**GARBAGE COLLECTION**

GC concepts

<https://javapapers.com/java/java-garbage-collection-introduction/>

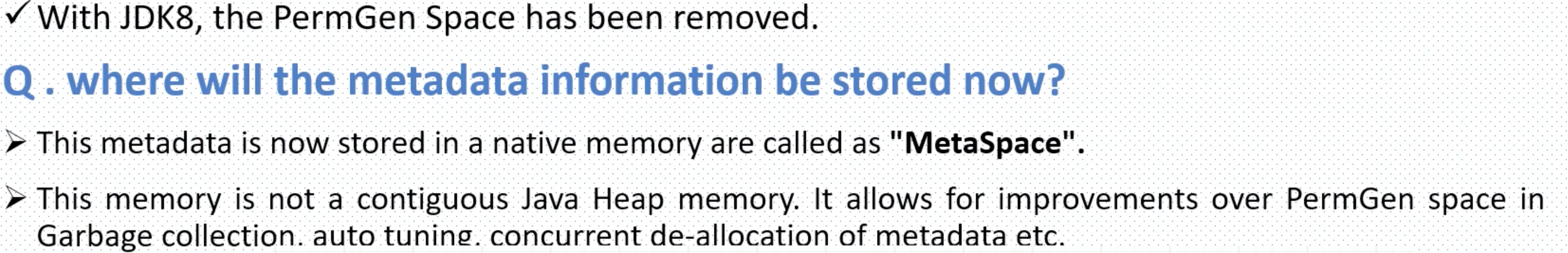
How GC Works

<https://javapapers.com/java/how-java-garbage-collection-works/>

Type of GC

<https://javapapers.com/java/types-of-java-garbage-collectors/>

**There are 4 type of garbage collections**



Java 8 Features

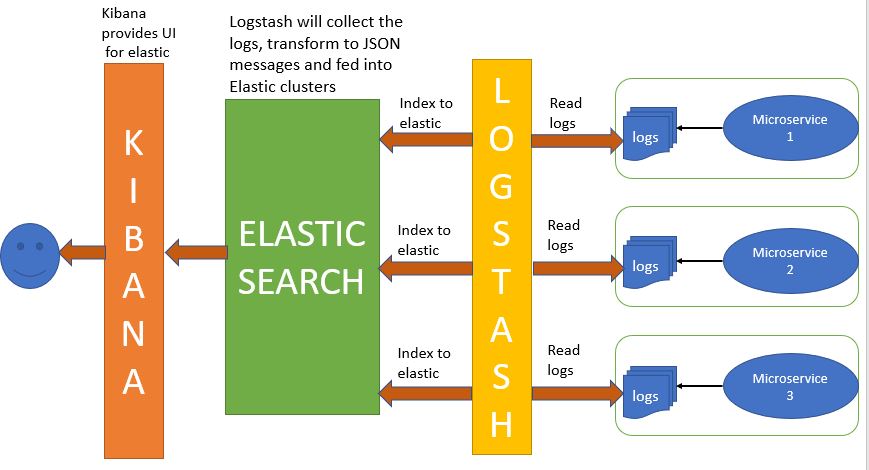
<https://www.javatpoint.com/java-8-features>

|  |  |  |  |
| --- | --- | --- | --- |
| Java 8  Lambda Expression 🡪code in functional style  Method Reference 🡪 Compact and es | Functional Interface  Stream API  Default Method |  |  |
|  |  |  |  |

Java Design Patterns

<https://www.javatpoint.com/design-patterns-in-java>

Microservices Current Project



To maintain the centralized logs of all the microservices then we use ELK

1. **Elasticsearch** is a NoSQL database that is based on the Lucene search engine.
2. **Logstash** is a log pipeline tool that accepts inputs from various sources, executes different transformations, and exports the data to various targets. It is a dynamic data collection pipeline with an extensible plugin ecosystem and strong Elasticsearch synergy
3. **Kibana** is a visualization UI layer that works on top of Elasticsearch.

