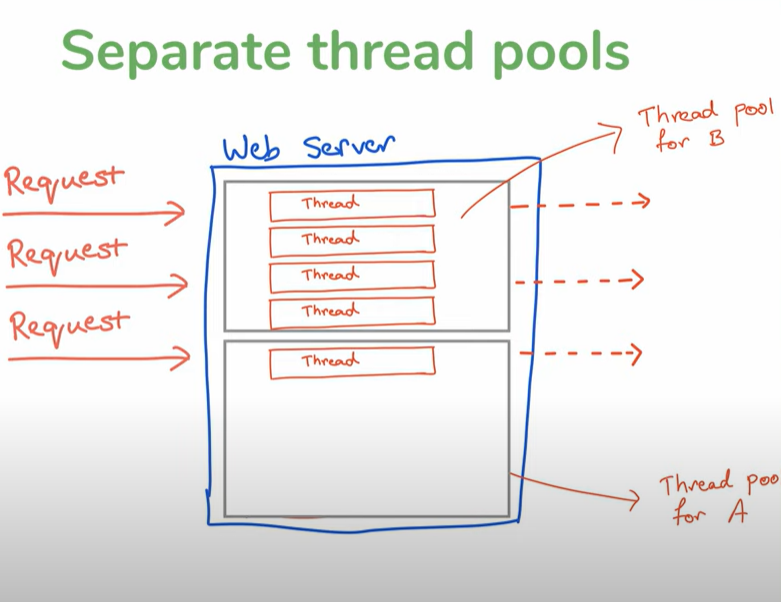




**If the methods are in the same class, hystrix won’t work.**

<https://www.youtube.com/watch?v=1EIb-4ipWFk&list=PLqq-6Pq4lTTbXZY_elyGv7IkKrfkSrX5e&index=18>





<https://www.youtube.com/watch?v=Kh3HxWk8YF4&list=PLqq-6Pq4lTTbXZY_elyGv7IkKrfkSrX5e&index=22>

### **How to disable a specific auto-configuration class?**

//use of exclude

@EnableAutoConfiguration(exclude={className})

**What is the use of Profiles in spring boot?**

While developing the application we deal with multiple environments such as dev, QA, Prod, and each environment requires a different configuration. For eg., we might be using an embedded H2 database for dev but for prod, we might have proprietary Oracle or DB2. Even if DBMS is the same across the environment, the URLs will be different.

To make this easy and clean, Spring has the provision of Profiles to keep the separate configuration of environment. https://dzone.com/articles/spring-boot-profiles-1

**Where do we define properties in the Spring Boot application?**

You can define both application and Spring boot-related properties into a file called application.properties. You can create this file manually or use Spring Initializer to create this file. You don’t need to do any special configuration to instruct Spring Boot to load this file, If it exists in classpath then spring boot automatically loads it and configure itself and the application code accordingly.

**What is an IOC container?**

IoC Container is a framework for implementing automatic dependency injection. It manages object creation and its life-time and also injects dependencies into the class.

**What is dependency Injection?**

The process of injecting dependent bean objects into target bean objects is called dependency injection.

