# My Introduction:

Hi, My name is Huy. I have experience working in the IT industry for more than 3 years. I started my career as a full stack developer (Backend heavy) in NAB. First of all, I joined it when I was a final year student in RMIT university. I had worked in NAB for approximately three years and the technology stacks are Java and Typescript. My project is about Loan Modification that supports the customer to split their loan or fix and refix the interest rate. I had the opportunity to serve as a service owner and champions of two features. After that, I have switched to HCLTech to work with ANZ plus project from May 2024. Therefore, I have 4-5 months to develop the application with golang. For the project in ANZ plus, I joined the customer engagement technology team that provided the customer to start the conversation with the coach. For example, when the customer has any question and needs help from ANZ, they can create the conversation, so they can chat or call for help. Our application will integrate with Twilio and be embedded into Salesforce. In more than my three years experience, I have improved my critical thinking, collaboration skills, communication and I love working with a team. The reason why I want to join another company is that I want to be proficient and expert with Java language instead of Golang. Moreover, my current company does not provide any learning material for me to improve myself.

# Java core:

| **Core Java** | **Advance Java** |
| --- | --- |
| Core Java covers the basic concepts of the Java programming language. | Advanced Java covers the advanced topics and concepts of the Java programming language. |
| Core Java is used for developing computing or desktop applications. | Advanced Java is used for developing enterprise applications. |
| It is the first step, to begin with, Java. | It is the next step after completing the Core Java. |
| Core Java is based on single-tier architecture. | Advanced Java is based on two-tier architecture. |
| It comes under Java SE. | It comes under Java EE or J2EE. |
| It covers core topics such as OOPs, inheritance, exception handling, etc. | It covers advanced topics such as JDBC, servlets, JSP, web services etc. |

## 

## Overloading vs Overriding

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have an IS-A (inheritance) relationship. |
| In case of method overloading, *parameters must be different*. | In case of method overriding, *parameters must be the same*. |
| Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| In Java, method overloading can't be performed by changing the return type of the method only. *Return type can be the same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

## Static

In Java, the static keyword is used to indicate that a member (variable, method, block, or nested class) belongs to the class rather than to instances (objects) of that class. It essentially means that the member can be accessed without creating an instance of the class.

The static keyword can be used in 4 scenarios

* static variables
* static methods
* static blocks of code
* static nested class

**Static variable**

* It is a variable which belongs to the class and not to object (instance).
* Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.
* A single copy to be shared by all instances of the class.
* A static variable can be accessed directly by the class name and doesn’t need any object.
* Syntax: Class.variable

**Static method**

* It is a method which belongs to the class and not to the object (instance).
* A static method can access only static data. It can not access non-static data (instance variables) unless it has/creates an instance of the class.
* A static method can call only other static methods and can not call a non-static method from it unless it has/creates an instance of the class.
* A static method can be accessed directly by the class name and doesn’t need any object.
* Syntax: Class.methodName()
* A static method cannot refer to this or super keywords in any way.

**Static class**

* Java also has "static nested classes". A static nested class is just one which doesn't implicitly have a reference to an instance of the outer class.
* Static nested classes can have instance methods and static methods.
* There's no such thing as a top-level static class in Java.

Side note:

**The main method is static since it must be accessible for an application to run before any instantiation takes place.**

| **class StaticVariable {  public static String STATIC = "static"; }  class StaticMethod {  public static void print(String method) {  System.out.println(StaticVariable.STATIC + " " + method);  } }  class StaticBlock {  private static String subject;   static {  System.out.println("Khối static được gọi");  }   static {  subject = "Khối static (static blocks)";  }   StaticBlock () {  System.out.println("hàm main() được gọi");  System.out.println("Subject = " + subject);  } }  class StaticClass {  static class NestedStaticClass {  public static String STATIC\_CLASS = "static class";  }   public void printStaticClass() {  System.out.println(NestedStaticClass.STATIC\_CLASS);  } }  public class StaticAndFinal {   public static void main(String[] args) {  System.out.println(StaticVariable.STATIC + " variable");   StaticMethod.print("method print");   StaticBlock staticBlock = new StaticBlock();   StaticClass staticClass = new StaticClass();  staticClass.printStaticClass();    } }** |
| --- |

## Final

The final keyword is used in several different contexts to define an entity which cannot later be changed.

* A final class cannot be subclassed. This is done for reasons of security and efficiency. Accordingly, many of the Java standard library classes are final, for example java.lang.System and java.lang.String. All methods in a final class are implicitly final.
* A final method can't be overridden by subclasses. This is used to prevent unexpected behavior from a subclass altering a method that may be crucial to the function or consistency of the class.
* A final variable can only be initialized once, either via an initializer or an assignment statement.  
  It does not need to be initialized at the point of declaration, this is called a blank final variable, but in this case:
  + A blank final instance variable must be assigned at every constructor of its class.
  + A blank final static variable must be assigned in a static initializer in its class.

## Exception And Error

| **Aspect** | **Errors** | **Exceptions** |
| --- | --- | --- |
| **Nature** | Serious issues, typically outside the control of the application | Conditions that a program might want to catch and handle |
| **Examples** | OutOfMemoryError, StackOverflowError,  NoClassDefFoundError | NullPointerException, IOException, IllegalArgumentException |
| **Handling** | Generally not meant to be caught or handled by the application | Can be caught and handled using try-catch blocks |
| **Types** | Not categorized further | Checked Exceptions ( IOException), Unchecked Exceptions (NullPointerException) |
| **Occurrence** | Can occur at both compile time and runtime, mostly runtime issues | Occur only at runtime, though checked exceptions are detected at compile time |
| **Recoverability** | Generally not recoverable | Often can be handled and recovered from |
| **Impact** | Usually indicate severe problems that might cause the application to terminate | Typically less severe and can be managed within the application |

## Check and Uncheck Exception

| **Check** | **Uncheck** |
| --- | --- |
| Checked at compile-time | Checked at runtime |
| Must be handled or declared | Not required to be handled |
| Typically external/environmental issues | Typically programming errors |
| * IOException * SQLException * FileNotFoundException | * NullPointerException * ArrayIndexOutOfBoundsException * ArithmeticException |

## Equals and Hashcode

In Java, hashCode() and equals() are two fundamental methods provided by the Object class, which is the superclass of all Java classes. These methods play a crucial role in how objects are compared and stored in hash-based collections like HashMap, HashSet, and Hashtable.

### Equals Method:

The equals() method is used to compare two objects for **equality**.

* **Default Behavior**: By default, the equals() method (inherited from the Object class) compares memory addresses (i.e., references). If two references point to the same object in memory, equals() will return true.
* **Custom Behavior**: You can override equals() in your class to compare objects based on their logical state (i.e., their properties or fields). For example, if two objects have the same values for certain fields, you may consider them equal.

### Hashcode Method:

The hashCode() method returns an integer representation (a **hash code**) of an object. This hash code is used by hash-based collections like HashMap and HashSet to efficiently store and retrieve objects.

* **Default Behavior**: The default implementation of hashCode() (inherited from Object) typically returns a unique integer for each object based on its memory address.
* **Custom Behavior**: When overriding equals(), you should also override hashCode() to ensure that equal objects have the same hash code. This is crucial for the correct functioning of hash-based collections.

### Important Contract:

* **Consistency**: If two objects are considered equal according to equals(), they must have the same hashCode(). This ensures that objects that are equal will be placed in the same bucket when stored in a hash-based collection.
* **Different Objects**: Two objects that are not equal may (but are not required to) have different hash codes. However, it's generally a good idea to ensure that different objects have different hash codes to reduce collisions in hash-based collections.

### 

### **Why are they important:**

* **In Collections**: When using hash-based collections like HashSet or HashMap, hashCode() is used to quickly find the bucket where the object might be stored, and equals() is used to compare objects within the same bucket to find the exact match.
  + If only equals() is overridden but not hashCode(), two equal objects might end up in different hash buckets, causing them to be treated as unequal by collections like HashMap or HashSet.
* **General Rule**: Whenever you override equals(), you must also override hashCode() to maintain the general contract that:
  + If two objects are equal (equals() returns true), their hashCode() must return the same value.
  + If two objects are not equal, their hashCode() values should be different as much as possible, though it's not required.

### **Summary:**

* **equals()**: Determines if two objects are equal based on their logical state (fields).
* **hashCode()**: Generates an integer hash code used for efficient storage and retrieval in hash-based collections.
* **Contract**: Equal objects must have the same hash code, but unequal objects can have the same or different hash codes.

Together, these methods ensure proper behavior in collections like HashMap and HashSet, which rely on both equality and hash codes for performance and correctness.

## Shallow Clone and Deep Clone

In Java, **deep cloning** and **shallow cloning** refer to different ways of copying objects. These approaches determine how the copy handles the object's fields, particularly when the object contains references to other objects.

### **Shallow Clone:**

A **shallow clone** creates a new object, but does not recursively copy all objects referenced by the original object. Instead, it copies the references to these objects. This means that both the original and cloned objects share references to the same inner objects (fields that are references to other objects).

#### **How Shallow Cloning Works:**

* For **primitive fields** (e.g., int, boolean), the values are copied.
* For **object references** (e.g., String, List, custom objects), the references are copied, not the actual objects. Both the original and the clone refer to the same object in memory.

#### **Shallow Cloning with clone():**

Java provides the clone() method (from the Cloneable interface) to perform shallow cloning by default. You need to override the clone() method and call super.clone() to achieve this.

### **Deep Clone:**

A **deep clone** creates a completely independent copy of the original object and **all** objects referenced by it. This means that the cloned object and the original object do not share any references. Instead, the deep copy duplicates all objects referenced by the original object, creating a fully independent copy.

#### **How Deep Cloning Works:**

* **Primitive fields** are copied directly.
* **Object references** are not copied by reference. Instead, the referenced objects are recursively cloned, meaning that the new object gets its own independent copies of any nested objects.

#### **Deep Cloning Example:**

To implement deep cloning, you need to override the clone() method and manually clone all objects referenced by the object being cloned. You may also need to ensure that all referenced objects are themselves Cloneable.

### **Key Differences:**

| **Aspect** | **Shallow Clone** | **Deep Clone** |
| --- | --- | --- |
| **Object References** | Copies references of inner objects (same reference) | Creates new objects for referenced objects (different references) |
| **Performance** | Faster and less memory-intensive | Slower and more memory-intensive due to recursion |
| **Usage** | Suitable when shared references are acceptable | Used when complete independence between objects is required |
| **Impact of Change** | Changes in inner objects affect both the original and clone | Changes in inner objects don't affect the other |

### **Summary:**

* **Shallow Clone**: Only the top-level object is cloned, and the references to inner objects are copied. Changes to inner objects are reflected in both the original and cloned objects.
* **Deep Clone**: Both the top-level object and all inner objects are cloned. The clone is a fully independent copy, and changes to inner objects do not affect the original object.

Deep cloning provides better isolation between objects but is more complex and resource-intensive. Shallow cloning is simpler and faster but can lead to unintended side effects if the inner objects are modified.

## Serialization and Deserialization:

In Java, **serialization** and **deserialization** are mechanisms used to convert an object into a format that can be easily stored or transmitted, and then reconstructed later. This is especially useful when saving objects to files, sending objects over networks, or for other forms of persistent storage.

### **Serialization**

Serialization is the process of converting a Java object into a byte stream. This byte stream can then be written to a file, sent over a network, or stored in a database.

* **Why Serialize?**
  1. To persist the state of an object (e.g., saving it to a file).
  2. To send the object over a network (e.g., in a distributed application).
  3. To communicate between different Java Virtual Machines (JVMs).
* **How to Serialize an Object in Java?**
  1. The class whose objects need to be serialized must implement the java.io.Serializable interface.
  2. The ObjectOutputStream class is used to write the object to an output stream.

### **Deserialization**

Deserialization is the reverse process of converting a byte stream back into a Java object. This allows you to reconstruct the object from its serialized form.

* **Why Deserialize?**
  1. To retrieve a previously saved object from persistent storage.
  2. To receive and reconstruct an object sent over a network.
* **How to Deserialize an Object in Java?**
  1. The class must still implement the java.io.Serializable interface.
  2. The ObjectInputStream class is used to read the object from an input stream.

