

1. What are the different features of java 8?

Answer:

Certainly! Here's a concise list of the main features introduced in Java 8, along with brief descriptions:

- Lambda expressions,
- Method references,
- Functional interfaces,
- Stream API,
- Default methods,
- Base64 Encode Decode,
- Static methods in interface,
- Optional class,
- Collectors class,
- ForEach() method,
- Nashorn JavaScript Engine,
- Parallel Array Sorting,
- Type and Repating Annotations,
- IO Enhancements,
- Concurrency Enhancements,
- JDBC Enhancements etc.

For description refer:

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

2. What is lambda expression and why we should use that

Answer:

Definition:

A lambda expression in Java is a concise way to represent an anonymous function (a function without a name) that can be passed around as if it were an object. Lambda expressions provide a clear and concise way to represent one method interface using an expression.

Syntax:

The syntax of a lambda expression is:

(parameters) -> expression

Example:

// Traditional way using an anonymous class

```
Runnable runnable = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Hello, World!");  
    }  
};
```

// Using a lambda expression

```
Runnable lambdaRunnable = () -> System.out.println("Hello, World!");
```

## Why Use Lambda Expressions?

- **Conciseness:** Lambdas significantly reduce code size compared to traditional anonymous inner classes, especially for short functions.
- **Improved Readability:** In some cases, lambdas can enhance code readability by keeping related logic close to where it's used.
- **Functional Programming Style:** Lambdas promote a more functional programming style in Java, where you work with functions as first-class objects. This can lead to more declarative and composable code.
- **Integration with Streams and collections:** Lambda expressions work seamlessly with the Streams API and collection operations, enabling you to perform bulk operations on collections more efficiently and elegantly.

3. What is stream in java 8?

Answer:

In Java 8, streams provide a declarative and efficient way to process collections of elements. They offer a powerful alternative to traditional for loops and iterators, especially for working with large datasets.

Stream Pipeline:

A stream pipeline consists of a source, zero or more intermediate operations, and a terminal operation.

Source: The source can be a collection, array, I/O channel, or generator function.

**Intermediate Operations:** These operations transform a stream into another stream. They are lazy and return a new stream. Examples include filtering, mapping, sorting, and more. These operations are typically chained together to create a processing pipeline.

**Terminal Operation:** A terminal operation marks the end of the stream pipeline and produces a result or performs an action on the elements. Examples include counting elements, collecting them into a new collection, or printing them. Once a terminal operation is called, the stream is consumed and cannot be reused.

### Benefits of Using Streams:

- **Declarative Style:** Streams encourage a declarative programming style, where you specify what you want to achieve with the data rather than explicitly writing imperative code (e.g., for loops) that dictates how to process it.
- **Improved Readability:** Stream operations can often improve code readability by making the processing logic more concise and easier to follow.
- **Parallel Processing:** Streams can take advantage of multiple cores on modern processors for parallel processing, potentially leading to significant performance gains for large datasets.
- **Immutability:** Stream operations generally don't modify the original data source. This promotes immutability, which can make your code more predictable and easier to reason about

4. What are the different annotations in spring boot project you have used ?

Answer:

used annotations in Spring Boot:

Core Spring Annotations:

1. **@SpringBootApplication:** A convenience annotation that combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`. It is typically used to mark the main class of a Spring Boot application.
2. **@Configuration:** Indicates that a class can be used by the Spring IoC container as a source of bean definitions.
3. **@ComponentScan:** Configures component scanning directives for use with `@Configuration` classes.
4. **@EnableAutoConfiguration:** Enables Spring Boot's auto-configuration mechanism, which attempts to automatically configure your Spring application based on the jar dependencies you have added.

Component Stereotype Annotations:

5. **@Component:** Indicates that a class is a Spring component.

6. `@Service`: Indicates that a class is a service, a specialization of `@Component`.
7. `@Repository`: Indicates that a class is a repository, which is a mechanism for encapsulating storage, retrieval, and search behavior, a specialization of `@Component`.
8. `@Controller`: Indicates that a class serves the role of a controller in the MVC pattern.
9. `@RestController`: A convenience annotation that combines `@Controller` and `@ResponseBody`. It is typically used to create RESTful web services.

#### **Dependency Injection Annotations:**

10. `@Autowired`: Marks a constructor, field, setter method, or configuration method to be autowired by Spring's dependency injection facilities.
11. `@Qualifier`: Specifies which bean should be autowired when there are multiple beans of the same type.
12. `@Primary`: Indicates that a bean should be given preference when multiple candidates are qualified to autowire a single-valued dependency.

#### **Spring Data JPA Annotations:**

13. `@Entity`: Specifies that the class is an entity and is mapped to a database table.
14. `@Table`: Specifies the table in the database with which the entity is mapped.
15. `@Id`: Specifies the primary key of an entity.
16. `@GeneratedValue`: Provides for the specification of generation strategies for the values of primary keys.
17. `@Column`: Used to specify the mapped column for a persistent property or field.
18. `@Repository`: Indicates that the class is a Spring Data repository, often used in conjunction with Spring Data JPA.

#### **Spring MVC Annotations:**

19. `@RequestMapping`: Provides routing information and specifies that an HTTP request should map to a method.
20. `@GetMapping`: A shortcut for `@RequestMapping(method = RequestMethod.GET)`, mapping HTTP GET requests onto specific handler methods.
21. `@PostMapping`: A shortcut for `@RequestMapping(method = RequestMethod.POST)`, mapping HTTP POST requests onto specific handler methods.
22. `@PutMapping`: A shortcut for `@RequestMapping(method = RequestMethod.PUT)`, mapping HTTP PUT requests onto specific handler methods.
23. `@DeleteMapping`: A shortcut for `@RequestMapping(method = RequestMethod.DELETE)`, mapping HTTP DELETE requests onto specific handler methods.

24. `@PatchMapping`: A shortcut for `@RequestMapping(method = RequestMethod.PATCH)`, mapping HTTP PATCH requests onto specific handler methods.

25. `@PathVariable`: Indicates that a method parameter should be bound to a URI template variable.

26. `@RequestParam`: Indicates that a method parameter should be bound to a web request parameter.

27. `@RequestBody`: Indicates that a method parameter should be bound to the body of the web request.

28. `@ResponseBody`: Indicates that the return value of a method should be used as the response body.

29. `@CrossOrigin`: Enables cross-origin resource sharing (CORS) on a specific handler class or method.

#### Spring Boot Testing Annotations:

30. `@SpringBootTest`: Indicates that the class is a Spring Boot test class, which loads the complete application context.

31. `@MockBean`: Used to add mock objects to the Spring application context.

32. `@WebMvcTest`: Used to test Spring MVC controllers, loading only the web layer.

33. `@DataJpaTest`: Used to test JPA repositories, loading only the JPA components.

#### Transaction Management Annotations:

34. `@Transactional`: Indicates that a method or class should have transactional behavior.

5. What is `ConcurrentHashMap` in java? What is the internal working of `ConcurrentHashMap`?

Answer:

internal working of `concurrentHashMap` in java?