**Distributed Transactions in Microservices**

1. We can achieve distributed transactions in Microservices based on
   1. 2 Phase commit
      1. The Coordinator first creates a global transaction with all the context information
      2. Prepare for the Microservice (Microservice name) 🡪 once the microservice is ready to perform the operations, then object is locked for further changes and tell the co-ordinator
      3. Prepare the second microservice is ready
      4. Once the coordinator confirm the all the microservices are ready, then prepare for commit, at this point all the services are unlocated.

Advantages 🡪 Strong consistency

Disadvantages 🡪 Locking the microservices that leads to latency

* 1. 3 Phase commit have additional step as Pre-Commit
  2. Type of Sega Pattern Implementation
     1. <https://dzone.com/articles/microservices-integration-patterns-saga-pattern>
     2. Chorography – Event based
        1. Event-Driven Communication Using an Event Bus
     3. Orchestration – Command based

**Distributed Log tracing using spring Cloud Sleuth and Zipkin**

Scenario: if the bank service is running on two different ports, 9090, 9091, if any of the microservice is down, it is very hard to find the which microservices is down.

To track microservice flow we use sleuth and zipkin

Spring cloud **sleuth** will produce meta data, that metadata consist of four elements

1. Service Name
2. Track Id 🡪 A unique ID that remains the same throughout the microservices for particular request
3. Span Id 🡪 Unique Id per microservices
4. Export Flags

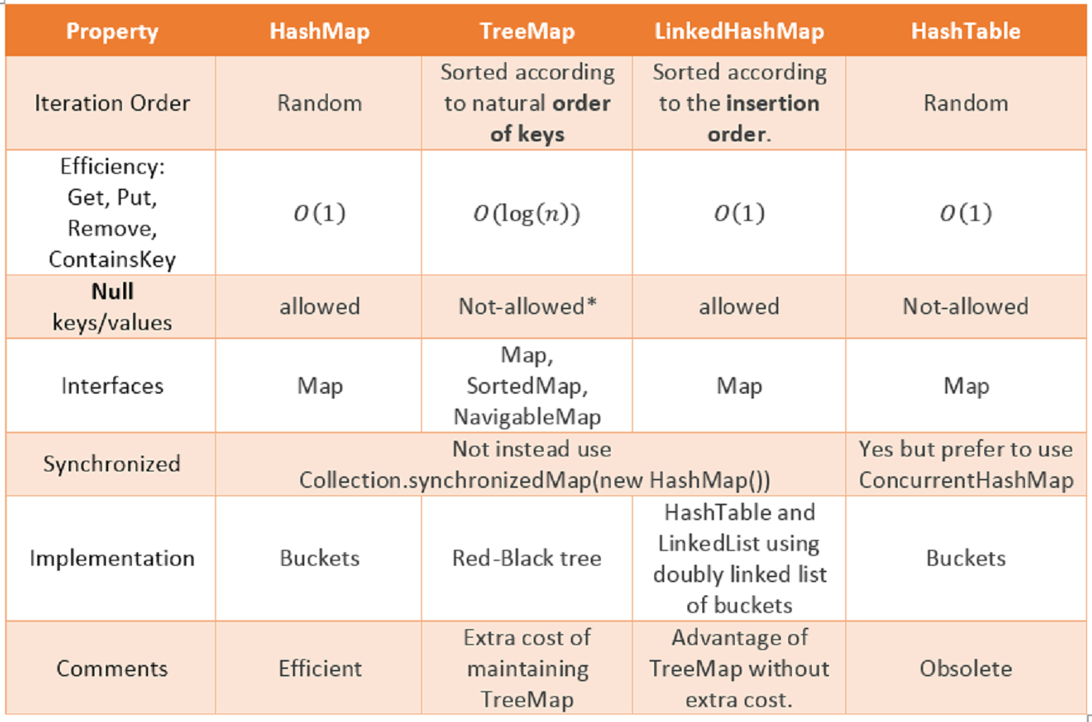
Zipkin server 🡪 start Zipkin microservice and register all the microservices with zipkin server (Zipkin is UI based)

Add dependencies to microservices **strater-zipkin**, and **starter-sleuth** and specify the zipkin where it is running, in application properties

