**Q. what are the different feature of Java 8?**

Ans:-Java 8 comes into picture for the functional programming approach. To Promote functional programming approach it comes with different feature.

1. Lambda Expression:- It is a concise way to represent anonymous function using functional interface. Function interface is nothing but an interface which has only one abstract method in it. It should be annotated with @FuctionalInterface annotation.

2.Stream API:- Stream API is the most powerful tool of java 8 as it is use to process the collection API. Using Stream API we can filter ,map, collect,reduce the collections.

3.Default and static method:- Default and static method in interface is introduced in the java 8. Using these method we can define the method inside the interface.

4.Method reference:- Method reference is another feature of java 8 using method reference we can easily call the method of the class using :: colon we just need to specify the class name after that we just need to use :: and we need to write the method name to call it. It is like shorthand to call the method.

5.Optional:- It is a kind of container object which may or may not contain non-null values. It is designed to avoid null pointer exceptions in java programs.

**Q:- Can one Functional interface extend another functional interface?**

Ans:-yes, Functional interface can extend another functional interface provided parent class has only one abstract method and child class has only default or static method. It shouldn’t have any other abstract method in it.

**Q:-What is Function Programming?**

Ans:-It’s a pattern that treats computation as the evaluation of mathematical function and avoids changing the state and mutable data.

If you see stream, in will perform operation one time if we try to use same stream it’s not going to happen and in stream it didn’t change the value of the existing variable it takes the value from the variable and execute the operation and stores inside their own heap area until and unless we need to collect and need to store in other variable.

**Q. Why default and static method introduced in java 8?**

Ans:- Default method is introduced for backward compatibility, backward compatibility means suppose there is some change in the interface then we need to write in the implemented class. It will create extra line of code in implemented class which is working properly so we can directly write in the default method in the interface which can be used in the future.

Static block is like a initialisation block in the interface which is use to define the method in the interface that is only be used in the interface and we know that static method are class method so these method are introduced so that it is only be used for the purpose of the interface.

**Q) What is FlatMap and how it is different from map?**

Ans:-Map is used for data transformation, it works on entire stream and gives one output for one input, it mean if you give one stream it will provide one stream. It mean it will do one to one operation

Suppose

input[2,3,4,5]-->[2,3,4,5]

But Flat Map is combination of flat and map it also does data transformation but suppose the is stream of stream like list of list if we do Flat map operation it will provide one stream from stream of stream input like

Suppose

Input ->[[2,3],[3,4]] output->[2,3,3,4]

**Q) What is Multi Threading?**

Ans:-Multi Threading in java is like multiple worker in a single factory which does different task at the same time so in case of java multiple threads do different sub-processes which is independent of each other.

**Q) What are different types of Thread in java?**

Ans:-1. Main Thread- This thread is a thread which is managed by the application code itself.

2.Daemon Thread:- This Thread looks after the auxiliary work like garbage collection and other thing it has less priority

3.Slave/worker thread :- This thread is created if we want to do some subprocess which are not related with other processes.

**Q)What are the different stage of the Thread?**

Ans:-A) New B)Runnable C)Running(wait,sleep, notify) d) terminate

**Q What are reflection api**

Ans:- In java, the Reflection api is a set classes in the java.lang.reflect package that allows a java program to inspect and manipulate itself at runtime. Method used in reflection api are getclass , Class.forName. isInstance, of Instance.

**Q) can we overload run method in multi threading?**

ans:-You can overload run() method in multithreading in java. In java we can achieve multiThreading

⇒ By Extending Thread

⇒ By implementing Runnable

Runnable interface has only one method i.e. run method and thread class extend runnable method and internally start method of the Thread class use the run method.

So whenever we overload the run method , the overloaded run method will acts like an normal method as the signature of it will be different.