Answer:

an explanation of the internal workings of `ConcurrentHashMap` in Java :

`ConcurrentHashMap` in Java is a thread-safe implementation of the `Map` interface, designed for concurrent access from multiple threads without the need for external synchronization. It provides high-performance concurrent operations for put, get, and other map operations while maintaining consistency and avoiding contention between threads.

#### Internal Working of ConcurrentHashMap:

##### 1. Segmentation:

- `ConcurrentHashMap` internally uses a concept called segmentation or partitioning to achieve concurrency.

- It divides the underlying data structure into several segments (also known as hash buckets or partitions), each managed by its own lock.

- The number of segments is determined by the concurrency level specified during `ConcurrentHashMap` construction (default is 16).

##### 2. Segment Array:

- Internally, `ConcurrentHashMap` maintains an array of segments (`Segment<K, V>[]`) to hold the key-value pairs.

- Each segment is a separate hash table (similar to a regular `HashMap`) and implements its own locking mechanism.

##### 3. Hashing and Segment Selection:

- When a key-value pair is added or accessed, the key's hash code is used to determine the segment in which the entry should be placed or accessed.

- The hash code is mapped to a segment using bitwise operations (`hash & (segments.length - 1)`).

##### 4. Concurrent Operations:

- Concurrent read operations (e.g., `get`) can occur simultaneously on different segments without blocking each other.

- Write operations (e.g., `put`, `remove`) that modify the map require acquiring a lock on the specific segment in which the operation is performed.

- Locking at the segment level allows multiple threads to perform read operations concurrently while ensuring thread safety for write operations within a segment.

## 5. Segment-Level Locking:

- Each segment in `ConcurrentHashMap` is guarded by a separate lock (reentrant lock or striped lock) to allow concurrent writes within different segments.
- Lock contention is reduced because threads contending for entries in different segments don't block each other.

## 6. Updates and Rehashing:

- When updating the map (e.g., adding or removing entries), only the affected segment's lock is acquired, minimizing contention and allowing concurrent updates in different segments.
- Periodically, as the map grows or shrinks, `ConcurrentHashMap` performs rehashing operations on individual segments, not the entire map, reducing the scope of locking during rehashing.

## Advantages of ConcurrentHashMap:

1. Concurrency: Supports high concurrent read and write operations without blocking.
2. Performance: Provides efficient concurrent operations by partitioning the data structure and using segment-level locking.
3. Scalability: Scales well with increased concurrency levels and the number of threads accessing the map.
4. Thread Safety: Ensures thread safety for updates while allowing concurrent reads across different segments.
5. Reduced Contention: Minimizes lock contention by isolating locks to specific segments, reducing the likelihood of thread contention.

In summary, `ConcurrentHashMap` achieves thread safety and high concurrency by partitioning its internal data structure into segments, using segment-level locking, and allowing concurrent read and write operations across different segments. This design balances thread safety with performance, making it suitable for concurrent applications where multiple threads access and modify a shared map concurrently.

## Example:

```
import java.util.concurrent.ConcurrentHashMap;
```

```
public class ConcurrentHashMapExample {  
    public static void main(String[] args) {  
        // Create a ConcurrentHashMap with an initial capacity of 16 and a concurrency level of 4  
        ConcurrentHashMap<Integer, String> concurrentMap = new ConcurrentHashMap<>(16, 0.75f, 4);  
  
        // Adding elements to the ConcurrentHashMap  
        concurrentMap.put(1, "One");  
        concurrentMap.put(2, "Two");  
        concurrentMap.put(3, "Three");  
  
        // Displaying the ConcurrentHashMap  
        System.out.println("ConcurrentHashMap before modification: " + concurrentMap);  
  
        // Updating an existing value  
        concurrentMap.put(2, "UpdatedTwo");  
  
        // Adding a new element  
        concurrentMap.put(4, "Four");  
  
        // Removing an element  
        concurrentMap.remove(3);  
  
        // Displaying the modified ConcurrentHashMap  
        System.out.println("ConcurrentHashMap after modification: " + concurrentMap);  
  
        // Accessing a value  
        String value = concurrentMap.get(1);  
        System.out.println("Value associated with key 1: " + value);  
    }  
}
```



6. what is the Internal working of Hashmap in java? or how hashmap internally works?

Answer:

Certainly! Here's a comprehensive answer to the common interview question, "What is the internal working of a HashMap?"

"A HashMap is a widely used data structure in Java that provides efficient storage and retrieval of key-value pairs. Its internal working involves the use of an array of linked lists, which enables quick access to values based on their keys.

1. Hashing Mechanism: When you put a key-value pair into a HashMap, the key's `hashCode()` method is called to compute a hash code, which is an integer representation of the key. This hash code is used to determine the index at which the value will be stored in the underlying array.

2. Array of Buckets: The underlying data structure of a HashMap is an array of buckets. Each bucket is essentially a linked list or, in modern implementations, a tree (in the case of Java 8+ with high collision keys).

3. Index Calculation: The hash code is transformed into a valid index by using a modulo operation with the array's size. This index is where the value associated with the key will be stored.

4. Collision Handling: Due to the possibility of two different keys producing the same hash code (collision), HashMap uses linked lists (or trees) in each bucket to store multiple key-value pairs. If two keys have the same hash code, they will be placed in the same bucket.

5. Get and Put Operations: When you want to retrieve a value based on a key, the key's hash code is computed, and the appropriate bucket is located. The linked list (or tree) in that bucket is then traversed to find the key-value pair. For put operations, you follow the same process to store the key-value pair in the appropriate bucket.

6. Load Factor: A load factor is a measure of how full the HashMap is. When the load factor reaches a certain threshold (usually 0.75 in Java), the HashMap is resized, and the number of buckets is increased. This helps maintain efficient access times by reducing the likelihood of collisions.

7. Key Equality: To find the correct key-value pair within a bucket, HashMap uses the `equals()` method of keys to determine equality. If two keys have the same hash code but are not equal, they are considered distinct.

In summary, a HashMap uses the principles of hashing and indexing to provide fast access to values based on their keys. It handles collisions by storing multiple key-value pairs in the same bucket, and it dynamically resizes itself to maintain a balance between performance and memory usage. Understanding the internal workings of a HashMap is crucial for efficient data retrieval and manipulation in Java applications."

---

7. What if two keys have same has code means there is hash collision so how user will get the value? And which value he will get?

Answer:

In the case of a hash collision, where two keys produce the same hash code, the HashMap employs a mechanism to handle this situation and allow users to retrieve the correct value associated with their key. Here's how it works:

1. Bucket with Collisions: When two or more keys have the same hash code, they are placed in the same bucket. Each bucket in the HashMap contains a linked list (or a tree in more recent Java versions) to store key-value pairs.

2. Key Equality: To distinguish between keys with the same hash code, HashMap uses the `equals()` method of keys to determine equality. When you request a value based on a key, the HashMap searches the linked list (or tree) in the corresponding bucket to find the key that is equal to the key you provided.

3. Value Retrieval: The value associated with the key that matches the provided key (based on the `equals()` method) is retrieved and returned to the user.

In summary, if two keys produce the same hash code (hash collision), the HashMap stores them in the same bucket and uses the `equals()` method to determine the correct key-value pair. The user will get the value associated with the key that is equal to the key they used for the retrieval, ensuring they receive the appropriate value even in the presence of hash collisions.