Spring.zipkin.base-url:<Zipkin server url>

|  |  |  |  |
| --- | --- | --- | --- |
| **Factory Method**    Define an interface for creation of object, let the sub class decide on which class should be instantiate | **Abstract Factory**    With Abstract Factory design pattern we create a concrete implementation of a Factory.    E.g. DeviceFactory can be Abstract and it can give us GoogleDeviceFactory, AppleDeviceFactory etc. With AppleDeviceFactory we will get products like- iPhone, iPad, Mac etc. With GoogleDeviceFactory we will get products like- Nexus phone, Google Nexus tablet, Google ChromeBook etc. |  | **Builder**    We can use Builder pattern to create complex objects with multiple options.    E.g. when we have to create a Meal in a restaurant we can use Builder pattern. We can keep adding options like- Starter, Drink, Main Course, and Dessert etc.    Main feature of Builder pattern is step-by-step building of a complex object with multiple options. |
| **Prototype**    A fully loaded class should be copied or cloned. | **Singletone**    Ensure a class has one instance and provide a global point of access to it. | **Structural**  **Adapter**    Converts the interface of a class into another interface client expects | **Bridge**  Decouples an abstraction from its implementation so that the two can vary independently |
| **Composite**  Composes objects into tree structures to represent part-whole hierarchies | **Decorator**  Attaches the additional responsibility at run time | **Façade**  Provide a unified interface to a set of interfaces in a subsystem | **Flyweight**  Uses sharing to support large number of fine-grained objects efficiently |
| **Proxy**  Provides a surrogate or placeholder for another object to control access to it | **Behavioral Patterns**  **Chain of** |  |  |