### **Key Points:**

* The **Serializable** interface is a marker interface with no methods.
* Objects are serialized to byte streams and can be deserialized back to objects.
* During serialization, the object's class, state, and data are saved.
* **Transient** fields in an object are not serialized.
* To customize the serialization process, you can implement readObject() and writeObject() methods.

This process helps in saving and transferring objects, maintaining their state across different sessions or systems.

## Memory Module:

In Java, memory management is primarily handled by the **Java Virtual Machine (JVM)**, which divides the memory into different regions based on the types of objects or data being stored. These memory areas form what is often referred to as the **Java Memory Model**.

The two main memory areas are:

### **1. Heap Memory:**

* The **heap** is used to store **objects** and **instance variables**.
* **Heap memory** is shared among all threads, which means that multiple threads can access the objects in the heap concurrently.
* The JVM automatically manages heap memory with the help of the **Garbage Collector** (GC), which reclaims memory that is no longer referenced by any part of the application.
* The heap is divided into regions:
  + **Young Generation**: Stores short-lived objects, and it is further subdivided into:
    - **Eden Space**: Where newly created objects are initially allocated.
    - **Survivor Spaces**: Two regions (S0 and S1) where objects that survive a garbage collection cycle in Eden are moved.
  + **Old Generation (Tenured Generation)**: Stores long-lived objects that survive multiple garbage collection cycles.
  + **Metaspace**: Stores the metadata about the classes used in the application. Unlike the earlier "Permanent Generation (PermGen)," metaspace grows dynamically depending on the class metadata.

### **2. Stack Memory:**

* **Stack memory** is used for storing **method calls** and **local variables** (including primitive data types and references to objects).
* Each thread has its **own stack**. As a method is called, a new stack frame is created in the memory containing method-specific local variables and references.
* Stack memory is automatically managed (last in, first out), and memory is freed as soon as a method call ends.
* If the stack memory limit is exceeded, a **StackOverflowError** is thrown.

### **3. Memory Model Regions:**

* **PC (Program Counter) Register**: Each thread has its own PC register that keeps track of the current instruction being executed. This is thread-specific and not part of the heap or stack.
* **Native Method Stack**: This is used when native (non-Java) methods are invoked. It’s separate from the Java stack.

### **Java Memory Management Summary:**

1. **Heap Memory**: Stores objects, managed by the garbage collector.
2. **Stack Memory**: Stores method calls and local variables, managed automatically by the JVM.
3. **Garbage Collection**: Periodically reclaims unused heap memory.

### 

### 

### **Heap vs. Stack Memory Overview:**

| **Aspect** | **Heap Memory** | **Stack Memory** |
| --- | --- | --- |
| **Stores** | Objects and instance variables | Method calls, local variables, and references |
| **Size** | Larger, dynamically resized | Smaller, fixed size for each thread |
| **Access** | Shared by all threads | Local to each thread |
| **Lifetime** | Managed by Garbage Collector (GC) | Memory is freed when the method call completes |
| **Speed** | Slower compared to stack due to global management | Faster as it follows a straightforward LIFO structure |
| **Errors** | OutOfMemoryError when heap is exhausted | StackOverflowError when stack memory limit is reached |

### **Garbage Collection:**

Garbage collection is a crucial part of Java’s memory management. It automatically frees up memory by removing objects that are no longer referenced by any part of the program. The GC operates mostly on the heap memory and uses various algorithms, such as **Mark-and-Sweep** and **Generational GC**, to manage the lifecycle of objects.

Java developers can also influence memory management using JVM arguments, for example:

* -Xms: Sets the initial heap size.
* -Xmx: Sets the maximum heap size.

#### Key Concepts:

1. **Automatic Memory Management**: Java’s GC automatically deallocates memory for objects that are no longer reachable.
2. **Heap Segmentation**: The heap is divided into several regions, mainly the **Young Generation** (for short-lived objects) and the **Old Generation** (for long-lived objects).
3. **Finalization**: Before an object is reclaimed, it may go through a process called "finalization" where it can perform clean-up operations, though this process is generally discouraged due to performance impacts.

#### Major Garbage Collection Algorithms in Java:

Java provides multiple garbage collection algorithms, and the most suitable one can be selected depending on the performance requirements of the application. Some of the commonly used algorithms are:

1. **Serial Garbage Collector (Serial GC)**:
   * Uses a single thread for garbage collection.
   * Works best for simple, single-threaded applications.
   * Divides the heap into young and old generations and uses a **mark-and-sweep** approach.
2. **Parallel Garbage Collector (Parallel GC)**:
   * Also known as **Throughput Collector**.
   * Uses multiple threads for garbage collection, especially during the stop-the-world events.
   * Best suited for applications that require high throughput but can tolerate occasional pauses.
3. **CMS Garbage Collector (Concurrent Mark-Sweep GC)**:
   * Aims to minimize the pauses during garbage collection by running most of its tasks concurrently with the application.
   * Works well for applications that require low latency.
   * Is a concurrent garbage collector but has been deprecated in favor of G1 GC.
4. **G1 Garbage Collector (Garbage First GC)**:
   * Designed for large heap memory and reduces the overall pause times.
   * Divides the heap into smaller regions and prioritizes regions that are most profitable to collect (hence "Garbage First").
   * Uses a combination of **marking**, **copying**, and **compacting** techniques to manage memory.
   * Replaces CMS and is currently the default garbage collector for Java from JDK 9 onwards.
5. **Z Garbage Collector (ZGC)**:
   * A low-latency garbage collector introduced in JDK 11.
   * Capable of handling heaps as large as several terabytes while keeping pause times minimal.
   * Uses **colored pointers** and concurrent techniques to minimize pause times.
6. **Shenandoah GC**:
   * Another low-pause garbage collector introduced in JDK 12.
   * Works similarly to ZGC in minimizing pause times, but focuses on parallel and concurrent operations.

#### Core Algorithms Used by Java Garbage Collectors:

* **Mark-and-Sweep**: Objects are "marked" as live, and then the unmarked (unreachable) objects are "swept" (removed) from memory.
* **Mark-and-Compact**: After marking live objects, it compacts them to eliminate memory fragmentation.
* **Copying**: Divides the heap into two and copies live objects from one half to the other, compacting memory as a result.
* **Generational Garbage Collection**: Segregates objects by their lifetime and collects young-generation objects more frequently since they tend to become unreachable sooner.

Java’s garbage collection strategy can be configured using various JVM options, allowing developers to choose the best-fit GC for their application's needs.

## Comparable and Comparator:

In Java, the Comparator interface is used to define a custom ordering of objects. It is part of the java.util package and allows you to sort collections of objects or arrays according to different criteria, separate from their natural ordering (if they implement the Comparable interface).

### **Key Features of Comparator:**

* **Custom Sorting Logic**: You can define multiple ways to sort objects, which is particularly useful when you want to sort the same objects differently in different situations.
* **Functional Interface**: Since Java 8, Comparator is a functional interface, meaning it can be implemented using lambda expressions.
* **Multiple Comparisons**: You can chain multiple Comparators to perform multi-level sorting.

### **Comparator Interface Methods:**

1. **compare(T o1, T o2)**:
   * This method compares two objects o1 and o2.
   * It returns:
     + A negative integer if o1 is less than o2,
     + Zero if o1 equals o2,
     + A positive integer if o1 is greater than o2.
2. **equals(Object obj)** (Inherited from Object):
   * This method checks whether a given object is "equal to" this comparator.

Since Java 8, some additional methods are available for convenience:

1. **reversed()**:
   * Returns a comparator that imposes the reverse of the current comparator.
2. **thenComparing(Comparator<? super T> other)**:
   * Used for multi-level sorting. It allows chaining comparators to compare secondary criteria if the primary comparison results in equality.
3. **Static factory methods** (introduced in Java 8):
   * **comparing(Function<T,U> keyExtractor)**: Returns a comparator that compares objects based on the result of a key extractor function.
   * **comparingInt(ToIntFunction<T> keyExtractor)**, **comparingDouble(ToDoubleFunction<T> keyExtractor)**, etc.: Specialized comparators for primitive data types.

### **When to Use Comparator:**

* **Custom sorting**: When you need multiple sorting strategies for the same class.
* **When the class doesn’t implement Comparable**: Use Comparator if you can't modify the class to implement Comparable (e.g., it's a third-party class).
* **For complex sorting**: When you need to sort on multiple fields or with custom logic that can change dynamically.

### 

### **Difference between Comparable and Comparator:**

| **Feature** | **Comparable** | **Comparator** |
| --- | --- | --- |
| **Purpose** | Defines natural ordering of objects. | Defines custom or multiple sorting orders. |
| **Implementation** | Implemented by the class itself using compareTo(). | Implemented in a separate class or inline, typically using compare(). |
| **Modifying Class** | You must modify the class to implement Comparable | No need to modify the class. |
| **Single vs Multiple Orders** | Defines a single sorting order. | Can define multiple sorting strategies. |
| **Example** | Sorting by a single field like name or ID. | Sorting by various fields or in different orders. |

In conclusion, Comparator is a flexible and powerful way to define multiple sorting strategies, especially when the Comparable interface doesn't meet your needs.

# Java 8:

## Functional Interface:

* Interface contains only one abstract method
* Can have any number of default, static methods
* @FunctionalInterface annotation to mark an interface
* Static method called by using the interface name preceding the method name
* Default method allow us to add new methods to an interface that are automatically available in the implementations

**Example:**

| **interface Interface1 {  default void doSomething() {  System.out.println("interface 1");  } }  interface Interface2 {  default void doSomething() {  System.out.println("interface 2");  } }  class Parent {  public void doSomething() {  System.out.println("interface 2");  } }  class MultiInheritance implements Interface1, Interface2 {  @Override  public void doSomething() {  Interface1.super.doSomething();  } }  class MultiInheritance2 extends Parent implements Interface2 {}  interface Interface4 {   void speak();   default void setColor(String color) {  System.out.println("Draw shape with color " + color);  }   static void setLeg(int leg) {  System.out.println("Animal has " + leg);  } }  class Inheritance implements Interface4 {   @Override  public void speak() {  Interface4.setLeg(5);  System.out.println("quack quack");  } }  public class DefaultStaticInterface {  public static void main(String[] args) {   Interface4 inheritance = new Inheritance();  inheritance.speak();  inheritance.setColor("red");   Interface4.setLeg(5);   MultiInheritance m = new MultiInheritance();  m.doSomething(); // Execute in Interface 1   MultiInheritance2 m1 = new MultiInheritance2();  m1.doSomething(); // Execute in Parent  } }** |
| --- |

## Method:

* Is always belongs to class or object in Java
* Method has the main parts: Name, Parameter list, Body, Return type

| class Person {  public void walk() {  System.out.println("walking");  } }  public class MyMethod {  public static void main(String[] args) {  Person person = new Person();  person.walk();  } } |
| --- |

## Lambda Expression:

The key point here like you say is lambdas allow method definitions to be used as variables/objects. Many other languages allow for function names to be passed as variables for callbacks and whatnot and Java never had a clean way to do that until lambdas. Only thing I would add to this is that putting the parameter type in the lambda expression is allowed and makes it easier to understand that the lambda is a method definition: (String p, String s) -> //code

* Is an anonymous function.
* Function without name and does not belong to any class.
* Is mainly used to implement functional interface
* Lambda has the main parts: Parameter list, Body, No Name, No Return Type

**Example:**

| **interface Shape {  void draw(); }  interface Addable {  int add(int a, int b); }**  **public class LambdaFunction {   public static void main(String[] args) {  // void method - one line  Shape rectangle = () -> System.out.println("draw rectangle");  rectangle.draw();   // return value method - multiple line  Addable addable = (a, b) -> {  int c = 10;  return a + b + c;  };  int result = addable.add(3,7);  System.out.println(result);   // with thread  Thread threadLambda = new Thread(() -> System.out.println("run method called using lambda ..."));  threadLambda.start();    } }** |
| --- |

## 

## Methods References:

Method references are a special type of lambda expressions..