---

8. Why strings are immutable in java?

Answer:

1. Security:

Immutable strings are crucial for security, especially when dealing with sensitive data such as usernames, passwords, and file paths. Since strings are immutable, they cannot be altered after creation, preventing accidental or malicious changes.

## 2. String Pooling:

Java uses a special memory area called the string pool to store string literals. When a new string is created, the JVM checks the string pool first. If the string already exists in the pool, a reference to the existing string is returned instead of creating a new one. This saves memory and improves performance. The immutability of strings ensures that the string pool remains consistent and reliable.

## 3. Thread Safety:

Since strings are immutable, they are inherently thread-safe. Multiple threads can access the same string instance without the need for synchronization, which simplifies concurrent programming and avoids potential synchronization issues.

---

## 9. How to create Immutable class in Java?

Answer:

Immutable class in java means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

- Declare the class as final so it can't be extended.
- Declare all fields as private and final to prevent modification outside the class and ensure they are initialized only once.
- Don't provide any setter methods to prevent the fields from being modified after the object is created.
- Initialize all fields via a constructor that takes all necessary parameters.
- Ensure deep copies are made of mutable objects passed to the constructor 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)

- Example:

```
import java.util.HashMap;  
import java.util.Map;
```

```
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}

```

---

10. What is the significance of equals and hashCode method?

Answer:

The `equals()` and `hashCode()` methods are fundamental methods in Java that play a crucial role in object comparison and hashing. Here's a detailed explanation of their significance:

### **`equals()` Method:**

- **Purpose:** Determines whether two objects are considered equal in value.
- **Functionality:** You override the default `equals()` method in your class to define your own criteria for object equality. This method typically compares the relevant fields of the objects to determine if they represent the same data.
- By default, the `equals` method in the `Object` class compares memory addresses, meaning two references are considered equal if they point to the same object in memory.

### **`hashCode()` Method:**

A hash code is an integer value that is associated with each object in Java which is a numeric representation of Java object. Its main purpose is to facilitate hashing in hash tables, which are used by data structures like `HashMap`.

- **Purpose:** Generates a unique integer hash code for an object. `HashSet`.
- **Importance:**
  - **Hashing:** The hash code is used as an index in hash tables. A good hash code implementation helps distribute objects evenly across buckets in the hash table, minimizing collisions (situations where multiple objects map to the same bucket). This leads to faster retrieval and insertion operations.
  - **Contract with `equals()`:** There's a crucial contract between `equals()` and `hashCode()`: If two objects are considered equal by `equals()`, they must generate the same hash code. This ensures consistency in how objects are stored and retrieved in hash-based collections.

### **Example:**

Java

```
public class Person {  
  
    private String name;  
    private int age;  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;
```

```

Person person = (Person) o;
return name.equals(person.name) && age == person.age;
}

```

**@Override**

```

public int hashCode() {
    return Objects.hash(name, age);
}
}

```

11. Contract between equals() and hashCode() method?

Answer:

- If 2 objects are equal by .equals() method compulsory their hashcodes must be equal (or) same. That is If r1.equals(r2) is true then r1.hashCode()==r2.hashCode() must be true.
- If 2 objects are not equal by .equals() method then there are no restrictions on hashCode() methods. They may be same (or) may be different. That is If r1.equals(r2) is false then r1.hashCode()==r2.hashCode() may be same (or) may be different.
- If hashcodes of 2 objects are equal we can't conclude anything about .equals() method it may returns true (or) false. That is If r1.hashCode()==r2.hashCode() is true then r1.equals(r2) method may returns true (or) false.
- If hashcodes of 2 objects are not equal then these objects are always not equal by .equals() method also. That is If r1.hashCode()==r2.hashCode() is false then r1.equals(r2) is always false.

To maintain the above contract between .equals() and hashCode() methods whenever we are overriding .equals() method compulsory we should override hashCode() method. Violation leads to no compile time error and runtime error but it is not good programming practice

<https://www.javatpoint.com/equals-and-hashcode-in-java>

---

12. what is hashing and what is the time complexity of hashing?

Answer:

Hashing is a process of applying a hash function to some data. A hash function is just a mathematical function. Hashing is widely used in various applications such as hash tables, hash maps, hash sets and more. The primary goal of hashing is to quickly locate a data record in a data structure based on its hash code, reducing the search time significantly compared to linear search.

**Here's how hashing works:**

**Hash Function:** A hash function takes an input (or key) and generates a fixed-size hash code as output. The hash code is typically a numeric value but can also be a string or byte array.

**Hash Table:** In hash-based data structures like hash tables or hash maps, the hash code is used as an index to store and retrieve data records efficiently. The hash code determines the location (bucket) where the data record should be stored or searched for.

**Collision Handling:** Since different keys can sometimes produce the same hash code (known as collisions), hash-based data structures employ collision resolution techniques such as separate chaining (using linked lists or arrays within each bucket) or open addressing (probing for an alternative empty location).

**Time Complexity of Hashing:** time complexity for search, insertion, and deletion operations in a well-designed hash table is  $O(1)$ . This is because the hash function helps locate the relevant bucket in the hash table quickly.

---

13. Difference between String vs StringBuilder vs StringBuffer in Java?

Answer:

Feature	String	StringBuilder	StringBuffer
Introduction	Introduced in JDK 1.0	Introduced in JDK 1.5	Introduced in JDK 1.0
Mutability	Immutable	Mutable	Mutable
Thread Safety	Thread Safe	Not Thread Safe	Thread Safe
Memory Efficiency	High	Efficient	Less Efficient
Performance	High(No-Synchronization)	High(No-Synchronization)	Low(Due to Synchronization)
Usage	This is used when we want immutability.	This is used when Thread safety is not required.	This is used when Thread safety is required.

---

14. How two can microservices communicate with each other ?

Answer:

#### **HTTP/HTTPS Communication:**

RESTful APIs: Microservices can communicate over HTTP using Representational State Transfer (REST) APIs. Each microservice exposes a set of endpoints (URLs) that other microservices or clients can access to perform CRUD (Create, Read, Update, Delete) operations on resources.

#### **Messaging Protocols:**



Message Queues: Microservices can use message queues such as RabbitMQ, Apache Kafka, or Amazon SQS for asynchronous communication. They send messages to queues, and other microservices or consumers can retrieve and process these messages later.