Functional Programming advantages

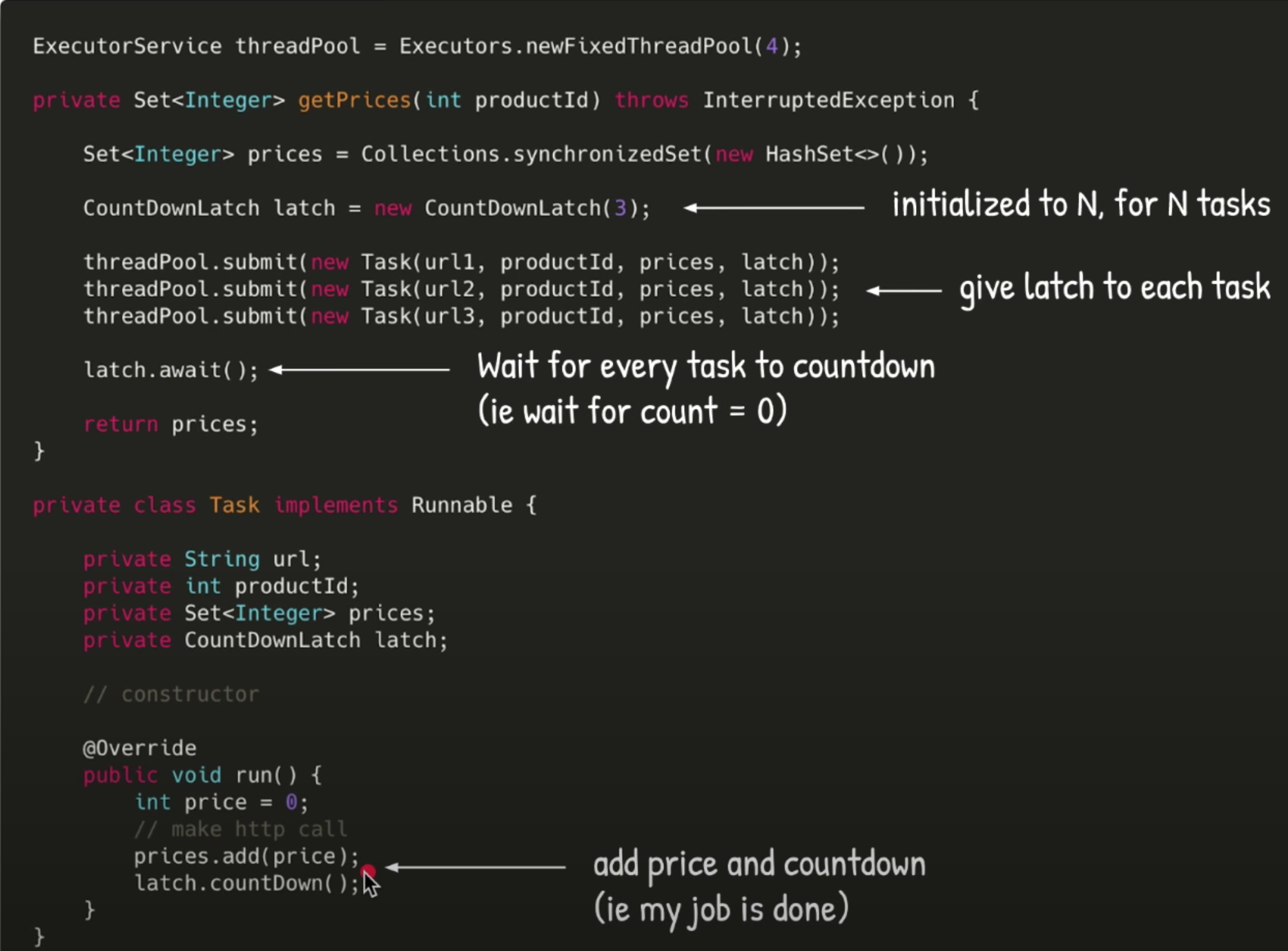


|  |  |  |  |
| --- | --- | --- | --- |
| **List vs Set vs Map**  List🡪 items in the list are ordered, accepts duplicates and nulls,  Set🡪un-ordered list, and will not accept any duplicates, and only one null value  Map->accepts key, pair values, and no duplicates values | **Array List Vs Vector**  Non synchronized  Grow by half size  Gives better performance  **Vector**  Synchronized  Double the size while resizing  Less performance | **Array List Vs LinkedList**  Search operation is fast in Array List,  Insert and update option is fast in linked list, linked list maintain double linked list | **HashMap vs Hash table**  HashMap is non synchronized, so it is not thread safe,  whereas Hash Table is synchronized so it is thread safe  HashMap implements linked HashMap |
| **HashMap vs TreeMap vs Linked Hashmap**  HashMap is implemented as a hash table, and there is no ordering on keys or values.    TreeMap is implemented based on red-black tree structure, and it is ordered by the key.  LinkedHashMap preserves the insertion order. Hashtable is synchronized in contrast to HashMap | **Hashset vs TreeSet vs LinkedHashset**  Hashset uses hashmap internally, and not maintain any order, gives better performance.  Treeset uses Treemap internally, maintains order based on supplier compitator,  LinkedHashSet maintains insertion order of elements  Performance similar to hashset | **Queue vs Priority Queue**  **Queue** is a list where insertion is done at one end and removal is done at the other end  **Priority Queue**, elements can be inserted in **any order** but removal of the elements is in a sorted order. | **Deque vs Array Deque**  The **Deque** interface supports insertion, removal and retrieval of elements at both ends  The **ArrayDeque** class is the resizable array implementation of the Deque interface |

|  |  |
| --- | --- |
| **Collections**  **Set List Queue**  **TreeSet Vector🡪 Stack PriorityQue**  **SortedSet ArrayList Deque**  **HashSet LinkedList** | Hashmap creates 16 buckets initially,  Once the threshold is increased to .75, then it will double the hash map,  Hash map calculate the hash code and index based on hashcode and n-1,  Hash key stores the info based on bucket index, if the key is the same then it will replace, else if there is already an element in that index it will stored the this to next to existing index  If the linked list threshold is greater based on “THREEIFY\_TRESHOLD” then it will convert to tree structure call Red-black tree, binary search tree, self-balancing tree |