There are four kinds of method references:

* Static methods:

| messages.forEach(StringUtils::capitalize); |
| --- |

* Instance methods of particular objects:

| createBicyclesList().stream().sorted(bikeFrameSizeComparator::compare); |
| --- |

* Instance methods of an arbitrary object of a particular type:

| numbers.stream().sorted(Integer::compareTo); |
| --- |

* Constructor

| bikeBrands.stream().map(Bicycle::new).toArray(Bicycle[]::new); |
| --- |

## Function & Consumer & Supplier & Predicate:

* Function is a functional interface that accepts a single input and returns output.
* Consumer is a functional interface that accepts a single input and returns no output.
* Supplier is a functional interface that has no input and returns output.
* Predicate is a functional interface that determines whether a value could be true or false

| public class FunctionalInterface {  public static Integer testFunctionParam(Function<String, Integer> fuc) {  return fuc.apply("hello moi nguoi");  }   public static void testConsumerParam(Consumer<String> consumer) {  consumer.accept("hello moi nguoi consumer");  }   public static LocalDateTime testSupplierParam(Supplier<LocalDateTime> supplier) {  return supplier.get();  }  public static void main(String[] args) {  MyFunctionInterface myFunctionInterface =  (String msg) -> System.out.println(msg);   myFunctionInterface.print("hello");   System.out.println(testFunctionParam((String s) -> s.length()));   testConsumerParam((String s) -> System.out.println(s));   System.out.println(testSupplierParam(() -> LocalDateTime.now()));   Predicate<Integer> lessThanEighteen = i -> (i < 18);  Predicate<Integer> greaterThanTen = i -> (i > 10);   System.out.println(greaterThanTen.test(11));   Predicate<Integer> greaterThanEighteen = lessThanEighteen.negate();   System.out.println(greaterThanEighteen.test(20));  } } |
| --- |

## Optional Class:

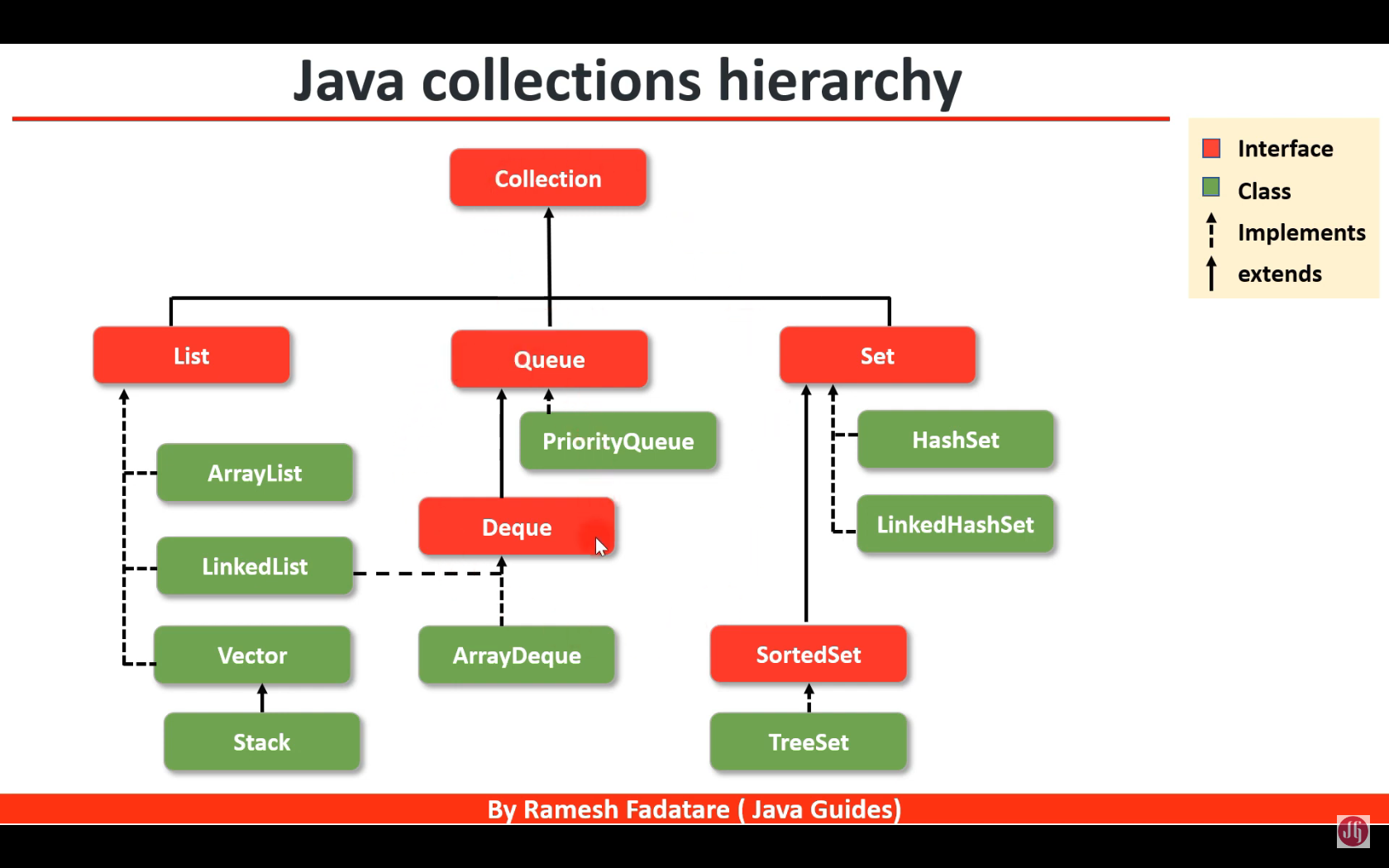
The Optional class in Java 8 is a container object which is used to contain a value that might or might not be present. It was introduced as a way to help reduce the number of NullPointerExceptions that occur in Java code. It is a part of java. util package and was added to Java as part of Java 8

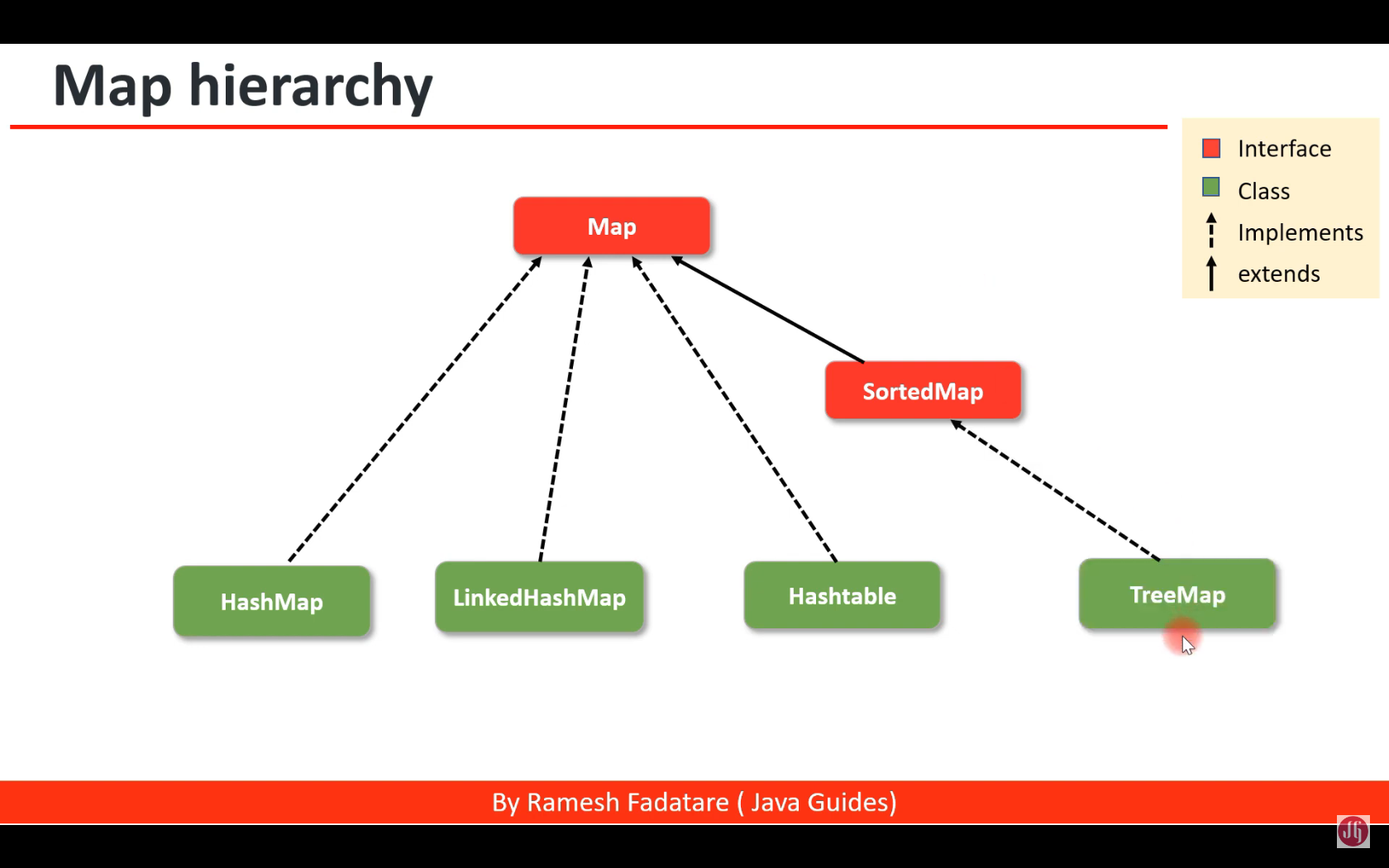
| class Customer {  private Long id;  private String firstName;  private String lastName;   public Long getId() {  return id;  }   public String getFirstName() {  return firstName;  }   public String getLastName() {  return lastName;  }   public Customer(Long id, String firstName, String lastName) {  this.id = id;  this.firstName = firstName;  this.lastName = lastName;  } }  public class OptionalClass {  public static void main(String[] args) {  Customer customer = new Customer(1234L, "Huy", null);   //of, empty, ofNullable  Optional<Object> emptyOptional = Optional.empty();  System.out.println(emptyOptional); // Optional.empty   Optional<String> firstNameOptional = Optional.of(customer.getFirstName());  System.out.println(firstNameOptional); // Optional[Huy]   Optional<String> lastNameOptional = Optional.ofNullable(customer.getLastName());  System.out.println(lastNameOptional); // Optional.empty   String defaultOptional = lastNameOptional.orElse("default");  System.out.println(defaultOptional); // default   String defaultOptional1 = lastNameOptional.orElseGet(() -> "default");  System.out.println(defaultOptional1); // default   String defaultOptional2 = lastNameOptional.orElseThrow(() -> new IllegalArgumentException("No such value")); //throw error  System.out.println(defaultOptional2);   Optional<String> gender = Optional.of("MALE");  gender.ifPresent((s) -> System.out.println("Value is present: " + s)); // Value is present: MALE  emptyOptional.ifPresent((s) -> System.out.println("no value present")); // Not print   List<String> result = List.of("abc", "def", "abcsae");  List<String> filteredResult = Optional  .of(result)  .stream()  .flatMap(Collection::stream)  .filter(s -> s.contains("abc"))  .collect(Collectors.toList());  System.out.println(filteredResult);   } } |
| --- |

## Collections:

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on data such as searching, sorting, insertion, manipulation, and deletion.





## Stream API:

Streams in Java enable data processing without altering the original list. Once consumed, streams cannot be reused, preventing data leakage and resource wastage.

# Spring Framework:

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

Examples of how you, as an application developer, can use the Spring platform advantage:

* Make a Java method execute in a database transaction without having to deal with transaction APIs.
* Make a local Java method a remote procedure without having to deal with remote APIs.
* Make a local Java method a management operation without having to deal with JMX APIs.
* Make a local Java method a message handler without having to deal with JMS APIs.