---

15. What is the role of API gateways in microservices?

Answer:

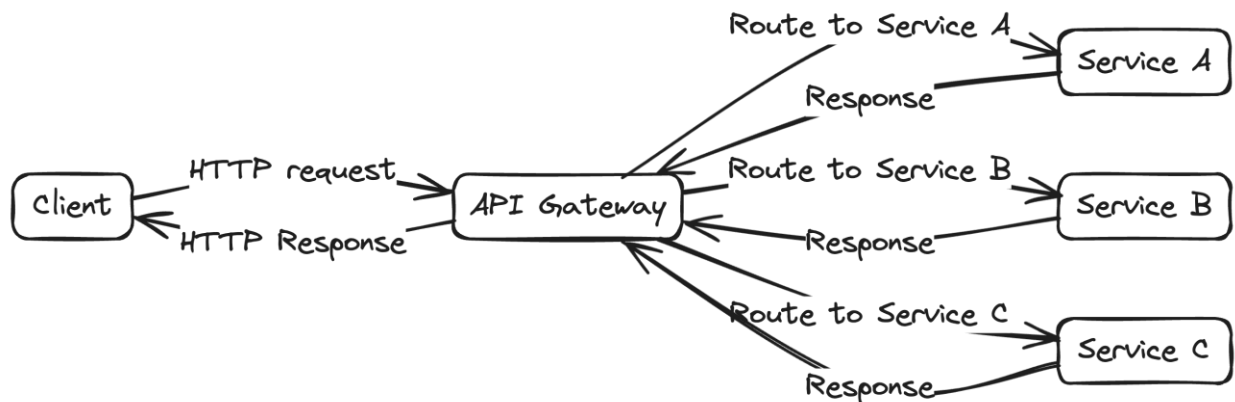
API gateways act as the entry point for client requests in a microservices architecture. They provide a unified interface to the outside world, abstracting the internal microservices landscape. API gateways handle tasks such as:

Request routing: Routing client requests to the appropriate microservices based on the request path, headers, or other criteria.

Protocol translation: Converting between different protocols used by clients and microservices, such as HTTP to gRPC or WebSocket.

Authentication and authorization: Enforcing security policies and validating client credentials before forwarding requests to microservices.

Rate limiting and throttling: Controlling the rate of incoming requests to prevent overloading the system.



---

16. How do you handle versioning in microservices?

Answer:

Versioning is important in microservices to allow for the independent evolution of services and to maintain compatibility with existing clients.

A common way of doing this is via URL versioning: This is where you include the version number in the API URL, such as `/api/v1/users` or `/api/v2/products`.

When we're running a migration from v1 to v2

---

17. Difference between linkedlist and Arraylist

Answer:

ArrayList	LinkedList	
1.	This class uses a dynamic array to store the elements in it. With the introduction of <a href="#">generics</a> , this class supports the storage of all types of objects.	This class uses a <a href="#">doubly linked list</a> to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects.
2.	Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted.	Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed.
3.	Inefficient memory utilization.	Good memory utilization.
4.	It can be one, two or multi-dimensional.	It can either be single, double or circular LinkedList.
5.	Insertion operation is slow.	Insertion operation is fast.
6.	This class implements a <a href="#">List interface</a> . Therefore, this acts as a list.	This class implements both the List interface and the <a href="#">Deque interface</a> . Therefore, it can act as a list and a deque.
7.	This class works better when the application demands storing the data and accessing it.	This class works better when the application demands manipulation of the stored data.

ArrayList	LinkedList	
8.	Data access and storage is very efficient as it stores the elements according to the indexes.	Data access and storage is slow in LinkedList.
9.	Deletion operation is not very efficient.	Deletion operation is very efficient.
10.	It is used to store only similar types of data.	It is used to store any types of data.
11.	Less memory is used.	More memory is used.
12.	This is known as static memory allocation.	This is known as dynamic memory allocation.

---

18. What are the rules in method overriding in java?

<https://metoro.io/blog/microservice-interview-questions>

---

19. What is auto wiring in spring or spring boot? And what are its type

Answer:

Autowiring in Spring and Spring Boot is a feature that allows Spring to automatically resolve and inject dependencies into beans. It reduces the need for explicit configuration by automatically wiring beans together based on certain criteria.

#### **Types of Autowiring in Spring:**

1. No Autowiring (no)

- This is the default setting. No autowiring is performed, and you need to explicitly wire dependencies using `<property>` or `<constructor-arg>` elements in XML configuration or using `@Autowired` annotations in Java configuration.

## 2. By Type (`byType`)

- Spring will look for a bean in the context whose type matches the property to be autowired. If exactly one bean of the required type is found, it will be injected. If no such bean is found, or if multiple beans of the same type are found, an exception is thrown.

## 3. By Name (`byName`)

- Spring will search for a bean in the context with the same name as the property to be autowired. If a matching bean is found, it will be injected. If no bean with the matching name is found, no injection occurs.

## 4. Constructor (`constructor`)

- Similar to `byType` but applies to constructor arguments. Spring will look for beans matching the constructor parameter types and will use those to autowire the constructor.

## 5. Autodetect (`autodetect`)

- This is a hybrid mode where Spring first tries to autowire by constructor. If that fails, it falls back to `byType`.

## Annotations for Autowiring in Spring:

### - `@Autowired`

- This is the most common annotation used for autowiring. It can be applied to fields, setter methods, and constructors.

- Example:

```
```java
@Autowired
private MyService myService;
```
```

### - `@Qualifier`

- Used in conjunction with `@Autowired` to resolve ambiguity when multiple beans of the same type exist. It specifies the name of the bean to be injected.

- Example:

```
```java
@Autowired
@Qualifier("specificBeanName")
private MyService myService;
```
```

### - `@Primary`

- This annotation is used on a bean definition to indicate that this bean should be given preference when multiple beans of the same type exist.

- Example:

```
```java
@Primary
```

```
@Bean
public MyService primaryMyService() {
    return new MyServiceImpl();
}
...

```

- `@Resource`

- This is a JSR-250 annotation that works similarly to `@Autowired`, but it allows injection by name.

- Example:

```
```java
@Resource(name = "myService")
private MyService myService;
...

```

These autowiring mechanisms make it easier to manage dependencies and keep the configuration cleaner and more manageable in Spring and Spring Boot applications.

---

## 20. What is dependency injection in spring boot and what are its type?

Answer:

Dependency Injection (DI) in Spring Boot is a design pattern used to implement Inversion of Control (IoC) where the control of object creation and dependency management is transferred from the application code to the Spring container. DI allows for better modularity, easier testing, and improved manageability of application components.

### Types of Dependency Injection:

#### 1. Constructor Injection:

- Dependencies are provided through a class constructor.
- Ensures that the injected dependency is available as soon as the object is created.
- Helps in making the dependencies immutable.
- Example:

```
```java
@Service
public class MyService {
    private final MyRepository myRepository;

    @Autowired
    public MyService(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
...

```

#### 2. Setter Injection:

- Dependencies are provided through setter methods after the object is created.
- Allows for optional dependencies.
- Example:

```

```java
@Service
public class MyService {
    private MyRepository myRepository;

    @Autowired
    public void setMyRepository(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

```

### 3. Field Injection:

- Dependencies are provided directly into the class fields.
- Simple and quick to implement but makes unit testing harder since dependencies are private and cannot be mocked easily.

#### - Example:

```

```java
@Service
public class MyService {
    @Autowired
    private MyRepository myRepository;
}
```

```

## How Dependency Injection Works in Spring Boot:

### 1. Bean Creation:

- Spring scans the classpath for components annotated with `@Component`, `@Service`, `@Repository`, `@Controller`, etc.
- These classes are instantiated and managed by the Spring container.

### 2. Dependency Resolution:

- When a bean is created, Spring checks its dependencies and tries to resolve them by looking for matching beans in the application context.
- Dependencies can be autowired using the `@Autowired` annotation, or they can be explicitly defined in a configuration class using `@Bean` methods.