|  |  |  |
| --- | --- | --- |
| **Process vs Thread**  self-contained execution environment  Thread is a single task of execution within the process | **Thread vs daemon thread**  When we create thread in java, it’s known as user thread. A daemon thread runs in background and doesn’t prevent JVM from terminating  setDaemon(true) | **How to create thread**  There are two ways to create thread  🡪 Runnable interface  🡪 extend the Thread class |
| **Thread state vs life cycle**  When we create thread then it is New.  Thread.start makes to Runnable.  Thread Scheduler is responsible to allocate CPU to threads in Runnable thread pool then change to Running. Other Thread states are Waiting, Blocked and Dead | **How to achieve thread safety in java**  synchronization,  atomic concurrent classes,  using volatile keyword,  using immutable classes | **volatile keyword**  directly reads data from memory and don’t cache it.  This makes sure that the value read is the same as in the memory. |
| **Thread Pool**  manages the collection of Runnable threads and worker threads execute Runnable from the queue.  We can use ExecutorService  and ThreadPoolExecutor  to achieve this | **How to avoid dead lock in Java**  Java Thread Dump, lists all the threads in JVM  VisualVM Profiler: analysing  application for slow ness  jstack**:-> Used for**  **Avoid Nested Locks, Lock only when it required, avoid waiting infinity** | **atomic operation**  single unit of task without interference, avoid data inconsistency. |
| Wait(), notify(), notifyall() |
| Bean Scope  Singleton – Single instance for IOC container  Prototype – any number of object instances  Request -- http Request, Application Context  Session – http session,  Global-Session – Http Global-Session, application context | Disable specific autoconfiguration  @SpringBootApplication(exclude = DataSourceAutoConfiguration.class) | Spring boot JAR to WAR  In pom.xml  <packaging>jar</packaging>  <packaging>war</packaging> |
| When we make any method with keyword synchronized, only one thread work on that method.  Private synchronized void counter() | @Bean – method-level annotation  @compenent – class level  @controller – web request handler  @RequestMapping  @Service – business logic  @Repository – DAO (Data Access Object)  that access database directly  @primary 🡪 to give higher preference to a bean when there are multiple beans of the same type.  @prototype🡪  @async🡪  @import 🡪  @async 🡪 | AOP🡪 we have non-functional requirements like  logging, profiling, caching, and transaction management  called cross cutting concerns  aspectJ  spring-boot-starter-aop  @Aspect 🡪 Class level  Type of advices @Before, @AfterReturning, @AfterThrowing, @After, @around. |
|  |  |  |

CountDownLatch 🡪 wait for the threads



|  |  |
| --- | --- |
| **Executer Service**  [**https://www.youtube.com/watch?v=6Oo-9Can3H8**](https://www.youtube.com/watch?v=6Oo-9Can3H8) |  |
| **Thread Pool Runnable/ Callable**  Runnable vs Callable  Runnable will Is not return any value  void run()🡪ExecuterService.**Execute**(  Callable<integer> have method call() will return value 🡪  Feature<Integer> feature= ExecuterService.**Submit**(new Task())  Integer = Feature.get() waits till the thread returns the value. |  |
| **Volatile vs Atomic vs Synchronization**  **Volatile 🡪** Generally solves the visibility problem, it will read the value form directly from memory instead of catch. It will read from the shared catch, instead of local cache |  |

<https://www.youtube.com/watch?v=sIkG0X4fqs4&list=RDCMUCiz26UeGvcTy4_M3Zhgk7FQ&index=2>

Fixed Thread Pool -> will fetch the tasks from the Q and executes one after the other

Cached Thread Pool 🡪 have synchronizes queue can hold only 1 task, if all the threads are busy then create a new thread and assign in the pool, it the thread is ideal for more than 60 seconds then kill the thread.

Scheduled Thread Pool 🡪 Schedules the tasks to run based on the time delay (and re-triggers for fixed rates / fixed delay), Delay Queue 🡪 in the quey it will change priority of thread based on the earliest to execute.

1. Schedule 🡪 Trigger after a certain delay
2. ScheduleAtFixedRate 🡪 Keep triggering after certain Time, run the task after previous task completed
3. ScheduleWithFixedDelay 🡪

Single Threaded Executer 🡪 size of the pool is only one, it will re-creates a new thread after any exception or thread killed

Steam API

Stream is a collection of objects, and stream iterates the internally, without explicit for each loop.