*PATH* and *CLASSPATH* are both environment variables. The operating system uses *PATH* to locate executable files, whereas the Java Virtual Machine (JVM) uses *CLASSPATH* to locate Java class files.*PATH* specifies the directories in which executables are located. *CLASSPATH*, on the other hand, specifies the directories and JAR files in which Java classes are located.

# Spring Boot:

## IOC container:

In Spring Boot (and Spring Framework in general), the **IoC (Inversion of Control) Container** is a core concept that manages the lifecycle and configuration of application components (beans). The IoC container is responsible for creating objects, configuring them, injecting their dependencies, managing their lifecycle, and ensuring that the dependencies between objects are handled correctly.

### **Key Concepts:**

1. **Inversion of Control (IoC)**:
   * IoC is a design principle where the control of object creation and dependency management is transferred from the developer to a container or framework (in this case, the Spring IoC container). Instead of manually creating and managing object dependencies, the IoC container handles this automatically.
   * The most common way IoC is implemented in Spring is through **Dependency Injection (DI)**.
2. **Dependency Injection (DI)**:
   * DI is a technique in which the IoC container injects the dependencies (or collaborators) that an object needs, rather than the object creating them itself. This helps in decoupling the code and making it more modular, testable, and maintainable.
   * Spring Boot supports **constructor injection**, **setter injection**, and **field injection**.

### **Types of IoC Containers:**

Spring offers two types of IoC containers that are commonly used:

1. **BeanFactory**:
   * A basic IoC container that provides the configuration framework and manages the lifecycle of beans.
   * It is lazy in nature, meaning it initializes beans only when they are requested. It is lightweight but lacks advanced features like event propagation, declarative transactions, or automatic BeanPostProcessor registration, which are supported by the ApplicationContext.
2. **ApplicationContext**:
   * This is an extension of BeanFactory and is the more commonly used IoC container in Spring Boot applications. It provides all the functionality of BeanFactory along with additional enterprise features such as AOP (Aspect-Oriented Programming), declarative transaction management, event listeners, and internationalization support.
   * In Spring Boot, the ApplicationContext is automatically created and managed by the framework when the application starts.

### **Key Responsibilities:**

1. **Bean Creation and Management**:
   * The IoC container creates and manages beans, which are objects that are instantiated, configured, and wired together by Spring. Beans are typically defined using annotations (like @Component, @Service, @Repository, and @Controller) or XML configuration.
2. **Dependency Injection**:
   * The IoC container is responsible for injecting dependencies into the beans, either through constructor injection, setter injection, or field injection.
3. **Bean Lifecycle Management**:
   * The container controls the lifecycle of the beans from creation, initialization, and destruction. Beans can implement lifecycle interfaces like InitializingBean and DisposableBean or use annotations like @PostConstruct and @PreDestroy to perform actions during these phases.
4. **Scope Management**:
   * The IoC container manages the scope of beans, such as singleton (default) or prototype. Other scopes include request, session, and application, especially in web applications.
5. **Automatic Bean Discovery and Wiring**:
   * Spring Boot uses annotations like @Autowired, @Component, and @Configuration to automatically discover beans and inject dependencies without requiring extensive configuration.
6. **Event Handling**:
   * The ApplicationContext can handle and propagate application events. For example, you can listen for events like context refresh, startup, or custom events through the ApplicationListener interface.

### **Summary of IoC Container Responsibilities:**

* **Manages beans**: Creates, configures, and manages the lifecycle of beans.
* **Inject dependencies**: Handles dependency injection between beans, reducing the need for manual object creation.
* **Provides additional features**: Manages transactions, events, and other enterprise-level concerns.

In Spring Boot, you don't need to configure the IoC container explicitly, as it is automatically set up for you when the application starts, making it simple to use dependency injection and other features provided by the Spring framework.

## IoC Singleton and Singleton:

### Key Differences

| **Aspect** | **Normal (Traditional) Singleton** | **Spring IoC Singleton** |
| --- | --- | --- |
| **Management** | Self-managed by the class itself using static methods | Managed by the Spring IoC container |
| **Instance Creation** | Created using a static field and a static method like getInstance() | Created and injected by Spring's IoC container automatically |
| **Initialization** | Can be eagerly or lazily initialized | Managed by Spring, typically eagerly initialized (though can be configured as lazy) |
| **Global Scope** | Globally accessible in the entire JVM | Singleton within the context of the Spring container; multiple instances may exist in different containers |
| **Thread Safety** | Developer is responsible for ensuring thread safety | Spring provides basic thread safety for singleton beans |
| **Flexibility** | Less flexible, hard-coded singleton pattern | Flexible, can change scope (singleton, prototype, request, session, etc.) easily |
| **Access Method** | Accessed through a static getInstance() method | Injected into other classes via dependency injection (e.g., @Autowired) |
| **Tight Coupling** | The singleton class tightly couples instance creation logic | The class itself has no responsibility for its own creation, making it loosely coupled |
| **Lifecycle Management** | Managed manually, including object destruction | Fully managed by Spring, including initialization and destruction |

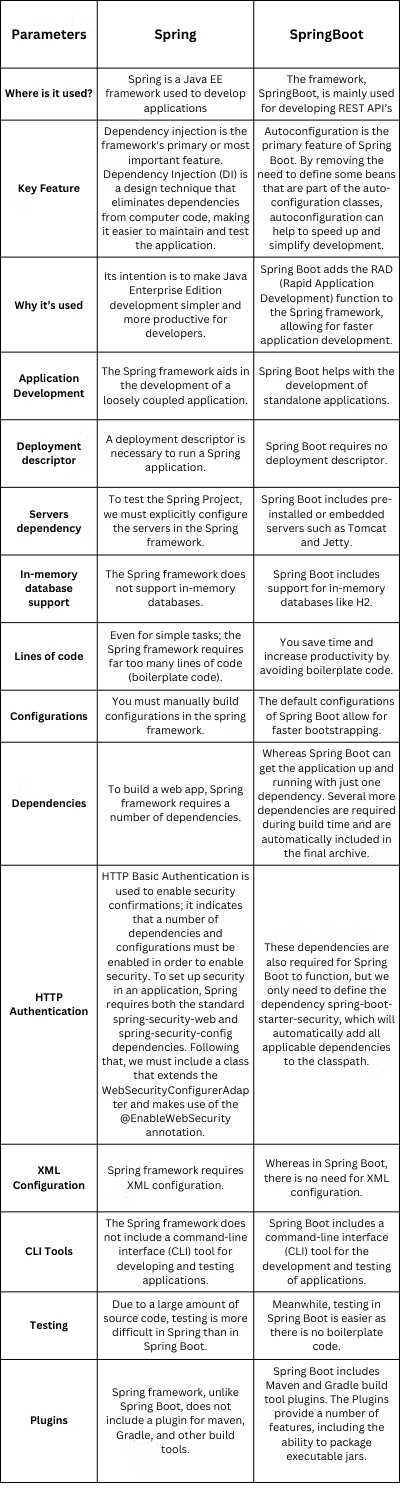
### **Key Considerations:**

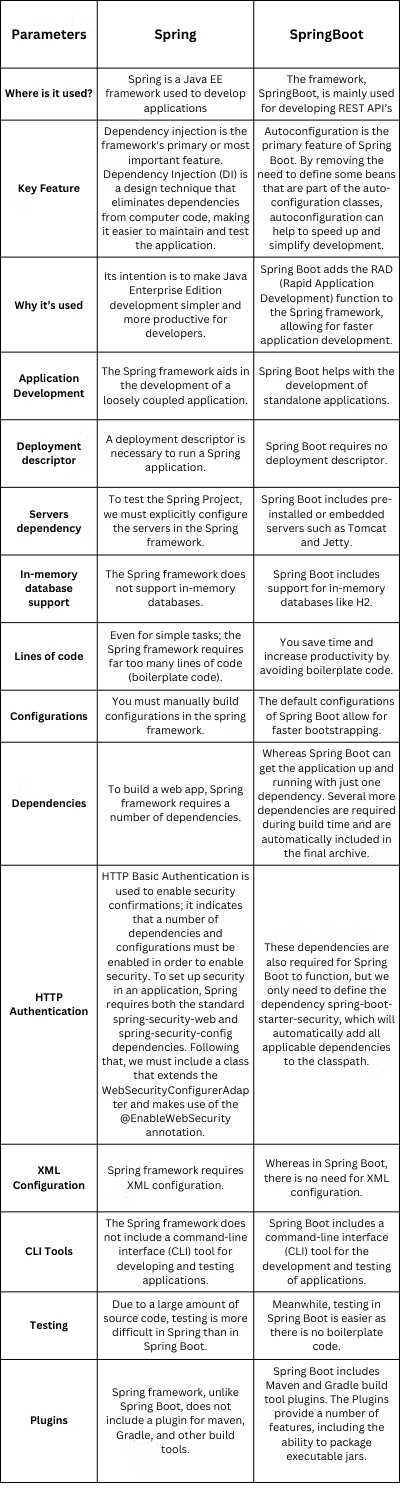
* **Use of Dependency Injection**: In Spring IoC, you don’t need to call static methods like getInstance(). Instead, Spring injects the singleton wherever it’s required. This allows for better **separation of concerns** and easier testing, as dependencies can be mocked or replaced.
* **Scope within Spring Container**: A Spring singleton is not truly global like a traditional Java singleton. It is **singleton within the Spring context**. Multiple Spring contexts could result in multiple instances of the singleton if they belong to different contexts.
* **Thread Safety**: In a traditional singleton, thread safety needs to be handled by the developer (e.g., through synchronization). In Spring, the IoC container handles the singleton bean and ensures basic thread safety.

### **Conclusion:**

* A **traditional singleton** is controlled by the class itself and is globally available through static methods.
* A **Spring singleton** is managed by the IoC container, and although it behaves as a singleton within the Spring context, it is injected through dependency injection and is not accessed through static methods. Spring's IoC singleton provides more flexibility, better modularity, and easier testing compared to the traditional singleton pattern.

## Spring vs SpringBoot



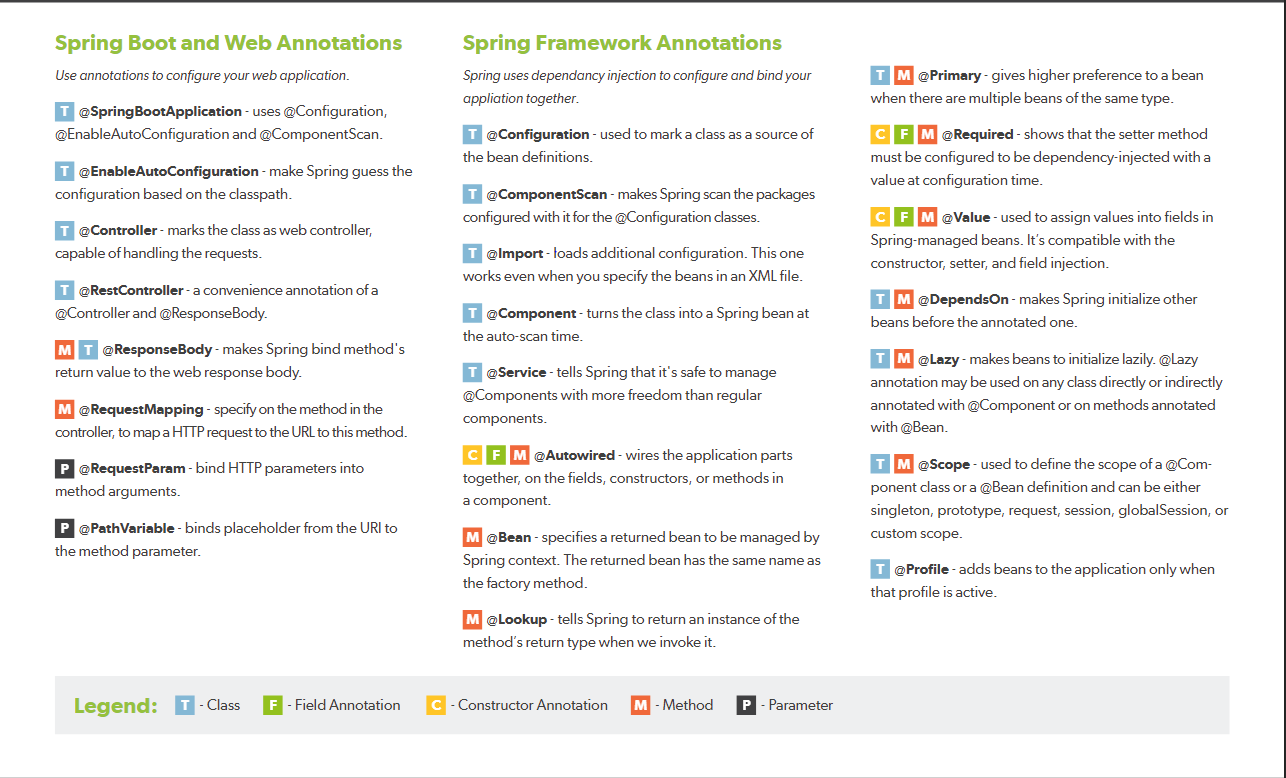


The three most basic parts of the Java ecosystem are the Java Virtual Machine (JVM), the Java Runtime Environment (JRE), and the Java Development Kit (JDK), which are *stock* parts that are supplied by Java implementations.