### 3. Injection:

- Once the dependencies are resolved, they are injected into the bean, either through constructors, setter methods, or directly into fields.

## Example of Dependency Injection in Spring Boot:

```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
```

```

```

@Service
public class UserService {

    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User findById(Long id) {
        return userRepository.findById(id).orElse(null);
    }
}

```

In this example, `UserService` depends on `UserRepository`. Spring Boot will automatically inject an instance of `UserRepository` into `UserService` when it is created, thanks to the `@Autowired` annotation on the constructor.

### Choosing the Type of Injection:

- Constructor Injection: Preferred for mandatory dependencies. It makes the dependencies immutable and ensures they are provided during object creation.
- Setter Injection: Suitable for optional dependencies or when the dependency might change during the lifecycle of the bean.
- Field Injection: Quick and easy to implement but should be avoided for better testability and clarity of dependencies.

By using DI, Spring Boot applications achieve a clean separation of concerns, where the application logic does not need to be aware of the creation and lifecycle management of dependencies, leading to more maintainable and testable code.

## 21. How auto wiring different from dependency injection or is it same?

Ans:

Autowiring and Dependency Injection (DI) are closely related concepts in Spring, but they are not exactly the same. Here's how they differ and relate to each other:

### Dependency Injection (DI):

- Definition: DI is a design pattern that allows an object's dependencies to be injected at runtime rather than being hard-coded within the object.
- Purpose: The primary purpose of DI is to decouple components and improve the manageability and testability of the code.
- Implementation: DI can be implemented in several ways, such as constructor injection, setter injection, or field injection.
- Scope: DI is a broader concept that applies to any framework or programming environment.

Autowiring:

- Definition: Autowiring is a specific feature provided by the Spring framework to automatically resolve and inject dependencies into Spring beans.
- Purpose: The purpose of autowiring is to reduce the need for explicit configuration and make dependency management more convenient.
- Implementation: Autowiring is typically done using annotations like `@Autowired`, `@Qualifier`, and configuration options like `byType`, `byName`, etc.
- Scope: Autowiring is a feature specific to the Spring framework, which simplifies the implementation of DI.

### **Relationship:**

- Autowiring Uses DI: Autowiring is essentially a way to implement DI in Spring applications. When you use autowiring, you are instructing Spring to automatically inject dependencies into your beans.
- Convenience: Autowiring makes the process of DI easier by automating the wiring of dependencies based on certain criteria (type, name, etc.).

Example:

Consider a simple example to illustrate both concepts.

### **# Dependency Injection (without autowiring):**

```
```java
import org.springframework.stereotype.Service;

@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```
```

Here, DI is being used by passing the `UserRepository` dependency through the constructor. However, this requires manual wiring in the configuration file or through a configuration class.

### **# Dependency Injection with Autowiring:**

```
```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    private final UserRepository userRepository;

    @Autowired
```



```

public UserService(UserRepository userRepository) {
    this.userRepository = userRepository;
}
}
```

```

In this example, `@Autowired` is used to autowire the `UserRepository` dependency. Spring will automatically resolve and inject the appropriate `UserRepository` bean.

---

22. What is the difference between association, composition and aggregation in java

Ans:

In Java (and in object-oriented programming in general), association, composition, and aggregation are types of relationships that can exist between objects. These relationships define how objects interact with each other and the nature of their connections. Here's an explanation of each:

### 1. Association:

- Definition: Association represents a relationship between two or more objects where they can communicate with each other. It is a loose relationship without any ownership.
- Characteristics: Objects remain independent, and the association can be either one-to-one, one-to-many, many-to-one, or many-to-many.
- Example: A `Teacher` and `Student` can be associated with each other in a school. A teacher can teach multiple students, and a student can have multiple teachers.

```

```java
public class Teacher {
    private String name;

    // Constructor and methods...
}

public class Student {
    private String name;

    // Constructor and methods...
}

public class School {
    public static void main(String[] args) {
        Teacher teacher = new Teacher("Mrs. Smith");
        Student student = new Student("John Doe");

        // Association between teacher and student
    }
}
```

```

## 2. Aggregation:

- Definition: Aggregation is a specialized form of association where one object (the whole) contains or is composed of other objects (the parts). The parts can exist independently of the whole.
- Characteristics: It represents a "has-a" relationship with shared ownership. The lifecycle of the parts does not depend on the whole. If the whole is destroyed, the parts can still exist.
- Example: A `Library` contains `Books`. The books can exist independently of the library.

```
```java
public class Book {
    private String title;

    // Constructor and methods...
}

public class Library {
    private List<Book> books;

    public Library() {
        this.books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    // Other methods...
}

public class Main {
    public static void main(String[] args) {
        Library library = new Library();
        Book book1 = new Book("1984");
        Book book2 = new Book("Brave New World");

        library.addBook(book1);
        library.addBook(book2);

        // Books can exist independently of the library
    }
}
```
```

## 3. Composition:

- Definition: Composition is a stronger form of aggregation where the parts cannot exist independently of the whole. If the whole is destroyed, the parts are also destroyed.
- Characteristics: It represents a "contains-a" relationship with strong ownership. The lifecycle of the parts depends on the whole.
- Example: A `House` contains `Rooms`. If the house is destroyed, the rooms do not exist independently.

```

```java
public class Room {
    private String name;

    // Constructor and methods...
}

public class House {
    private List<Room> rooms;

    public House() {
        this.rooms = new ArrayList<>();
        rooms.add(new Room("Living Room"));
        rooms.add(new Room("Bedroom"));
    }

    // Other methods...
}

public class Main {
    public static void main(String[] args) {
        House house = new House();

        // Rooms do not exist independently of the house
    }
}
```

```

### Summary:

- Association: A general relationship where objects can communicate, but neither owns the other.
- Aggregation: A "has-a" relationship with shared ownership. The parts can exist independently of the whole.
- Composition: A "contains-a" relationship with strong ownership. The parts cannot exist independently of the whole.

---

23. Difference between put vs patch HttpMethod?

Answer:

### PUT:

- **Purpose:** Used to completely replace an existing resource at a specific URL with the data sent in the request body.
- **Idempotent:** Sending a PUT request multiple times with the same data should produce the same result (i.e., replacing the resource with the same data).
- **Payload:** Requires a complete representation of the resource data in the request body.

- **Use Cases:**
  - Updating an entire resource when you have all the data and want to overwrite the existing representation.
  - Creating a new resource if it doesn't exist at the specified URL. (Some servers might treat PUT for a non-existent resource as a creation operation.)

## **PATCH:**

- **Purpose:** Used to partially update an existing resource at a specific URL. Only the provided data in the request body is modified, leaving the remaining resource properties unchanged.
- **Non-Idempotent:** Sending a PATCH request multiple times with the same data might lead to different results, as it depends on the current state of the resource.
- **Payload:** Contains a set of instructions specifying the changes to be applied to the resource. The format of the payload can vary depending on the server implementation (e.g., JSON Patch, XML Patch).
- **Use Cases:**
  - Updating specific fields or attributes of a resource without modifying the entire representation.
  - Applying specific changes to a resource based on business logic.