We can attach listeners to stream, these listeners are called stream iterators

We can obtain the stream from the java collections, we have intermediate operators and terminator operators in stream

|  |  |  |  |
| --- | --- | --- | --- |
| Intermediate 🡪  filter(Predicate<T>)  map(Function<T>)  flatMap(Function<T>)  sorted(Comparator<T>)  peek(Consumer<T>)  distinct()  limit(long n)  skip(long n) | Terminator operators 🡪 anyMatch,  allMatch, noneMatch, Collect, count, Findany, findFirst, foreach, min, max, reduce, toarray |  |  |

Functional Interface

|  |  |  |  |
| --- | --- | --- | --- |
| A *functional interface* in Java is an interface that contains only a single abstract (unimplemented) method. A functional interface can contain default and static methods which do have an implementation, in addition to the single unimplemented method. | Predicate 🡪 take single value return true/false,  (value) -> value != null; | Urinary Operator🡪 single parameter and returns a parameter of the same type  (person) -> { person.name = "New Name"; return person; }; | Binary Operator🡪 takes two parameters and returns a single value  (value1, value2) -> { value1.add(value2); return value1; }; |
| Function 🡪 Take single parameter and return single value  Function<Integer, Double> half = a -> a / 2.0; System.out.println(half.apply(10)); | Supplier 🡪 function that supplies a value of some sorts.  Supplier<Integer> supplier = () -> new Integer((int) (Math.random() \* 1000D)); | Consumer🡪 consumes the value without writing the value  Consumer<Integer> consumer = (value) -> System.out.println(value); |

MSSQL Interview questions

|  |  |  |  |
| --- | --- | --- | --- |
| **Primary Key Vs Unique Key**  Primary key – creates as cluster index, no null values  Unique Key creates non cluster index, accepts one null value | **Stored Procedure Vs Function**  Functions are computed values and cannot perform permanent environmental changes  A function can be used inline in SQL statements if it returns a scalar value or can be joined upon if it returns a result set | **Stored Procedure Vs Function** |  |
|  |  |  |  |
|  |  |  |  |

**JSON Web Token** (**JWT**) is the approach of securely transmitting data across communication channel. For **authentication** and authorization, it uses the technique of passing digitally signed **tokens**. **JWT** comprises of three parts: Header, Payloads and Signature

1. Decomposition Pattern
   1. Decompose by Business capability
   2. Decompose by Subdomain
   3. Strangler Pattern
2. Integration Pattern
   1. API Gateway Pattern
   2. Aggregator Pattern
   3. Client side UI Composition pattern
3. Database Pattern
   1. Database per service
   2. Shared database per service
   3. Common Query responsibility Segregation
   4. Saga Pattern
4. Observability Pattern
   1. Log Aggregation
   2. Performance Matrix
   3. Distributed Tracing
   4. Health Check
5. Cross cutting concern pattern
   1. External Configuration
   2. Service Discovery Pattern
   3. Circuit Breaker Pattern 🡪 Resilance4j 🡪
   4. Blue Green Deployment Pattern 🡪 it reduces the down time or no downtime at all
6. Sidecar, Chained Microservice, Branch Microservice, Event Sourcing Pattern, Continuous Delivery Patterns,
7. What are microservices 🡪 form of a service oriented architecture style applications are built as collection of smaller services rather than one single app.
8. Microservices Implementation
   1. Microservices (Discovery Clients)
      1. spring-boot-starter-web 🡪 (Acts as web application)
      2. spring-boot-starter-data-jpa🡪(Hybernate)
      3. spring-cloud-starter-netflix-eureka-client 🡪 Discovery Client
      4. spring-boot-devtools 🡪 pick the changes and restart the application
      5. modelmapper 🡪 maps one class to another class
      6. spring-boot-starter-security 🡪
      7. jackson-dataformat-xml
      8. jjwt
      9. spring-cloud-starter-config 🡪 Centralize the configuration
      10. spring-cloud-starter-bus-amqp 🡪 Spring Cloud Bus 🡪 once the configuration changes are updated
      11. spring-boot-starter-actuator 🡪
      12. mysql-connector-java
      13. spring-cloud-starter-openfeign 🡪 Communicate each other with writing the client code
      14. resilience4j-circuitbreaker 🡪
          1. Open->failure > threshold (deny), close🡪then allow the request, half-open🡪 only few calls
      15. spring-cloud-sleuth-zipkin 🡪 Distributed log tracing in microservices
   2. Discovery Server (Discovery Server)
      1. spring-cloud-starter-netflix-eureka-server
      2. spring-boot-starter-security
   3. API Gate way
      1. spring-boot-starter-web
      2. spring-cloud-starter-netflix-eureka-client
      3. spring-cloud-starter-netflix-zuul
      4. spring-boot-starter-security
      5. spring-cloud-starter-config
      6. spring-cloud-starter-bus-amqp
      7. spring-boot-starter-actuator
   4. API Config Server
      1. spring-cloud-config-server
      2. spring-cloud-starter-bus-amqp
      3. spring-boot-starter-actuator