**Q How JVM memory management work?**

Ans:-

Java Virtual Machine (JVM) memory management is a vital process that ensures efficient allocation, use, and recycling of memory during the execution of Java applications. It consists of several key components and processes:

**1. JVM Memory Structure:**

The JVM memory is divided into several regions:

* **Heap Memory:** Used to store objects and class instances. It is the primary area for dynamic memory allocation.
  + *Young Generation:* Stores short-lived objects and is further divided into:
    - **Eden Space**: Where new objects are created.
    - **Survivor Spaces (S0 and S1)**: Where objects are moved after surviving a garbage collection cycle in Eden.
  + *Old Generation:* Stores long-lived objects.
* **Method Area (Metaspace in Java 8+):** Stores class metadata, static variables, and method data.
* **Stack Memory:** Stores method call frames, including local variables and partial results.
* **Program Counter (PC) Register:** Holds the address of the currently executing instruction.
* **Native Method Stack:** Used for native (non-Java) method calls.

**2. Garbage Collection (GC):**

Garbage Collection is an automatic process by which the JVM reclaims memory by destroying unreachable objects. Common GC algorithms include:

* **Serial GC:** Single-threaded collector suitable for small applications.
* **Parallel GC:** Uses multiple threads for minor collections.
* **CMS (Concurrent Mark Sweep):** Reduces pause times by performing most work concurrently.
* **G1 (Garbage First) GC:** Divides heap into regions and collects in parallel; suitable for large heap sizes.

**3. Memory Allocation Process:**

* New objects are created in the Eden space.
* When Eden fills up, a Minor GC occurs, and surviving objects are moved to Survivor spaces.
* After multiple GCs, long-lived objects are promoted to the Old Generation.
* A Major GC (or Full GC) cleans the Old Generation.

**4. Memory Leaks in JVM:**

Even with automatic GC, memory leaks can occur if references to unused objects are unintentionally maintained. Tools like VisualVM, JConsole, and heap dump analyzers help identify leaks.

**5. JVM Tuning:**

Memory management can be optimized using JVM options:

* -Xms and -Xmx: Set initial and maximum heap size.
* -XX:NewSize and -XX:MaxNewSize: Control young generation size.
* -XX:+UseG1GC: Use G1 garbage collector.

**Q What is Garbage collection and can we implement garbage collection manually in java?**

Ans:- **Garbage Collection in Java**

Garbage Collection (GC) in Java is the process by which the JVM automatically deallocates memory by removing objects that are no longer reachable in the application. This helps in preventing memory leaks and reducing the burden of manual memory management on developers.

**How It Works:**

* Every time an object is created using the new keyword, it is stored in the heap memory.
* If no live thread can access an object, it is considered unreachable.
* The GC identifies and removes these unreachable objects to free up memory.

**Types of Garbage Collectors in Java:**

* **Serial GC**: Uses a single thread for both minor and major GC events.
* **Parallel GC**: Uses multiple threads for minor collections, improving performance for multi-core processors.
* **Concurrent Mark Sweep (CMS)**: Aims for low-latency by performing most GC steps concurrently.
* **G1 Garbage Collector**: Designed for applications running on multi-processor machines with large memory spaces.

Yes we can implements garbage collection manually using **System.gc() and finally()** methods

public class ManualGCExample {

public static void main(String[] args) {

// Create a large number of objects

for (int i = 0; i < 1000000; i++) {

DummyObject obj = new DummyObject(i);

}

// Nullify reference to encourage GC

System.out.println("Objects created. Suggesting garbage collection...");

// Suggesting garbage collection manually

**System.gc();**

System.out.println("Garbage collection requested.");

}

}

class DummyObject {

private int id;

private int[] largeArray = new int[1000];

public DummyObject(int id) {

this.id = id;

}

@Override

protected void finalize() throws Throwable {

// Called by GC before object is removed from memory

System.out.println("Finalizing object with id: " + id);

}

}

**Q.Does Finally executes every time , weather there is an error,return statement or system.exit() in a java code?**

**Ans:-**

public class FinallyBlockExample {

public static void main(String[] args) {

try {

System.out.println("Inside try block");

// Uncomment one line at a time to test different scenarios

// return;

// int result = 10 / 0; // Throws ArithmeticException

// System.exit(0); // Terminates JVM, finally will NOT execute

} catch (Exception e) {

System.out.println("Inside catch block");

} finally {

System.out.println("Inside finally block");

}

System.out.println("End of program");

}

}

/\*

Output Analysis:

1. If 'return;' is used => 'finally' executes before returning.

2. If an exception is thrown => 'finally' executes after catch.

3. If System.exit(0) is used => 'finally' does NOT execute.

4.If error occur finally will not executes.