## **Here's an analogy to understand the difference:**

Imagine you have a book (the resource).

- **PUT:** Replacing the entire book with a new edition (complete overwrite with new data).
- **PATCH:** Updating a specific chapter or section of the book while leaving the rest unchanged (partial modification).

**In essence, choose PUT for complete replacements and PATCH for partial updates, considering the idempotence, payload requirements, and complexity factors for your specific use case.**

---

24. What is index in SQL?

Answer:

In SQL, indexes are data structures that improve the performance of data retrieval operations on tables. They act like an organized filing system for your table data, allowing the database to quickly locate specific rows based on certain columns.

Here's how indexes work:

1. **Structure:** Indexes are typically implemented using B-trees (balanced search trees) or hash tables. These structures enable efficient searching and locating of data based on the indexed columns.
2. **Creation:** You can create indexes on specific columns in a table using SQL commands like `CREATE INDEX`.
3. **Benefits:** Indexes significantly speed up queries that involve:
  - **Filtering:** Queries that filter rows based on specific values in the indexed columns.
  - **Sorting:** Sorting operations on the indexed columns become faster as the database can leverage the pre-sorted order of the index.
  - **Joining:** Joining tables can be faster if the join condition involves indexed columns.
4. **Trade-offs:** While indexes enhance retrieval speed, they come with some trade-offs:
  - **Storage Overhead:** Indexes require additional storage space on the disk to maintain the index structure.
  - **Write Performance:** Inserting and updating data can become slightly slower due to the need to update the indexes along with the table data.

---

25. Which docker command you will use to look into the docker container?

Answer:

### 1. docker ps:

- This is the most basic command to get an overview of your running containers. It displays information like container ID, image name, status, ports, and names.
- Example output:
- | CONTAINER ID   | IMAGE        | COMMAND   | CREATED      | STATUS        | PORTS               | NAMES            |
|----------------|--------------|-----------|--------------|---------------|---------------------|------------------|
| <container_id> | <image_name> | <command> | <created_at> | Up <duration> | <port>-><host_port> | <container_name> |

### 3. docker logs <container\_id>:

- This command displays the logs generated by a container. Logs can be very helpful for debugging and monitoring the container's behavior.
- Example usage:
- `docker logs <container_id>`

#### 4. `docker exec -it <container_id> bash:`

- This command allows you to run a command (bash in this case) inside a running container. This is useful for directly interacting with the container's file system, running applications, or troubleshooting issues.
- **Important:** The `-i` flag allows interactive mode, and `-t` allocates a pseudo-tty (terminal).
- Example usage:
- `docker exec -it <container_id> bash`

26. When should we use LinkedList and when should we use ArrayList?

Answer:

In Java, `LinkedList` and `ArrayList` are both implementations of the `List` interface, but they have different performance characteristics due to their underlying data structures. Here's a detailed comparison and guidelines on when to use each:

#### **ArrayList**

Underlying Data Structure: Dynamic array.

Advantages:

1. Fast Random Access: Provides  $O(1)$  time complexity for accessing an element by index.
2. Efficient Iteration: Iterating through an `ArrayList` is faster because it maintains contiguous memory locations.
3. Low Memory Overhead: Generally has less memory overhead compared to `LinkedList`.

Disadvantages:

1. Costly Insertions/Deletions: Insertions and deletions (except at the end) are costly ( $O(n)$ ) because elements need to be shifted.
2. Resizing Cost: If the array needs to grow beyond its capacity, it must be resized, which involves creating a new array and copying elements.

Use Case:

- When you need fast access to elements via indexes.
- When you have a fixed or relatively stable number of elements.
- When the majority of operations involve iterating over the list rather than modifying it frequently.

#### **LinkedList**

Underlying Data Structure: Doubly linked list.

Advantages:

1. Efficient Insertions/Deletions: Insertions and deletions are efficient ( $O(1)$ ) as long as you have a reference to the node. No shifting of elements is needed.
2. Better at Handling Frequent Additions/Removals: Especially when modifications are frequent and near the beginning or end of the list.

Disadvantages:

1. Slow Random Access: Provides  $O(n)$  time complexity for accessing an element by index, as it requires traversal from the head or tail.
2. Higher Memory Overhead: Each element stores additional pointers to the previous and next elements.
3. Slower Iteration: Iterating through a `LinkedList` can be slower compared to an `ArrayList` due to non-contiguous memory locations.

Use Case:

- When you need efficient insertions and deletions at the beginning, end, or middle of the list.
- When the number of elements frequently changes.
- When you are more concerned about performance of add/remove operations than random access.

---

## 27. how do you estimate for efforts for your user story?

Answer:

Estimating efforts for a user story is a crucial part of agile development and sprint planning. Based on your experience and context, here's a detailed answer on how you can explain the estimation process for your user stories:

### Estimating Efforts for a User Story

#### 1. Understanding the User Story:

First, ensure that you fully understand the user story. This involves reviewing the story's description, acceptance criteria, and any attached wireframes or documents. If anything is unclear, discuss it with the product owner or business analyst to get a complete understanding.

Example:

Suppose the user story is to implement a new feature that allows users to filter search results based on certain criteria.

#### 2. Breaking Down the User Story:

Next, break down the user story into smaller tasks or subtasks. This helps in understanding the different components involved and makes the estimation process more manageable.

Example:

For the filtering feature:

- Design the filter UI component.
- Implement the backend logic to handle filter requests.
- Integrate the UI with the backend.
- Write unit and integration tests.
- Perform code review and testing.

#### 3. Using Estimation Techniques:

Utilize estimation techniques such as Planning Poker, T-Shirt Sizing, or the Fibonacci Sequence to estimate each subtask.

Example:

We typically use Planning Poker with story points, where each developer independently estimates the effort required for each task using a deck of cards with Fibonacci numbers (1, 2, 3, 5, 8, 13, etc.). We then discuss our estimates to reach a consensus.

#### 4. Considering Complexity, Uncertainty, and Effort:

Take into account the complexity, uncertainty, and amount of effort required for each task. Complexity involves the technical difficulty, uncertainty includes unknowns or risks, and effort is the actual work required.

Example:

- Designing the UI might be a 3 (relatively straightforward).
- Implementing the backend logic might be an 8 (more complex with potential integration issues).
- Integration could be a 5 (moderate complexity).
- Writing tests might be a 3 (standard effort).
- Code review and testing could be a 2 (routine).

#### 5. Adding Buffer Time:

Incorporate buffer time for unforeseen challenges or additional work that might arise. This is particularly important for tasks with higher uncertainty.

Example:

If a task is estimated to be 8 points, you might add a 1-2 point buffer for potential issues.

#### 6. Summarizing the Total Effort:

Sum up the story points for all subtasks to get the total effort required for the user story.

Example:

- UI Design: 3 points
- Backend Logic: 8 points
- Integration: 5 points
- Tests: 3 points
- Code Review and Testing: 2 points
- Buffer: 1-2 points

Total: 22 points

#### 7. Discussing and Finalizing in Sprint Planning:

During the sprint planning meeting, present your estimates and discuss them with the team. Adjust the estimates if necessary based on team feedback and finalize them.