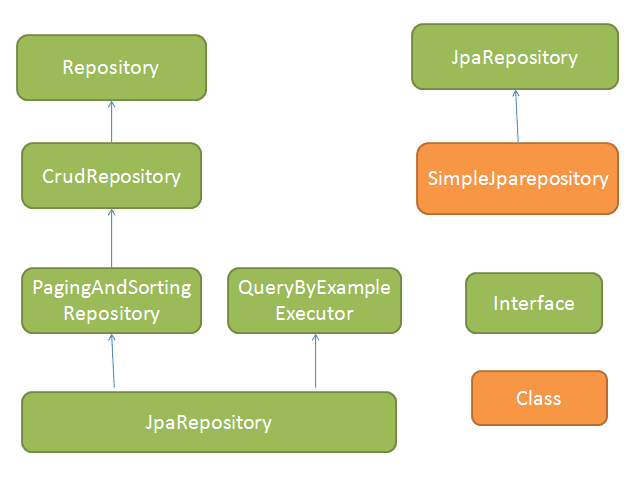
**Spring Boot**

**How to store passwords**

1. **AWS Secret manager**
2. **Vault from**

JPA interview questions

<https://www.netsurfingzone.com/jpa/spring-data-jpa-interview-questions-and-answers/>



**JPA / Hibernate**

|  |  |  |  |
| --- | --- | --- | --- |
| **NamedQuery Vs NamedNativeQuery**  @NamedQuery 🡪 Hibernate Query Language (HQL) or Java Persistence Query language  @NamedNativeQuery🡪 |  | **How to create custom repository**  Extend any interface like JPARepository/CurdRepository  findBy or getBy  or queryBy or countBy or readBy | JPARepository vs CURDRepository  JPARepositry extened CURD, PagingAndSortingRepository |
| @Controller and @RestController 🡪 included Controller + ResponeBody |  | @OneToOne – Default EAGER @OneToMany – Default fetch type is LAZY. @ManyToOne – Default fetch type is EAGER. @ManyToMany – Default fetch type is LAZY. @ElementCollection – Default fetch type is EAGER. |  |

**@NamedNativeQuery** lets you write a named SQL query, while **@NamedQuery** lets you write a named HQL query (or JPQL). In general, you should prefer to write HQL queries because then you can let Hibernate handle the intricacies of converting the HQL into the various SQL dialects

| **Scope** | **Description** |
| --- | --- |
| [singleton](https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html#beans-factory-scopes-singleton) | Scopes a single bean definition to a single object instance per Spring IoC container. |
| [prototype](https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html#beans-factory-scopes-prototype) | Scopes a single bean definition to any number of object instances. |
| [request](https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html#beans-factory-scopes-request) | Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext. |
| [session](https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html#beans-factory-scopes-global-session) | Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext. |
| [global session](https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html#beans-factory-scopes-global-session) | Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext. |

**Spring Cloud Config**

1. Private Git Repository for all config application properties
2. Spring Boot application retrieves the API keys stored in AWS Secret Manager by using the secret name and its region.
3. Aws-java-sdk-secretManager

Microservices Communication

1. Synchronous HTTP communication
2. Asynchronous communication over AMQP

**12 Factor App**

[**https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser**](https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser)