\*/

**Q Explain the role of class loader in java, including bootstrap extensible and system class loader**

**Ans:-** Class loader is the part of JRE(Java Runtime Environment) which dynamically load the class into the JVM during runtime.

There are various types of class loaders like :-

1. **BootStrap Class Loader:-**Load the class from the JDK’s rt.jar(java.lang…,java.util)
2. **Extension Class Loader:-**It load the class from the jar which is external like (mysql libraries)
3. **System Class Loader:-**Load the class from the system.

**Q.Can we call the not static method from static method or static block?**

**Ans:-** No, we can’t call the non static variable from the static method as static method are loaded only once during the loading of the class where as non static variable are loader after object creation. So we can’t call non-static method from static method.

**Q. Discuss the purpose of static method for memory management?**

Ans:- In Java, a **static method** belongs to the class rather than to instances of the class.

**1. Memory Efficiency**

* Static methods are stored in the **Method Area (or Metaspace in Java 8+)**.
* Only **one copy** of the static method is maintained per class, regardless of how many instances are created.
* This helps reduce memory footprint compared to instance methods, which may involve creating multiple object references.

### ****2. Shared Behavior Across Instances****

* Since static methods are shared among all instances, they can provide **shared functionality**, such as logging or configuration management, without redundant object storage.

**Q Why Character-Array is used for password instead of String?**

Ans:-Character-Array is used in password instead of string because character-array are mutable in nature where as string are immuatable in nature

Suppose we have use string literals as password then the value of string will be stored in SCP until the application is closed which leads to some hacking issue.

**Q) can you describe the life cycle of the bean?**

Ans:- Spring manages the entire lifecycle of a bean, from creation to destruction. This lifecycle involves several steps and callback methods:-

* 1. Instantiation
  2. Populate properties
  3. Bean name Awareness
  4. Bean factory awareness
  5. Application context awareness
  6. Initializing Bean
  7. Disposable bean

1 Instantiation:-

🡺Spring creates the bean instance using the constructor(no-args or constructor injection)

🡺Done based on bean definition in XML annotation

2. Populate properties

🡺Injects values and references into the bean’s properties

🡺Include @Autowired, constructors args.

3. Bean Name Awareness

🡺if the bean implements Bean name Awareness, Spring call setBeanName(String beanName) with the bean id

4. Bean Factory Awareness:-

🡺 The Bean factory Awareness interface is a part of the spring framework’s infrastructure, allowing beans to be aware of and interact with the container that created them.

5. AppicationContextAwareness

🡺The main responsibility of the applicationContextaware interface in spring is to provide a bean with access to the spring application context that manages it.

6. InitializingBean

🡺called after all properties are injected.

7.Disposable Bean

🡺 called before the bean in destroyed.

**Q) Describe the different annotation used in spring/spring boot?**

**Ans:-**

Core java annotation

**1. @Component:-**Marks a java class as spring component.

2. **@Controller:-** Marks a java class as a Spring MVC

3. **@Service:-** Marks a java class as a service layer component

4. **@Repository:-** Marks a java class as a data access object(Dao

5. **@Configuration:** Indicates that a class declares one or more @Bean methods.

6. **@Bean:** Indicates that a method produces a bean to be managed by the Spring container.

7. **@Autowired:** Used for automatic dependency injection

8. **@Qualifier:** Specifies which bean should be injected when multiple candidates are present.