Example:

You might discuss that the backend logic, initially estimated at 8 points, could be reduced to 6 points if another developer has prior experience with similar tasks.

Explanation:



"When estimating efforts for a user story, I start by thoroughly understanding the story's requirements and acceptance criteria. I then break the story down into smaller, manageable tasks. Using techniques like Planning Poker with Fibonacci sequences, I estimate each task by considering its complexity, uncertainty, and effort required. For instance, designing a new filter UI might be a straightforward task estimated at 3 points, while implementing the backend logic might be more complex, estimated at 8 points. I also account for buffer time to address any unforeseen challenges. Summing up the points for all tasks gives the total effort required. During sprint planning, I discuss these estimates with the team to finalize them, ensuring everyone is aligned and the estimates are realistic."

---

28. what if lets say even working on User story you find out the user story is getting spill or at risk what you will do ?

Answer:

If, during the course of working on a user story, you realize that it is at risk of spilling over into the next sprint or not being completed on time, it is essential to take proactive steps to manage the situation. Here's a detailed answer on what actions you would take:

#### Addressing a User Story at Risk

##### 1. Early Identification and Communication:

As soon as you identify that the user story is at risk, communicate this to the relevant stakeholders, such as the Scrum Master, Product Owner, and your team. Early communication is key to managing expectations and finding solutions.

Example:

"I noticed that due to unforeseen technical challenges with the backend integration, the user story might not be completed within the current sprint. I promptly informed the Scrum Master and Product Owner about this risk."

##### 2. Analyze the Root Cause:

Investigate and understand the reasons why the user story is at risk. This could be due to underestimated complexity, unexpected technical challenges, dependency on another team's work, or other factors.

Example:

"Upon further analysis, I found that the integration with the third-party API was more complex than initially estimated. This required additional research and troubleshooting time."

##### 3. Reassess and Adjust the Plan:

Reassess the remaining tasks and adjust the plan accordingly. Break down the remaining work into smaller, more manageable tasks, if possible, and re-estimate their effort.

Example:

"We broke down the remaining work into smaller tasks and reassessed the effort required. We also identified tasks that could be deprioritized or postponed."

##### 4. Seek Help and Collaborate:

Seek help from your team or other experts if needed. Collaboration can often lead to finding quicker or more efficient solutions to the problem at hand.

Example:

"I reached out to a colleague who had prior experience with similar integrations for assistance. Their insights helped us to overcome some of the technical hurdles more quickly."

#### 5. Prioritize and Focus on Critical Tasks:

Prioritize the most critical tasks that must be completed to deliver a minimum viable product (MVP) for the user story. This might mean deferring less critical features or functionalities to a later sprint.

Example:

"We prioritized the essential parts of the user story that would allow us to deliver a functional MVP, even if it meant deferring some of the less critical features."

#### 6. Update the Sprint Backlog and Adjust Scope:

Update the sprint backlog to reflect the changes in scope and communicate these changes to the team. If necessary, adjust the scope of the sprint by negotiating with the Product Owner on which tasks can be deferred or removed.

Example:

"We updated the sprint backlog to reflect the adjusted scope and communicated the changes to the team. The Product Owner agreed to defer a non-critical feature to the next sprint."

#### 7. Monitor and Report Progress:

Continue to monitor progress closely and provide regular updates to the team and stakeholders. This ensures everyone is aware of the current status and any additional risks that may arise.

Example:

"I provided daily updates during the stand-up meetings and ensured the Scrum Master and Product Owner were kept in the loop about our progress and any additional risks."

#### 8. Retrospective and Lessons Learned:

After the sprint, during the retrospective meeting, discuss what led to the user story being at risk and identify areas for improvement. This helps in refining estimation techniques and planning processes for future sprints.

Example:

"During the sprint retrospective, we analyzed why the story was at risk and identified that better estimation and early technical validation could prevent similar issues in the future. We decided to include technical spikes in our planning for complex integrations."

Explanation:

"If I find a user story at risk of not being completed within the sprint, I take immediate action by communicating the risk to the Scrum Master, Product Owner, and team. I analyze the root cause of the delay, reassess the remaining tasks, and adjust the plan accordingly. Seeking help from colleagues with relevant experience and prioritizing critical tasks are crucial steps. We update the sprint backlog and, if necessary, adjust the scope by negotiating with the Product Owner. Throughout this process, I ensure to provide regular

updates on progress and any additional risks. Finally, we review the situation during the sprint retrospective to learn from the experience and improve our future planning and estimation processes."

This approach demonstrates your proactive problem-solving skills, collaboration with the team, and commitment to continuous improvement, which are all valuable traits in an agile development environment.

---

29. What are the different agile ceremonies done in your project? and how do you follow them? also explain what happen in each and what is your role in it

Answer:

Agile ceremonies are key events that bring the team together to plan, execute, and review their work. Here's a breakdown of the different Agile ceremonies commonly practiced in an Agile project and how they are followed, along with my role in each:

### 1. Sprint Planning

Purpose: To define what can be delivered in the upcoming sprint and how that work will be achieved.

- Frequency: At the beginning of each sprint (typically every 2 weeks).
- Attendees: Scrum Master, Product Owner, Development Team.
- Activities:
  - Review of Product Backlog: The Product Owner presents the prioritized items from the product backlog.
  - Sprint Goal Setting: The team discusses and agrees on a sprint goal.
  - Task Breakdown: The team breaks down user stories into smaller tasks and estimates the effort required.
- My Role: As a team member, I participate in task estimation, provide input on what can realistically be accomplished, and commit to specific tasks.

### 2. Daily Stand-up (Daily Scrum)

Purpose: To synchronize the team's activities and plan for the next 24 hours.

- Frequency: Daily (usually in the morning).
- Attendees: Scrum Master, Development Team.
- Activities:
  - Each Team Member Answers Three Questions:
    1. What did I do yesterday?
    2. What will I do today?
    3. Are there any impediments in my way?
- My Role: I provide updates on my progress, plan my tasks for the day, and raise any issues or blockers.

### 3. Sprint Review

Purpose: To review the work completed during the sprint and demonstrate the product increment to stakeholders.

- Frequency: At the end of each sprint.
- Attendees: Scrum Master, Product Owner, Development Team, Stakeholders.
- Activities:
  - Demo: The team demonstrates the completed work to stakeholders.
  - Feedback Collection: Stakeholders provide feedback on the product increment.
- My Role: I may present the work I contributed to and help collect and understand feedback for future improvements.

#### 4. Sprint Retrospective

Purpose: To reflect on the past sprint and identify areas for improvement.