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Code base 🡪 one codebase tracked in revision control, many deploys   One version control per application and deploy for multiple environment  In Microservices, every service should have its own codebase. Having an independent codebase helps you to easy CI/CD process for your applications. | 1. Dependencies 🡪 Explicitly declare and isolate dependencies   **Microservices:** All the application packages will be managed through package managers like sbt, maven.  In non-containerized environments, you can go for configuration management tools like chef, ansible, etc. to install system-level dependencies.  For a containerized environment, you can go for dockerfile | 1. Config 🡪 Store Config in the environment   **Microservices:**Externalize the configurations from the application. In a microservice service environment, you can manage the configurations for your applications from a source control like git (spring-cloud-config) and use the environment variables to not to maintain the sensitive information in the source control. | 1. Backing Service 🡪 Treat backing services as attached resources   Any kind of service that application consume, database, catching services should be attached as single entry point, url stored as environment variable, that can be changed based on the environment |
| 1. Build, release, run 🡪 Strictly separate build and run stages   You can use CI/CD tools to automate the builds and deployment process. Docker images make it easy to separate the build, release, and run stages more efficiently. | 1. Process 🡪 Execute app as one or more stateless process   **Microservices:**By adopting the stateless nature of REST, your services can be horizontally scaled as per the needs with zero impact. If your system still requires to maintain the state use the attached resources (redis, Memcached, or datastore) to store the state instead of in-memory. | 1. Port binding 🡪 Export services via port binding   **Microservices:**Spring boot is one example of this one. Spring boot by default comes with embedded tomcat, jetty, or undertow. | 1. Concurrency 🡪 Scale out via the process model   **Microservices:** By adopting the containerization, applications can be scaled horizontally as per the demands. |
| 1. Disposability 🡪 maximize robustness with fast start-up and gracefull shutdown | 1. Dev/Prod parity 🡪Keep development, stating and production as similar as possible | 1. Logs 🡪 Treat logs as event stream   Implement the ELK, Elastic Search, Logstash, Kibena | 1. Admin 🡪 Run admin/management tasks as one-off process. |

**Distributed Catching**

[**https://hazelcast.com/blog/architectural-patterns-for-caching-microservices/**](https://hazelcast.com/blog/architectural-patterns-for-caching-microservices/)

**Integration Test in Spring Boot**

Classlevel --> @SpringBootTest with RANDOM\_PORT

MethodLevel 🡪 @Test

**TestRestTemplate** to test the Rest API

TestRestTemplate have getForOjbect, PostForObject, exchange

Json output is verified by using JSONAssert

Java 8/11, Spring Boot, Junit / Mockito, Karate / Gherkin, MariaDB, Kafka / Avro, git, Pivotal Cloud Foundry, Jenkins

<https://www.baeldung.com/mockserver>

Mock Server 🡪 Add dependencies mockserver-netty, mockserver-client-java

the tool can:

* generate and return fixed responses
* forward a request to another server
* execute callbacks
* verify a request

1. We can use the *startClientAndServer()* Java API to start the server
2. **private** ClientAndServer mockServer;
3. @BeforeClass **public** **void** **startServer**() { mockServer = startClientAndServer(1080); }
4. @AfterClass **public** **void** **stopServer**() { mockServer.stop(); }

*MockServerClient*API 🡪 **new** MockServerClient

Karate / Gherkin / JBehave

<https://www.baeldung.com/karate-rest-api-testing>

Karate BDD 🡪 add dependencies, karate-apache, karate-junit

1. Testing the status code test.feature, test.java 🡪@runwith(karate.class)

Scenario: Testing valid GET endpoint

Given url 'http://localhost:8097/user/get'

When method GET

Then status 200

Scenario: Testing the exact response of a GET endpoint Given url 'http://localhost:8097/user/get' When method GET Then status 200 And match $ == {id:"1234",name:"John Smith"}

Testing Response

Scenario: Testing the exact response of a GET endpoint

Given url 'http://localhost:8097/user/get'

When method GET

Then status 200

And match $ == {id:"1234",name:"John Smith"}

And match $.id == "1234"

Scenario: Testing that GET response contains specific field

Given url 'http://localhost:8097/user/get'

When method GET

Then status 200

And match $ contains {id:"1234"}

### ****Validating Response Values With Markers****

*#null, #notnull, #boolean*, *#number*, *#string, #array*, *#object,* #uuid, #regex, #? EXPR

Scenario: Testing a POST endpoint with request body

Given url 'http://localhost:8097/user/create'

And request { id: '1234' , name: 'John Smith'}

When method POST Then status 200

And match $ contains {id:"#notnull"}

Rest VS SOAP