Moreover, IDE and libraries are also Java ecosystem

# Annotations:

In Java, **annotation** is a form of metadata that provides additional information about the code, but does not directly affect program logic. Annotations can be applied to classes, methods, variables, parameters, or other elements to convey instructions, provide metadata, or assist with code generation, documentation, and runtime processing.



## 

## Common Uses for Annotations:

1. **Code Documentation**: Annotations can serve as documentation for developers and tools. For example, the @Deprecated annotation indicates that a particular class or method is outdated and should no longer be used.
2. **Code Analysis**: Tools and frameworks can use annotations to perform static code analysis. For example, @Override helps the compiler detect when a method is meant to override a method in a superclass, catching errors if the method signature doesn’t match.
3. **Code Generation**: Annotations can be used by tools to generate code automatically. For instance, in frameworks like Lombok, annotations like @Getter and @Setter automatically generate getter and setter methods.
4. **Configuration**: Annotations are commonly used in frameworks to configure how components are managed. For example:
   * **Spring Framework**: Uses annotations like @Component, @Service, @Repository, and @Autowired for dependency injection and configuration.
   * **Java Persistence API (JPA)**: Uses annotations like @Entity, @Table, @Column, etc., to define the mapping between Java objects and database tables.
5. **Validation**: Annotations can be used to specify validation constraints on fields. For example, @NotNull, @Size, and @Pattern are used in bean validation (JSR 380) to enforce rules on data.
6. **Testing**: Testing frameworks use annotations to manage test cases. For instance, JUnit uses @Test, @Before, @After, and @Ignore to define and manage test methods and their setup/teardown.
7. **Framework Integration**: Annotations facilitate integration with various frameworks. For example, @RequestMapping in Spring MVC is used to map HTTP requests to handler methods.
8. **Dependency Injection**: Annotations like @Inject and @Autowired are used to inject dependencies into classes, simplifying dependency management and reducing boilerplate code.

# **OOP:**

The four principles of object-oriented programming (abstraction, inheritance, encapsulation, and polymorphism) are features that - if used properly - can help us write more testable, flexible, and maintainable code.

| **Abstraction** | The process of hiding implementation details and exposing only the functionality to the user. In abstraction, we deal with ideas and not events. This means the user will only know “what it does” rather than “how it does”.   * The class should be abstract if the class has one or many abstract method * An abstract class can have constructors, concrete methods, static method, and final method * Abstract class can’t be instantiated directly with the ***new*** operator. It can be:   A b = new B();   * The child class should override all the abstract methods of parent else the child class should be declared with abstract keyword |
| --- | --- |
| **Encapsulation** | The process of wrapping code and data together into a single unit.   * Declare the private variables * Declare getter and setter to get and set variable values |
| **Inheritance** | The process of one class inheriting properties and methods from another class in Java. Inheritance is used when we have **is-a** relationship between objects. Inheritance in Java is implemented using **extends** keyword.   * Single Inheritance * Multilevel Inheritance * Hierarchy Inheritance * Multiple Inheritance * Hybrid Inheritance |
| **Polymorphism** | The ability to perform many things in many ways. The word Polymorphism is from two different Greek words- poly and morphs. “Poly” means many, and “Morphs” means forms. So polymorphism means many forms. The polymorphism can be present in the case of inheritance also. The functions behave differently based on the actual implementation.   * Static and compile polymorphism: * Dynamic and runtime polymorphism |

# Interface vs Abstract:

## Interface

Interface in OOP (Object-Oriented Programming) is a contract or blueprint for classes that can contain only abstract methods, static methods, default methods and constants. Classes that implement the interface must provide implementations for the methods declared in the interface.

## Abstract

An abstract class is a class that cannot be instantiated directly. It is meant to be subclassed, and it can have both abstract methods (without implementation) and concrete methods (with implementation).

## The Difference

* **Instantiation**: You cannot instantiate an abstract class or an interface directly.
* **State and Behavior**: Abstract classes can have fields and constructors, and they can maintain state. Interfaces cannot have fields (only constants) and cannot maintain state.
* **Multiple Inheritance**: Java supports multiple inheritance through interfaces but not through classes. A class can implement multiple interfaces but can extend only one abstract class.
* **Default Methods**: Interfaces can provide default implementations for methods (from Java 8), while abstract classes can provide implementations for any methods.
* **Constructor**: Abstract classes can have constructors, which are called when a subclass is instantiated. Interfaces cannot have constructors.

## When to Use:

* **Abstract Class**: Use when you want to provide a common base with shared code and state that other classes can inherit. It’s useful when you have a base class with some default behavior and other related classes that need to extend this behavior.
* **Interface**: Use when you want to define a contract that can be implemented by any class, regardless of where it sits in the class hierarchy. It’s particularly useful for providing a common set of functionalities across unrelated classes or for enabling multiple inheritance.

# Access Modifier:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

# SOLID:

**S - Single responsibility:** a class should only have one responsibility. Furthermore, it should only have one reason to change.

**O - Open/Close:** Classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application.

**L - Liskov Substitution:** if class *A* is a subtype of class *B*, we should be able to replace *B* with *A* without disrupting the behavior of our program.

**I - Interface Segregation:** Larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

**D - Dependency Inversion:** The principle of dependency inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.

# Design pattern:

**Singleton:**

Singleton is a creational design pattern that lets you ensure that a class has only one instance and provide a global access point to this instance

**Factory:**

Factory Method is a creational design pattern that defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Builder:**

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

**Prototype:**

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

**Strategy:**

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

**State:**

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

**Observer:**

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

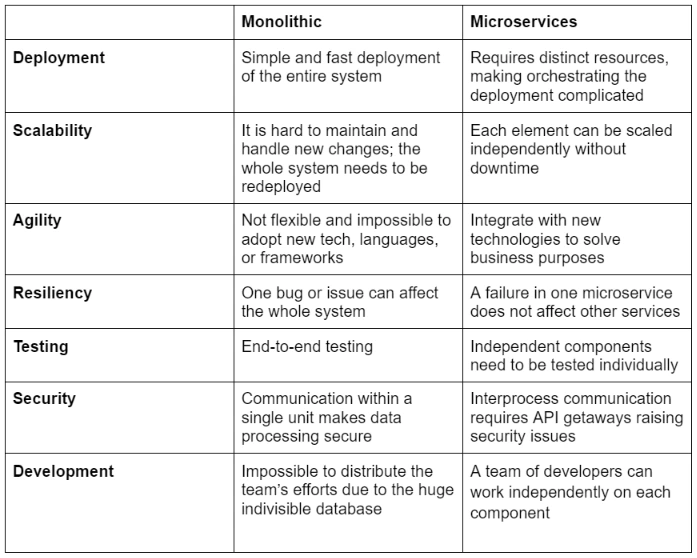
**Command:**

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue the request's execution, and support undoable operations.