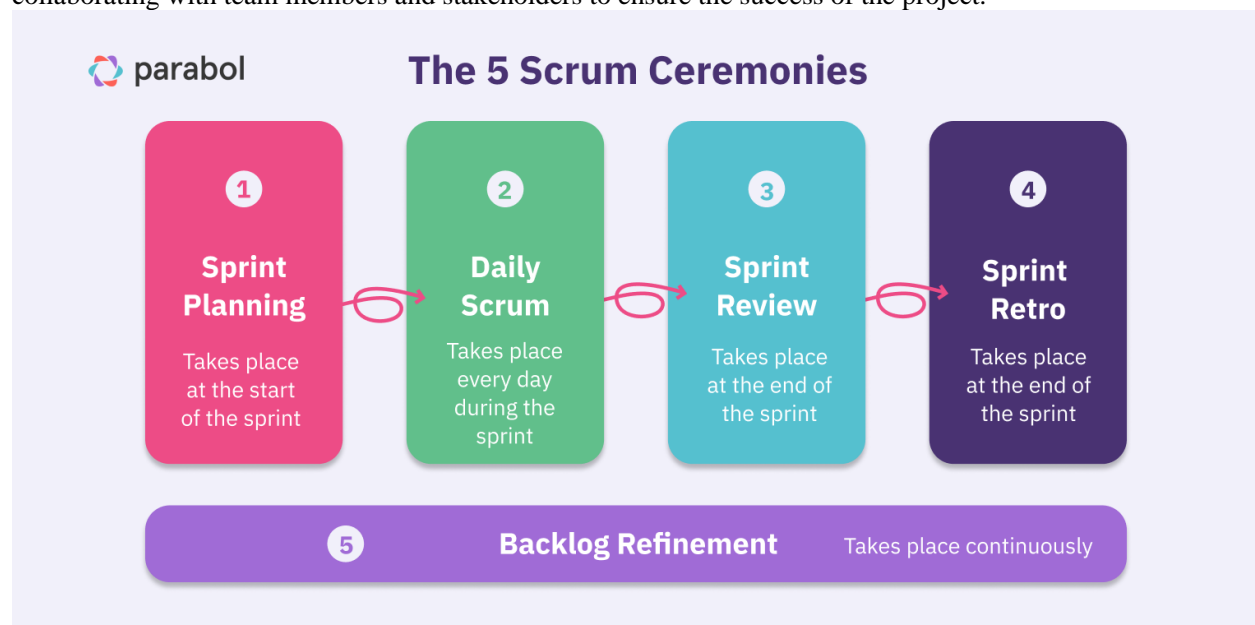
- Frequency: At the end of each sprint, after the Sprint Review.
- Attendees: Scrum Master, Development Team.
- Activities:
  - Review of Previous Actions: Discuss whether previous action items were completed.
  - Identify What Went Well and What Didn't: The team identifies positive aspects and areas needing improvement.
  - Action Items for Improvement: The team agrees on actionable steps to improve processes or teamwork in the next sprint.
- My Role: I actively participate in discussions, provide feedback, and contribute to identifying actionable improvements.

#### 5. Backlog Refinement (Grooming)

Purpose: To ensure the product backlog is up-to-date and has well-defined items for future sprints.

- Frequency: Regularly throughout the sprint (often once a week).
- Attendees: Product Owner, Development Team.
- Activities:
  - Review and Prioritize: The Product Owner and team review the backlog items, prioritize them, and clarify any ambiguities.
  - Estimation: The team estimates the effort required for the backlog items.
- My Role: I help in clarifying user stories, providing estimates, and ensuring that the backlog items are ready for the next sprint planning.

In summary, participating in these Agile ceremonies ensures that the team is aligned, obstacles are addressed, and continuous improvement is achieved. My role involves active participation, providing input, and collaborating with team members and stakeholders to ensure the success of the project.



<https://www.parabol.co/blog/scrum-ceremonies-for-remote-teams/>

---

30. explain the entire flow for requirement like how or who gets the requirement to you?  
what is your role in this ? how do you decide scope of requirement? gathering etc till it delivery

Answer:

Sure, I can explain the entire flow for requirements in an Agile project, including how requirements are received, the role of different team members, how scope is decided, gathering requirements, and the process until delivery:

1. Requirement Reception:

1. Product Owner (PO) Receives Requirements: The Product Owner is typically responsible for gathering and managing requirements. They may receive requirements from various sources such as stakeholders, customers, business analysts, or market research.

2. Role of Team Members:

1. Product Owner (PO):

- Receives and understands the requirements.
- Prioritizes requirements based on business value.
- Defines the scope of each requirement.

2. Development Team:

- Works closely with the Product Owner to understand requirements.
- Provides input on technical feasibility and effort estimation.
- Collaborates on defining the scope and acceptance criteria.

3. Scope and Requirement Definition:

1. Scope Definition:

- The Product Owner defines the scope of each requirement, including what is in and out of scope.
- Scope is determined based on business priorities, available resources, and project constraints.

2. Requirement Definition:

- The Product Owner and Development Team collaborate to clearly define each requirement.
- This includes writing user stories, acceptance criteria, and any necessary documentation.

4. Requirement Gathering:

1. Collaborative Sessions:

- The Product Owner conducts collaborative sessions such as backlog refinement meetings with the Development Team to gather detailed requirements.
- Stakeholder meetings may also be held to gather input and feedback on requirements.

2. Feedback Incorporation:

- Feedback from stakeholders, customers, and team members is incorporated into refining and finalizing requirements.

5. Requirement Prioritization:

1. Product Backlog:

- The finalized requirements are added to the product backlog by the Product Owner.

- Requirements are prioritized based on business value, urgency, dependencies, and risk.

## 6. Development and Delivery:

### 1. Sprint Planning:

- During sprint planning, the Development Team selects a set of requirements (user stories) from the product backlog to work on during the sprint.
- The scope for each sprint is defined based on the selected requirements.

### 2. Development and Testing:

- The Development Team works on implementing the requirements, following Agile practices such as daily stand-ups, continuous integration, and iterative development.
- Testing is conducted concurrently with development to ensure the quality of the deliverables.

### 3. Review and Feedback:

- Completed work is reviewed during the sprint review meeting, where stakeholders provide feedback.
- Feedback is used to make adjustments and improvements for subsequent sprints.

### 4. Delivery:

- At the end of each sprint, potentially shippable increments of the product are delivered.
- Continuous delivery or deployment practices may be used to deliver updates to users regularly.

## My Role in the Process:

### 1. Understanding Requirements:

- As a team member, I actively participate in understanding requirements during backlog refinement sessions and collaborative discussions.
- I seek clarity on requirements, ask questions, and provide input on technical aspects and feasibility.

### 2. Scope Definition:

- I collaborate with the Product Owner and team to define the scope of each requirement and ensure that it aligns with project goals and priorities.

### 3. Development and Testing:

- I contribute to the development and testing of the requirements assigned to me during the sprint.
- I follow Agile practices and collaborate with team members to deliver high-quality work.

### 4. Feedback and Continuous Improvement:

- I participate in sprint reviews to gather feedback from stakeholders and team members.
- I use feedback to improve processes, refine requirements, and enhance the quality of deliverables in subsequent sprints.

By following this structured approach, teams can effectively gather, define, prioritize, and deliver requirements in an Agile project, ensuring alignment with business objectives and customer needs.

---

31.

---

32.

---

33.

---

34.

---

35.

---

36.

---

37.

---

38.

---

39.

---

40.

---

41.

---

42.

---

43.

---

44.

---

45.

---

46.

---

47.

---

48.

---

49.

---

50.

---

51.

---

---

---

---