9. **@Value:** Injects values into fields, method parameters, or constructor arguments.

10. **@Scope:** Configures the scope of a bean (e.g., singleton, prototype)

11. **@Lazy:** Marks a bean to be lazily initialized.

12. **@Primary:** Indicates that a bean should be given preference when multiple candidates are qualified.

**Spring MVC Annotation:-**

1. **@RequestParam**: Binds a web request parameter to a method parameter

**http://localhost:8080/greet?name=Gaurav**

**@GetMapping("/greet")**

**public String greetUser(@RequestParam String name) {**

**return "Hello, " + name;**

**}**

1. @PathVariable: Binds a URI template variable to a method parameter.
2. **@RequestHeader: Binds a method parameter to a request header.**
3. **@CookieValue: Binds a method parameter to a cookie value.**
4. **@SessionAttribute: Binds a method parameter to a session attribute.**
5. **@ModelAttribute: Binds a method parameter or return value to a named model attribute and exposes it to a web view.**
6. **@InitBinder: Identifies methods which initialize the WebDataBinder, which are used for customizing request parameter binding.**
7. **@ExceptionHandler: Defines the method that handles exceptionsthrown by request-handling methods**

Spring Boot Annotations

1. **@SpringBootApplication:** Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes.

2. **@EnableAutoConfiguration:** Enables Spring Boot’s auto-configuration mechanism.

3. **@ComponentScan:** Configures component scanning directives for use with @Configuration classes.

4. **@ConfigurationProperties:** Binds the properties defined in the external configuration files to the fields in a Java class.

5. **@SpringBootTest:** Used to bootstrap the entire container for integration tests.

6. **@TestConfiguration:** Indicates that a class declares one or more @Bean methods to be used in tests.

7. **@RestController:** Combines @Controller and @ResponseBody, simplifying the creation of RESTful web services.

8. **@RequestScope:** A specialized version of @Scope for a single HTTP request lifecycle.

9. **@SessionScope:** A specialized version of @Scope for a HTTP session ifecycle.

10. **@ApplicationScope:** A specialized version of @Scope for a web application lifecycle.

**Q.What is markerInterface? How it is useful in java?**

**Ans :-** A marker interface is a interface which don’t have any method or field. It is generally used in the class to give information to java about the properties that java class may implements like suppose if java class implements Serializable and if we use serialisation in the class then first JVM will check if it implements Serializable. If it implements Serializable it will check all the properties of serialisation and implements in the class.

**Q. How to handle exception in spring boot ?**

**Ans:-** In Spring boot exception is handled globally be using :-

1. **Globally using @ControllerAdvice**

Use @ControllerAdvice with @ExceptionHandler to handle exceptions across the whole application.

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleAll(Exception ex) {

return new ResponseEntity<>("Internal Server Error: " + ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

}}

2. **Per-Controller using @ExceptionHandler**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

public User getUser(@PathVariable int id) {

if (id < 1) {

throw new IllegalArgumentException("Invalid user ID");

}

return new User(id, "John");

}

@ExceptionHandler(IllegalArgumentException.class)

public ResponseEntity<String> handleBadId(IllegalArgumentException ex) {

return ResponseEntity.badRequest().body(ex.getMessage());

}}

**Q. What is response entity in Spring boot?**

**Ans :-**Response entity is a generic HTTP response wrapper provided by spring. It provides full control over:-

* HTTP status code
* Response header
* Response body

@GetMapping("/hello")

public ResponseEntity<String> sayHello() {

return new ResponseEntity<>("Hello, World!", HttpStatus.OK);

}

**Q Difference between configuration vs component?**

**Ans** @configuration is used in java based configuration to declare bean method in it. It is like java based replacement for xml configuration

@component is used to marked the spring based component(bean).It is auto detected during component scan(@component Scan)

**Q what is try with resource and when we need to use it?**

**Ans:-** Try-with-resources is a feature introduced in Java 7 that automatically closes resources (like files, database connections, input/output streams) when you're done using them — no need to manually close them in a finally block.