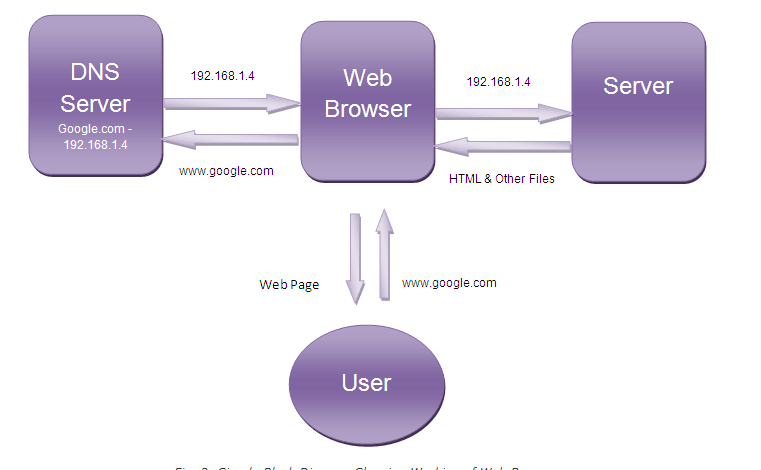
**Iterator:**

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

# Microservice vs Monolith:

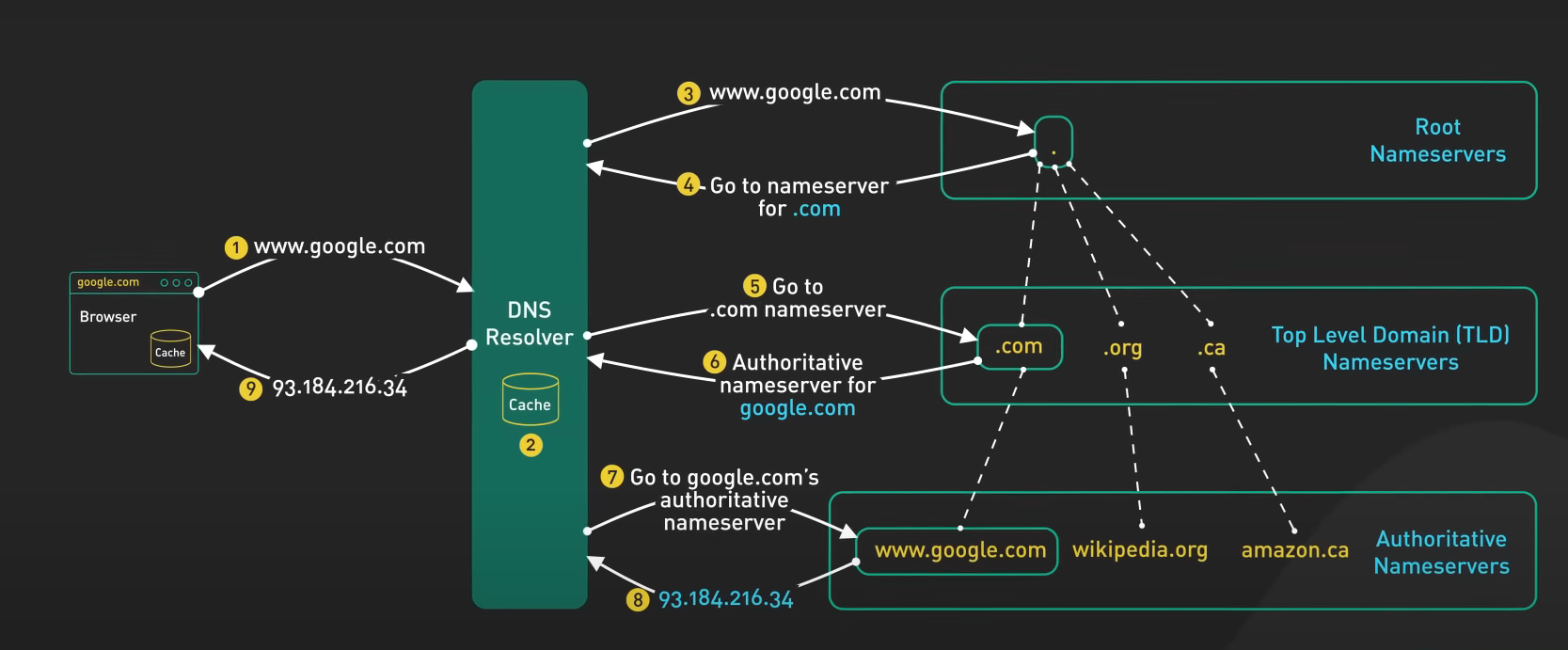


# Web Browser:



## DNS: Domain name system: translate domain names to IP address

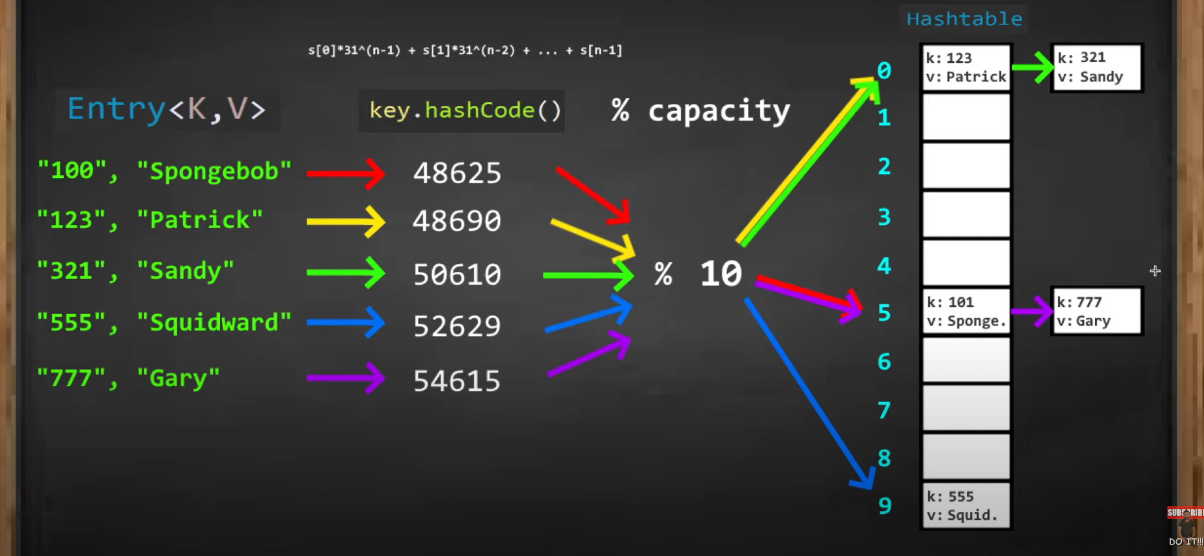
1. Check cache in browser. if not, call DNS resolver
2. Check cache in DNS resolver, if not, call Root name server (Root DNS)
3. Call Root name server
4. Return TLD .com
5. Call TLD (top level domain)
6. Return authoritative nameserver
7. Call Authoritative name server
8. Return IP address for DNS resolver
9. DNS resolver return for browser



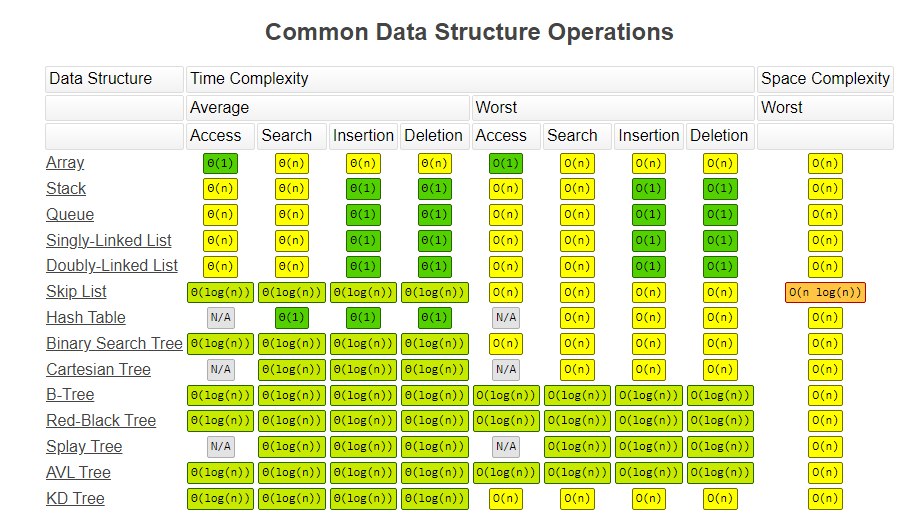
# Data Structure:

## Hashtable:

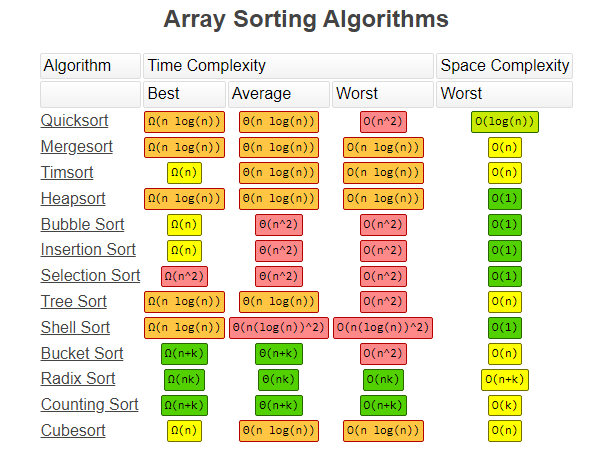
is an array, the key will be hash by modulus the capacity. But got problem with Collision, to solve that we need will store the linked list



## Complexity and Space:



## Sorting with array:



# Kafka:

Apache Kafka is the solution for the distributed and scalable problem.

Producer will update the message to queue so the consumer will consume this message.

In Queue, we have multiple partitions. the message will be distributed to partitions with the partition key.

All the partitions will have the same topic

Each message is a record and has an offset.

Each consumer need to have consumer group Id to know the current offset

# Improve Performance for application:

Improving the performance of a backend project can involve optimizing various aspects, including code, architecture, database interactions, and server configurations. Here’s a detailed guide to help you enhance your backend performance:

### **1. Optimize Database Queries**

* **Indexing**: Ensure your database tables are properly indexed, especially for frequently queried columns.
* **Use Efficient Queries**: Avoid unnecessary joins, and use efficient query structures (e.g., LIMIT, pagination).
* **ORM Optimization**: If using an ORM (e.g., Hibernate, Django ORM), be cautious of lazy loading and n+1 query problems.
* **Database Caching**: Use caching mechanisms like Redis or Memcached to store frequently accessed data.

### **2. Implement Caching**

* **Application-Level Caching**: Cache the results of expensive operations (e.g., API responses, complex calculations) to reduce load.
* **HTTP Caching**: Use ETags, Cache-Control, or other HTTP headers to control client-side caching.
* **Content Delivery Networks (CDNs)**: Serve static files (images, CSS, JS) through a CDN to offload server load and decrease latency.

### **3. Optimize Application Code**

* **Avoid Unnecessary Computation**: Remove redundant logic or heavy computations from frequently used endpoints.
* **Concurrency**: Make use of concurrency where appropriate. For example, in Node.js, asynchronous code can help improve throughput.
* **Lazy Loading**: Load data only when it is required instead of upfront loading.
* **Optimize Loops and Data Structures**: Ensure loops are efficient, and you are using optimal data structures for the task.

### **4. Reduce Response Time**

* **Minimize Payload Size**: Compress data sent over the network (e.g., use gzip for text responses).
* **Efficient Data Serialization**: Choose lightweight data formats like JSON, Protocol Buffers, or MessagePack.
* **Reduce External API Calls**: If external API calls are slow, cache their responses, or batch multiple requests into one.

### **5. Horizontal and Vertical Scaling**

* **Vertical Scaling**: Upgrade your server's hardware resources (e.g., more RAM, better CPU).
* **Horizontal Scaling**: Distribute the load across multiple servers or containers using load balancers.
* **Use Microservices**: If your application is monolithic and has performance bottlenecks, consider refactoring parts into microservices to scale independently.

### **6. Optimize Middleware and Routing**

* **Minimize Middleware**: Only use necessary middleware and avoid adding layers that increase response time.
* **Route Optimization**: Reduce unnecessary routing logic, and make sure critical paths are prioritized.

### **7. Asynchronous Processing and Queues**

* **Background Processing**: Move time-consuming operations like file processing, email sending, or heavy database queries to background jobs using queues (e.g., RabbitMQ, Celery, Bull).
* **Message Queues**: Use message queues to handle distributed tasks efficiently and prevent overloading.

### **8. Use Connection Pooling**

* **Database Connection Pooling**: Instead of opening a new connection for every request, use connection pooling to reuse existing connections.
* **API Connection Pooling**: When interacting with third-party APIs, use connection pools to reduce overhead.

### **9. Monitor and Profile Regularly**

* **Performance Monitoring Tools**: Use tools like New Relic, Prometheus, or Grafana to monitor the performance of your application.
* **Code Profiling**: Identify bottlenecks using profilers (e.g., Py-Spy for Python, Chrome DevTools for Node.js, etc.).
* **Logging**: Implement detailed logging to trace issues and diagnose performance bottlenecks.

### **10. Optimize Network Latency**

* **Reduce Number of Network Calls**: Bundle API requests when possible, and reduce the frequency of network requests.
* **Edge Locations**: Use edge servers to decrease latency for geographically distributed users (use AWS Lambda@Edge, Cloudflare Workers).

### **11. Security Considerations**

* **Rate Limiting and Throttling**: Implement rate limiting to prevent excessive requests that can cause performance issues.
* **DDoS Protection**: Use DDoS mitigation techniques like cloud-based firewalls or AWS Shield.

### **12. Server-Side Optimizations**

* **Optimize HTTP Headers**: Keep headers minimal to reduce network overhead.
* **Compression**: Enable Gzip/Brotli compression for text-based content like HTML, CSS, and JavaScript.
* **Use HTTP/2**: HTTP/2 offers multiplexing, header compression, and faster response times.
* **Tune Web Server Settings**: Adjust the configuration of web servers like Nginx or Apache for optimal thread pools and timeouts.

### **13. Use Advanced Techniques**

* **Event-Driven Architecture**: Instead of blocking threads, use event-driven or reactive programming models (e.g., Node.js, Spring WebFlux).
* **GraphQL**: Use GraphQL to fetch only the data you need instead of over-fetching with REST APIs.

Improving backend performance is an ongoing process, and adopting best practices like regular monitoring, refactoring, and scaling will ensure your project runs efficiently.

# Git Flow vs Trunk-Based

**Git Flow** and **Trunk-Based Development** are two different branching strategies used in software development. They cater to different workflows, team sizes, and release cycles. Here’s a breakdown of the key differences between them:

## 1. Branching Model:

* **Git Flow**:
  + Involves multiple branches: master (for production), develop (for ongoing development), and feature-specific branches.
  + Follows a more structured branching model:
    - feature/: for feature development
    - release/: for preparing releases
    - hotfix/: for urgent fixes
    - develop/: integration branch
    - master/: production-ready code
  + Releases are made by merging from develop to master.
* **Trunk-Based Development**:
  + Focuses on a single shared branch (typically main or master), where all developers frequently commit small changes directly.
  + Feature development happens in short-lived branches or directly on the main branch.
  + Emphasizes continuous integration (CI) and rapid merging into the trunk (main branch).

## 2. Commit Frequency and Size:

* **Git Flow**:
  + Encourages larger, less frequent merges due to longer-lived branches (e.g., feature branches can last days or weeks).
  + Developers work on isolated feature branches, which may only be merged into the develop branch when complete.
* **Trunk-Based Development**:
  + Encourages small, frequent commits and merges to the main branch, typically multiple times a day.
  + Developers commit as often as possible, with a focus on keeping the trunk stable and releasable at all times.

## 3. Release Strategy:

* **Git Flow**:
  + Structured around versioned releases.
  + Feature branches are only merged into develop, and release branches are created from develop. The release branch is then merged into master once the release is ready.
  + Works well with scheduled releases (e.g., every sprint or milestone).
* **Trunk-Based Development**:
  + Continuous deployment is often the goal, meaning the trunk is always in a deployable state.
  + Can support frequent releases, even multiple times a day, without needing feature or release branches.
  + Feature flags are often used to control which features are released without relying on branching.