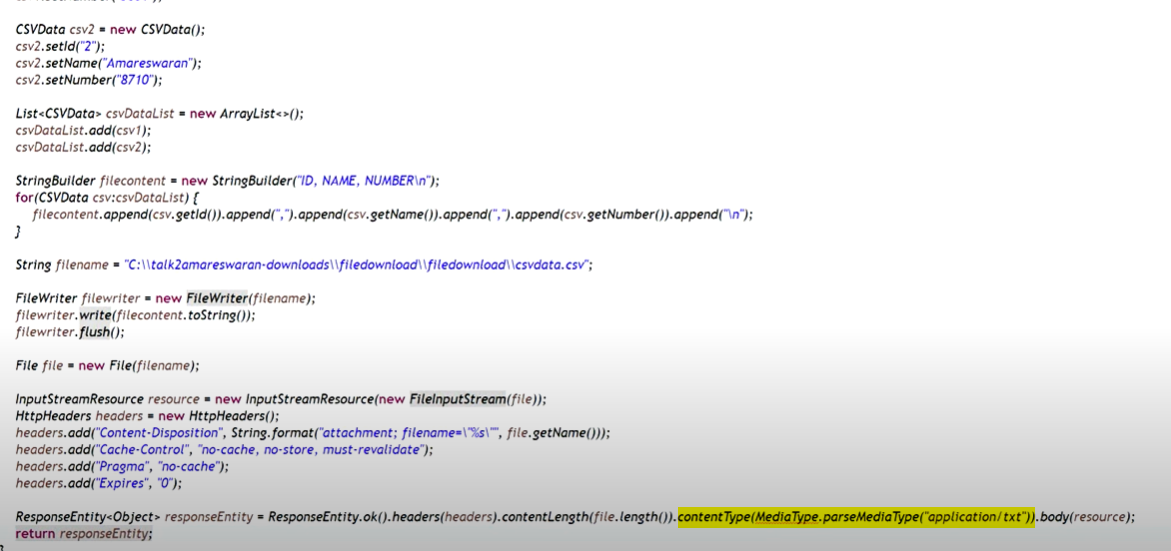
* **Setter Injection:** The IOC container will inject the dependent bean object into the target bean object by calling the setter method.
* **Constructor Injection**: The IOC container will inject the dependent bean object into the target bean object by calling the target bean constructor.
* **Field Injection:** The IOC container will inject the dependent bean object into the target bean object by Reflection API.

<https://www.javatpoint.com/spring-tutorial-dependency-injection-by-constructor>

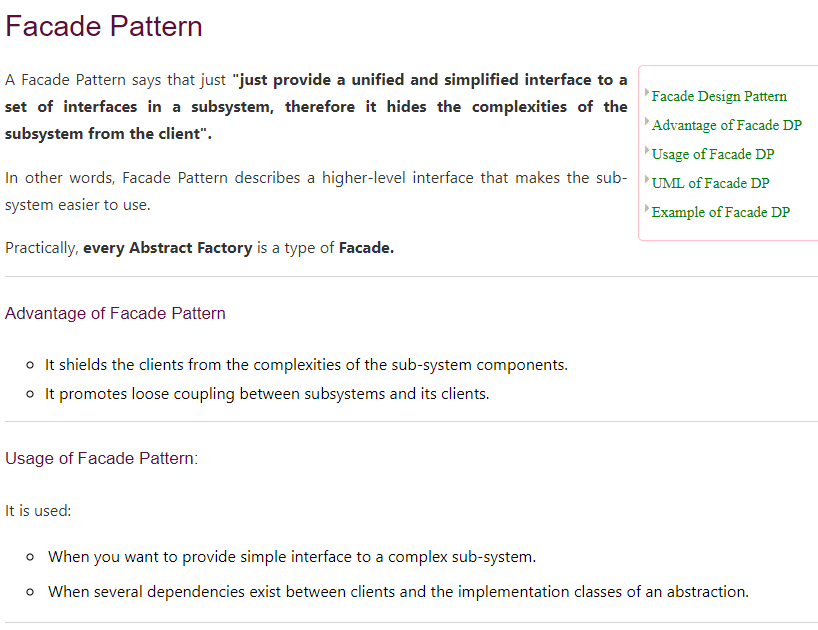
<https://www.javatpoint.com/spring-tutorial-dependency-injection-by-setter-method>

**Sent PDFfile(file) in apiResponse, if file is not there, then return json response with “File not found”. How can we develop same endpoint to return both json and pdfFile.**

ParseMediaType—Note it below



### Design Patterns:



**Observer Pattern:**

In youtube, subscribers are getting notified when video is added into the channel. There are two people, one is subject and another one is observer. When video got added, it takes all the subscriber of the channel and push notification to each subscriber objects.

* 1. How microservices A,B,C get connected?
  2. What happen if C fails. And what response returned?
  3. Difference between predicate and Consumer?
  4. Heap memory changes in java 8
  5. How concurrent hashmap is lock the bucket when there is a change during iteration
  6. LRU and ratelimiter – LLD
  7. Swiggy app, movie booking – HLD

**LLD:**

**Movie ticket booking**: https://github.com/naval41/Low-Level-Design/tree/master/MovieTicketBooking/src/com/codemate/bmshow