Like

try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {

String line;

while ((line = br.readLine()) != null)

System.out.println(line);

} catch (IOException e) {

e.printStackTrace();

}

**Q what changes occur in hashmap after java 8?**

**Ans:-** Before java 8 , In hashmap if hashcode of two key is same it will lead to collision, after java 8 optimization is made to reduce the collision.

**Before Java 8**:

* Each bucket was a **linked list** of entries.
* Performance degraded to **O(n)** for heavily collided keys.

**Since Java 8**:

* If too many collisions occur in a bucket, the **linked list is converted to a balanced binary tree** (specifically, a **Red-Black Tree**).
* Lookup time improves to **O(log n)** instead of O(n).

| Condition | Structure Used in Bucket |

| --------------------------- | -------------------------------------- |

| ≤ 8 entries | LinkedList |

| ≥ 8 entries & capacity ≥ 64 | Red-Black Tree (treeify) |

| < 6 entries after resizing | Convert back to LinkedList (untreeify) |

**Q. what is internal working of HashSet?**

**Ans:** HashSet is a collection that **stores unique elements** and is backed internally by a **HashMap**.

HashSet is just a **wrapper over HashMap** where:

* **Keys** = your set elements
* **Values** = a dummy constant (same for all)

**Q:-Diff between HashMap vs Treemap?**

**Ans:-**

| **Feature** | **HashMap** | **TreeMap** |
| --- | --- | --- |
| **Ordering** | No order guaranteed | Sorted by **natural order** or **custom comparator** |
| **Underlying structure** | Hash Table (Array + LinkedList/Tree) | Red-Black Tree (self-balancing BST) |
| **Performance (avg case)** | O(1) for get(), put() | O(log n) for get(), put() |
| **Null keys/values** | 1 null key, multiple null values | ❌ No null key, ✅ multiple null values |
| **Use case** | Fast access without ordering | Sorted data (ascending order) |
| **Thread-safe?** | ❌ Not thread-safe | ❌ Not thread-safe |

**Q:- Why null value are not allowed in TreeSet?**

**Ans:-** Because TreeSet uses **natural ordering** (or a **custom comparator**) to sort elements, and comparing null with other objects using compareTo() or Comparator.compare() results in a NullPointerException.

Q:- **How does load balancing work in microservices?**

Ans:- In a **microservices architecture**, **load balancing** ensures that incoming network traffic (e.g. API requests) is **evenly distributed** across multiple instances of microservices to:

* Avoid overload on a single service
* Improve fault tolerance
* Maximize performance and availability

**Different types of load balancing:-**

| **Type** | **Description** | **Example Tools** |
| --- | --- | --- |
| **Client-side Load Balancing** | The client decides which server to call based on a service registry | Spring Cloud LoadBalancer, Ribbon (deprecated) |
| **Server-side Load Balancing** | A dedicated server (reverse proxy or hardware) routes traffic | NGINX, HAProxy, AWS ALB |
| **API Gateway Load Balancing** | Gateway does intelligent routing and load distribution | Kong, Zuul, Spring Cloud Gateway |
| **DNS Load Balancing** | DNS returns different IPs on each query | Route53, Cloudflare DNS |
| **Container-level (Kubernetes)** | K8s service forwards traffic to pods using iptables/ipvs | kube-proxy, Istio |

**Based on traffic distribution strategy:**

| **Strategy** | **Description** |
| --- | --- |
| **Round Robin** | Each request goes to the next available instance in order. |
| **Random** | Requests are assigned to random instances. |
| **Least Connections** | Request goes to the instance with the fewest active connections. |
| **Weighted Round Robin** | Some servers get more traffic based on assigned weight (e.g. stronger machines). |
| **IP Hash** | Client’s IP determines the server (ensures session stickiness). |
| **Least Response Time** | Chooses server that responded fastest to previous requests. |
| **Geolocation-based** | Routes users to the nearest data center or region. |