## 4. Complexity and Overhead:

* **Git Flow**:
  + More complex with multiple branches and branching rules (features, releases, hotfixes, etc.).
  + Suitable for teams that want more control over their release cycle, especially for long-term projects or those that require stability in multiple environments.
* **Trunk-Based Development**:
  + Simpler and more streamlined, but requires discipline to ensure code is always stable.
  + Lower overhead due to fewer branches and less merging.
  + More suitable for teams that value speed, simplicity, and can maintain high-quality standards for code directly on the trunk.

## 5. Collaboration and Team Size:

* **Git Flow**:
  + Works well for larger teams or teams with longer release cycles, where developers work on independent features for extended periods.
  + Allows isolated development, which may reduce merge conflicts but can also lead to integration problems if branches diverge significantly.
* **Trunk-Based Development**:
  + Works better for smaller, fast-moving teams that can collaborate closely and handle frequent integration.
  + Encourages resolving conflicts early by integrating changes frequently into a shared branch, avoiding large, complex merges later.

## 6. Use Cases:

* **Git Flow**:
  + Well-suited for projects with structured, slower release cycles (e.g., enterprise software, traditional product development).
  + Provides clear branching strategies for hotfixes and emergency releases.
  + Good for teams that need multiple release environments (e.g., staging, production).
* **Trunk-Based Development**:
  + Ideal for teams practicing Continuous Integration (CI) and Continuous Deployment (CD).
  + Suited for teams aiming for fast delivery, such as startups, SaaS products, or teams focused on frequent releases.
  + Often used in modern DevOps practices where automation and rapid feedback loops are key.

## 7. Summary:

| **Aspect** | **Git Flow** | **Trunk-Based Development** |
| --- | --- | --- |
| **Branching Model** | Multiple branches (master, develop, etc.) | Single branch (main or master) |
| **Commit Frequency** | Less frequent, larger merges | Frequent, small commits to trunk |
| **Release Strategy** | Structured around feature/release/hotfix branches | Continuous deployment, always releasable trunk |
| **Complexity** | Higher complexity, more overhead | Simpler, lower overhead |
| **Collaboration** | Good for larger teams, isolated development | Better for smaller, fast-moving teams |
| **Use Case** | Structured projects with longer release cycles | Fast-paced, continuous delivery environments |

Both strategies have their strengths, and the choice depends on the team's workflow, project type, and release cadence.

# Database:

## Sharding vs Replication vs Partitioning

**Database sharding**, **replication**, and **partitioning** are two different techniques used to improve the performance, availability, and scalability of databases. While both involve spreading data across multiple systems, they achieve these goals in different ways and for different purposes.

### **Database Sharding**

Sharding is a form of **horizontal partitioning** where the data is divided across multiple databases or servers, with each shard holding a portion of the total data. Each shard operates independently, handling a subset of the overall dataset.

#### **Key Characteristics:**

* **Horizontal partitioning**: Data is split across multiple databases based on some criteria (e.g., user ID, geographical location).
* **Independent databases**: Each shard holds a different part of the dataset.
* **Scalability**: More shards can be added as the data grows, helping to scale the database horizontally.
* **Complexity in queries**: Queries across shards can become complex, especially when dealing with joins or transactions involving multiple shards.
* **Data isolation**: Each shard is typically unaware of the other shards, leading to better fault isolation. If one shard fails, only part of the data is affected.

#### **Example:**

* A company with millions of users might shard its database by user ID. Users with IDs between 1 and 1 million are stored in one database, while users with IDs from 1 million to 2 million are stored in another.

#### **Pros:**

* **Massive horizontal scalability**: Allows you to scale databases across multiple servers as data grows.
* **Improved performance**: Each shard contains a smaller portion of the data, reducing the load on individual databases.
* **Data isolation**: Failures in one shard affect only a portion of the data.

#### **Cons:**

* **Complexity**: Managing multiple shards introduces operational complexity, such as shard rebalancing, routing queries, and handling cross-shard transactions.
* **Cross-shard queries**: Joins and transactions across shards are difficult and inefficient.

### **Database Replication**

Replication involves **copying and synchronizing data** across multiple database servers. The goal is to ensure high availability, fault tolerance, and improve read performance by distributing read loads across multiple copies (replicas).

#### **Key Characteristics:**

* **Data duplication**: The same data is stored on multiple servers (replicas).
* **High availability**: If one replica fails, another replica can take over, ensuring the system remains available.
* **Load distribution**: Replication is often used to distribute read operations across multiple servers, reducing the load on the master or primary database.
* **Eventual consistency (in asynchronous replication)**: In some forms of replication (asynchronous), there may be a lag between when a write occurs and when it’s reflected across all replicas, leading to eventual consistency.

#### **Types of Replication:**

* **Master-slave replication**: One server (master) handles writes, while replicas (slaves) handle read operations.
* **Master-master replication**: Multiple servers can handle both reads and writes, but conflict resolution can be more complex.

#### **Example:**

* A website with a global user base might replicate its primary database to multiple data centers around the world. Reads can be directed to the nearest replica for lower latency, while the master database handles all writes.

#### **Pros:**

* **Improved read performance**: Distributing read operations across multiple replicas improves scalability for read-heavy applications.
* **High availability**: If one replica fails, another can take over, ensuring minimal downtime.
* **Disaster recovery**: Replication provides redundancy and backup for disaster recovery.

#### **Cons:**

* **Storage overhead**: Each replica requires additional storage, duplicating the data.
* **Consistency issues**: In asynchronous replication, there may be some lag, leading to inconsistencies between replicas.
* **Complex writes**: In master-master replication, resolving conflicts between simultaneous writes is challenging.

### **Database Partitioning**

Partitioning is the process of dividing a large database table into smaller, more manageable pieces (partitions). Unlike sharding, partitioning typically occurs within the same database instance or server. Each partition contains a subset of the data based on some partitioning scheme (e.g., range, list, or hash partitioning).

#### **Key Characteristics:**

* **Logical division of a single database**: Data is divided into smaller pieces within the same database.
* **Single database management**: All partitions belong to the same database instance, reducing the complexity of managing multiple databases.
* **Partitioning strategies**: Common strategies include:
  + **Range partitioning**: Dividing data based on a range of values (e.g., date ranges).
  + **List partitioning**: Dividing data based on specific values (e.g., region).
  + **Hash partitioning**: Using a hash function to distribute data across partitions evenly.

#### **Example:**

* A large e-commerce company might partition its orders table by date. Orders from 2022 might be in one partition, while orders from 2023 are stored in another partition. This allows queries for recent orders to focus only on the relevant partition, improving performance.

#### **Pros:**

* **Improved query performance**: Queries can be directed to specific partitions, reducing the amount of data scanned.
* **Manageability**: Large tables are broken down into smaller, more manageable pieces, improving maintenance and performance.
* **Optimized storage**: Different partitions can be stored on different storage media, optimizing cost and performance.

#### **Cons:**

* **Limited scalability**: Partitioning is limited to the capabilities of a single database server. It does not provide the horizontal scalability that sharding offers.
* **Complexity in managing partitions**: Adding, merging, or rebalancing partitions can add operational complexity, though less so than sharding.

### Key Differences:

| **Aspect** | **Sharding** | **Replication** | **Partitioning** |
| --- | --- | --- | --- |
| **Purpose** | Horizontal scalability by splitting data across multiple databases. | High availability and load distribution by duplicating data. | Logical division of a single table within the same database. |
| **Data distribution** | Each shard holds a different subset of the data. | Each replica holds the same copy of the data. | Each partition holds a different subset of the data (in the same database). |
| **Scalability** | High scalability, can add more shards to scale horizontally. | No horizontal scalability, mainly improves read performance. | Limited scalability within a single server, mainly improves query performance. |
| **Availability** | Not inherently fault-tolerant; failure of one shard affects part of the data. | High availability; replicas can take over if one fails. | Does not directly improve availability, but can improve performance. |
| **Data size** | Suitable for very large datasets that don’t fit on a single server. | Suitable for medium datasets where read performance or availability needs to be improved. | Suitable for large tables that can benefit from improved query performance. |
| **Use cases** | Large-scale applications with large datasets that need horizontal scalability. | Applications requiring high availability, fault tolerance, and read scalability. | Large tables where partitioning can improve query speed or management. |
| **Complexity** | High operational complexity: routing, cross-shard joins, and rebalancing. | Lower complexity for reads, but potential complexity in writes (especially master-master replication). | Lower complexity, but requires careful partition design for optimal performance. |
| **Write consistency** | No impact on write consistency (but cross-shard transactions can be complex). | In master-master, consistency issues and conflict resolution can be problematic. | No impact, as partitions belong to the same database. |
| **Fault tolerance** | Limited to the shard level (failure of a shard affects part of the data). | Highly fault-tolerant, with failover between replicas. | No built-in fault tolerance, relies on the single database instance. |

### **Summary:**

* **Sharding**: Best suited for massive datasets that need to be **horizontally partitioned** across multiple databases to handle growth in data and traffic. Complex to implement but offers significant scalability.
* **Replication**: Ideal for **high availability** and **read performance**. Data is duplicated across multiple servers, allowing failover and distributed reads. However, replication doesn't help with scaling large datasets or writes.
* **Partitioning**: Best for dividing large tables into smaller segments **within a single database** to improve query performance. It doesn't provide the horizontal scalability of sharding but is easier to implement.

Each technique serves a different purpose, and they can sometimes be used in combination (e.g., sharding with replication) to handle specific needs of large, distributed systems.

# Interview QA:

# HashSet - HashMap - HashTable:

| Hash Set | HashMap | HashTable - ConcurrentHashMap | TreeMap |
| --- | --- | --- | --- |
| * Implement Set interface * Store data as Objects * Internally uses Hashmap * Does not allow duplicate elements * Allow only one null element * Performance O(1), Space is O(n) | * Not thread safe * Implement Map interface * Store data as key-value pairs * Internally uses an array of Entry<K, V> objects * Does not allow duplicate keys, but allow duplicate values * Allow only 1 null key but multiple null values * Performance O(1), Space is O(n) | * Thread safe * Do not allow null value and null key * Slightly slower * Synchronized * Performance O(n), Space is O(n) | * Implement Map interface * Not allow null keys, but can multiple null values * Performance O(log(n)), Space is O(n) * Sorted |
|
|
|
|

## ArrayList - LinkedList:

| Feature | ArrayList | LinkedList |
| --- | --- | --- |
| Data Structure | Dynamic array | Doubly linked list |
| Access Time (get) | Fast (O(1)) | Slow (O(n)) – requires traversal |
| Insertion at End (add) | Fast (O(1)) if no resizing is required; O(n) if resizing occurs | Fast (O(1)) |
| Insertion in Middle | Slow (O(n)) – elements need to be shifted | Moderate (O(n)) – requires traversal but no shifting |
| Deletion in Middle | Slow (O(n)) – elements need to be shifted | Moderate (O(n)) – requires traversal but no shifting |
| Random Access | Fast (O(1)) – direct access by index | Slow (O(n)) – must traverse from head or tail |
| Iteration Performance | Fast – better cache locality due to contiguous storage | Slower – poorer cache locality due to node-based storage |
| Resize Cost | Must resize and copy elements when capacity is exceeded | No resizing needed |
| Best Use Case | When frequent access by index is needed and few insertions/deletions | When frequent insertions/deletions are needed in the middle or ends |

## Fail-safe and Fail-fast:

Fail-Fast systems abort operation as-fast-as-possible exposing failures immediately and stopping the whole operation. ArrayList, HashMap, HashSet, ...

Whereas, Fail-Safe systems don’t abort an operation in the case of a failure. Such systems try to avoid raising failures as much as possible. ConcurrentHashMap, CopyOnWriteArrayList,...

