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>

1. 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.**

1. 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

1. 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.

1. 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);

}

}

1. 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."

1. 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.

1. 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.

1. 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}

1. 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);

}

}

1. 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.hascode()==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>

1. 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.

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

Anwer:

| **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. |

1. 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.

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

Answer:

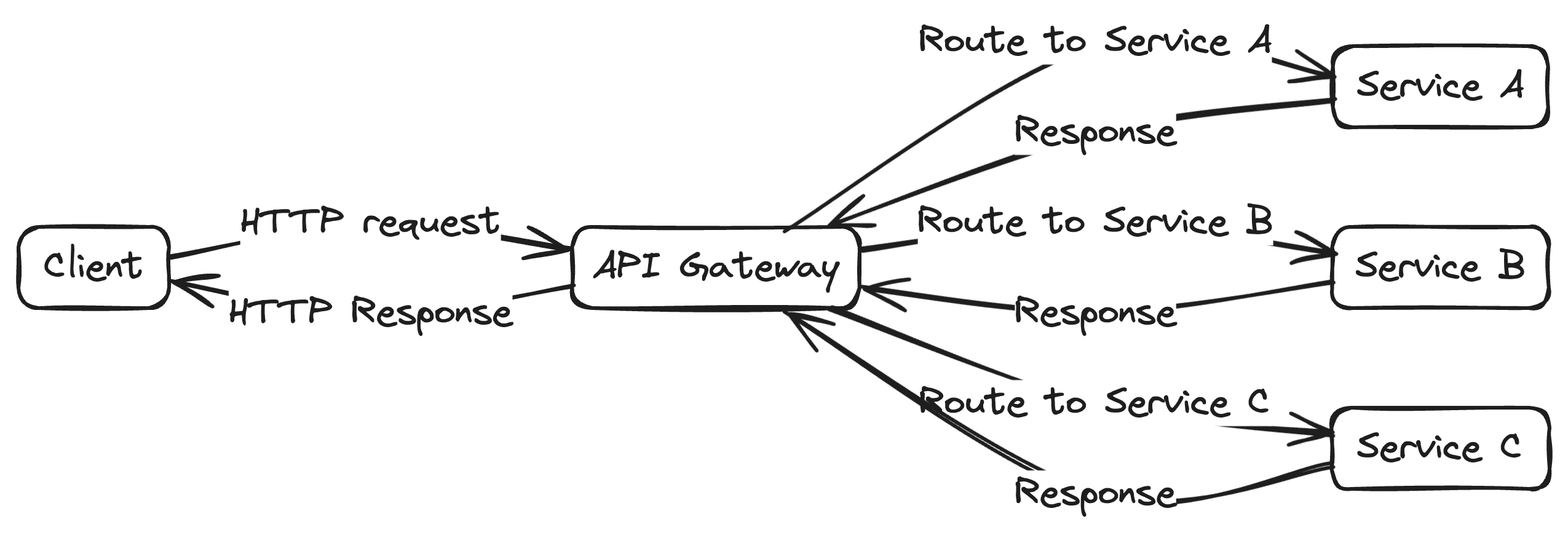
API gateways act as the entry point for client requests in a microservices architecture. They provide a unified interface to the outside world, abstracting the internal microservices landscape. API gateways handle tasks such as:

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.



1. 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

1. 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 [generics](https://www.geeksforgeeks.org/generics-in-java/), this class supports the storage of all types of objects. | This class uses a [doubly linked list](https://www.geeksforgeeks.org/doubly-linked-list/) 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 [List interface](https://www.geeksforgeeks.org/list-interface-java-examples/). Therefore, this acts as a list. | This class implements both the List interface and the [Deque interface](https://www.geeksforgeeks.org/deque-interface-java-example/). 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. |
| **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. |

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