## Streams and Collections:

| **STREAMS** | **COLLECTIONS** |
| --- | --- |
| It doesn’t store data, it operates on the source data structure i.e collection. | It stores/holds all the data that the data structure currently has in a particular data structure like Set, List or Map, |
| They use functional interfaces like lambda which makes it a good fit for programming languages. | Don’t use functional interfaces. |
| Java Streams are consumable i.e; to traverse the stream, it needs to be created every time. | Non-consumable i.e; can be traversable multiple times without creating it again. |
| Java streams support both sequential and parallel processing. | Supports parallel processing and parallel processing can be very helpful in achieving high performance. |
| All the Java stream API interfaces and classes are in j**ava.util.stream** package. | Specific classes for primitive types such as **IntStream**, **LongStream**, and **DoubleStream** are used in collections since primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time. |
| Streams are not modifiable i.e one can’t add or remove elements from streams. | These are modifiable i.e one can easily add to or remove elements from collections. |
| Streams are iterated internally by just mentioning the operations. | Collections are iterated externally using loops. |

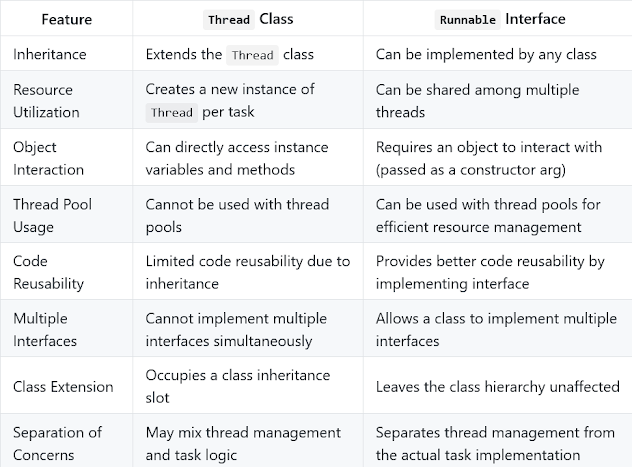
## Final keyword:

* Can the final function be inherited? - Yes, But cannot overriding this function

## What is static class:

A class can be said to be a static class if all the variables and methods of the class are static and the constructor is private. Making the constructor private will prevent the class from being instantiated. So the only possibility to access is using the Class name only.

## Thread vs Runnable



## N + 1 problem:

The **N+1 problem** is a common performance issue in databases, particularly when dealing with **ORMs (Object-Relational Mappers)** like Hibernate, Django ORM, or Active Record. It happens when an application needs to fetch related data, leading to multiple database queries where one or fewer would suffice.

The N+1 Problem:

Imagine you have two tables: **authors** and **books**, where each author has multiple books. If you query for a list of authors, and for each author, you then query the database for their associated books, you’ll execute **N+1 queries** (1 query to get all authors and N queries to get books for each author).

Example:

**Query 1**: Fetch all authors.  
SELECT \* FROM authors;

For each author, you execute another query to get their books:  
SELECT \* FROM books WHERE author\_id = 1;

SELECT \* FROM books WHERE author\_id = 2;

...

SELECT \* FROM books WHERE author\_id = N;

So, for **N authors**, you are making **N+1 queries**. This results in inefficiency, especially as **N** grows large, leading to performance bottlenecks.

Solution:

To fix the N+1 problem, you typically optimize your queries using **eager loading** or **JOINs**, so that related data is fetched in fewer queries.

1. **Eager Loading**: Instead of lazily fetching related data, eager loading fetches all related records in a single query.

Using the @OneToMany or @ManyToOne annotations, you can set the fetch type to EAGER to force loading of related entities in a single query.

1. **JOINs**: Use SQL JOIN to fetch related data in a single query.

SQL Example:  
SELECT authors.\*, books.\*

FROM authors

LEFT JOIN books ON authors.id = books.author\_id;

1. This query fetches both authors and their books in one go.
2. **Batching**: In some cases, you can optimize by batching queries, though this is less common compared to eager loading.

By using eager loading or JOINs, you reduce the number of queries to just 1 or 2, instead of N+1. This significantly improves performance for large datasets.

## Try with Resource:

In Java, **try-with-resources** is a feature introduced in **Java 7** that allows developers to manage resources (like files, database connections, sockets, etc.) automatically, ensuring that they are properly closed after use. The try-with-resources statement is particularly useful when working with resources that must be closed after their work is done (like InputStream, OutputStream, Reader, Writer, Socket, etc.).

### **Key Points of Try-With-Resources:**

* The resource must implement the AutoCloseable interface (or the older Closeable interface).
* The try-with-resources block automatically closes the resource, so there’s no need to explicitly call the close() method.
* Resources are closed in reverse order of their declaration, ensuring that dependent resources are closed properly.

### **Benefits of Try-With-Resources:**

1. **Automatic Resource Management**: The main advantage is that resources (like streams, files, and database connections) are automatically closed, reducing the chance of resource leaks.
2. **Cleaner Code**: It simplifies the code by avoiding explicit try-finally blocks for closing resources.
3. **Improved Exception Handling**: If an exception is thrown, the resource’s close() method is still called, even if another exception occurs in the block.

### **Example Without Try-With-Resources:**

Before try-with-resources, you would have to manually close resources using try-finally blocks:

## Type safety problem:

public void testMethod1(List<? extends Date> list, Date date) {

Date now = list.get(0);

System.*out*.println(now);

list.add(date); -> got problem

}

The method accepts a list of ? extends Date, which means it can contain elements of any type that is a subclass of Date. However, using ? extends Date makes the list immutable for adding new elements. You can read from the list but cannot add to it because the compiler cannot guarantee type safety.

Solution: You cannot add an element (date in this case) to a list that is declared with ? extends Date. To fix the issue, you should remove the wildcard or change the structure of the method, depending on what behavior is desired.

public void testMethod1(List<? super Timestamp> list) {

Timestamp timestamp = new Timestamp(System.*currentTimeMillis*());

list.add(timestamp);

Timestamp time = list.get(0); -> got problem

}

The issue in this code lies in the way generics are used. The method signature specifies a wildcard ? super Timestamp, which means the list can accept objects of type Timestamp or any of its superclasses, but you cannot safely retrieve elements as a Timestamp.

## Assessment 1

This problem can be solved by focusing on how many modules the employee can complete in such a way that no two consecutive modules come from the same project.

The key constraint is that modules chosen on two successive weeks must come from different projects. Here’s a step-by-step algorithm to solve the problem:

Step-by-Step Algorithm

Input:

The number of projects N.

For each project i, you have a number Ci which represents the number of modules in project i.

Observation:

If there is only one project, the employee can complete the modules continuously for Ci weeks without any restrictions.

If there are multiple projects, the employee needs to switch between them, so we need to consider how to schedule the modules effectively to maximize the number of weeks the employee can work.

Solution Strategy:

The employee should work on as many modules as possible by switching between projects.

To avoid selecting two successive modules from the same project, the maximum number of modules in any one project must not exceed the combined number of modules in all the other projects by more than 1. This is because if one project has significantly more modules than others, the employee would be forced to select modules from that project consecutively, which violates the rule.

Greedy Scheduling:

Let maxC be the project with the maximum number of modules.

Let sumOtherC be the total number of modules in all other projects combined.

If maxC is less than or equal to sumOtherC + 1, the employee can alternate between projects for all sum(Ci) modules.

Otherwise, the employee can complete at most 2 \* sumOtherC + 1 weeks (since there would be no way to alternate after exhausting all other projects).

Algorithm Steps:

Parse the input.

Calculate the total number of modules across all projects.

Find the project with the maximum number of modules (maxC).

Calculate the sum of modules in the other projects (sumOtherC).

If maxC is less than or equal to sumOtherC + 1, the result is sum(Ci) weeks (i.e., the total number of modules).

Otherwise, the result is 2 \* sumOtherC + 1 weeks.

Example

Input:

N = 3 (3 projects)

C = [3, 2, 2] (Project 0 has 3 modules, Project 1 has 2 modules, Project 2 has 2 modules)

Output:

The employee can work for 7 weeks.

Pseudocode

python

Copy code

def max\_weeks(N, C):

total\_modules = sum(C) # Total number of modules

maxC = max(C) # Maximum number of modules in any one project

sumOtherC = total\_modules - maxC # Sum of modules in other projects

# If the project with the most modules can be alternated with others

if maxC <= sumOtherC + 1:

return total\_modules # All modules can be completed

else:

return 2 \* sumOtherC + 1 # Max possible weeks with alternating projects

# Example usage

N = 3

C = [3, 2, 2]

print(max\_weeks(N, C)) # Output: 7

Explanation of Example

Total modules: 3 + 2 + 2 = 7

maxC = 3 (Project 0 has the most modules).

sumOtherC = 2 + 2 = 4 (sum of the other two projects).

Since maxC <= sumOtherC + 1, the employee can alternate between projects and complete all 7 modules in 7 weeks.

Time Complexity

The time complexity of this solution is O(N), where N is the number of projects. This is because we only need to compute the sum and the maximum, both of which take linear time.

Let me know if you need further clarifications!

## Assessment 2

The screenshots you uploaded show a problem statement and partially implemented Java code. The problem is asking to write a function that outputs the most frequently occurring character in different segments of a string based on multiple input pairs of indices. Let me help you structure this solution.

Problem Breakdown

Input:

A string strS of lowercase English letters.

Two integers numPair (number of index pairs) and valPair (always 2).

numPair number of index pairs [L, R], where for each pair, we need to find the most frequent character in strS[L:R].

Output:

For each range [L, R], print the most frequent character. If multiple characters have the same frequency, return the lexicographically smallest character.

Steps:

Parse the input.

For each pair of indices, extract the substring strS[L-1:R] (adjusted for 0-based indexing).

Count the frequency of each character in the substring.

Identify the character with the highest frequency. In case of a tie, choose the lexicographically smaller one.

Print the result for each pair.

Modifying Your Code:

mostFrequent Method: This function should take the string strS and the list of pairs, and return a string where each character corresponds to the most frequent character in each substring defined by the input pairs.

Counting the Frequencies: Use a frequency array of size 26 (for each letter from 'a' to 'z').

Main Method: Read inputs, call the mostFrequent function, and output the results.

Here is a suggested code structure:

java

Copy code

import java.util.Scanner;

public class Solution {

// Method to find most frequent character in each pair's range

public static String mostFrequent(String strS, int[][] pairList) {

StringBuilder answer = new StringBuilder();

// Iterate over each pair

for (int[] pair : pairList) {

int L = pair[0] - 1; // Convert to 0-based index

int R = pair[1] - 1;

// Frequency array for characters 'a' to 'z'

int[] freq = new int[26];

// Calculate frequency of characters in the range strS[L..R]

for (int i = L; i <= R; i++) {

freq[strS.charAt(i) - 'a']++;

}

// Find the most frequent character (lexicographically smallest in case of tie)

int maxFreq = 0;

char result = 'z'; // Initialize with a high lexicographical value

for (int i = 0; i < 26; i++) {

if (freq[i] > maxFreq || (freq[i] == maxFreq && (char) (i + 'a') < result)) {

maxFreq = freq[i];

result = (char) (i + 'a');

}

}

// Append the result to the answer string

answer.append(result);

}

return answer.toString();

}

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

// Input first line (the string)

String strS = in.nextLine();

// Input second line (numPair and valPair)

int numPair = in.nextInt();

int valPair = in.nextInt(); // This is always 2

// Input all pairs

int[][] pairList = new int[numPair][valPair];

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

pairList[i][0] = in.nextInt(); // valL

pairList[i][1] = in.nextInt(); // valR

}

// Get the result string of most frequent characters

String result = mostFrequent(strS, pairList);

// Output the result

System.out.println(result);

in.close();

}

}

Explanation of Changes:

Frequency Counting:

We use an array freq[26] to count the occurrences of each letter in the range [L, R].

The loop iterates through the substring strS[L:R] and populates the frequency array.

Lexicographical Ordering:

We find the most frequent character by iterating through freq[] and selecting the lexicographically smallest one in case of a tie.

Input Parsing:

The input parsing now works correctly, taking the number of pairs and each pair of indices.

This should solve the problem efficiently. Let me know if you encounter any issues with it!