# My Introduction:

Hi, My name is Huy. I’m a Senior software engineer at HCLTech. I work in ANZ plus project, which is a new generation of ANZ bank. In my three years in the IT industry, I’ve improved my critical thinking, collaboration skills, and I love working with a team. I also had the opportunity to serve as a service owner and champions of two features.

# 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 vs Final

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();    } }** |
| --- |

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.

# 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:

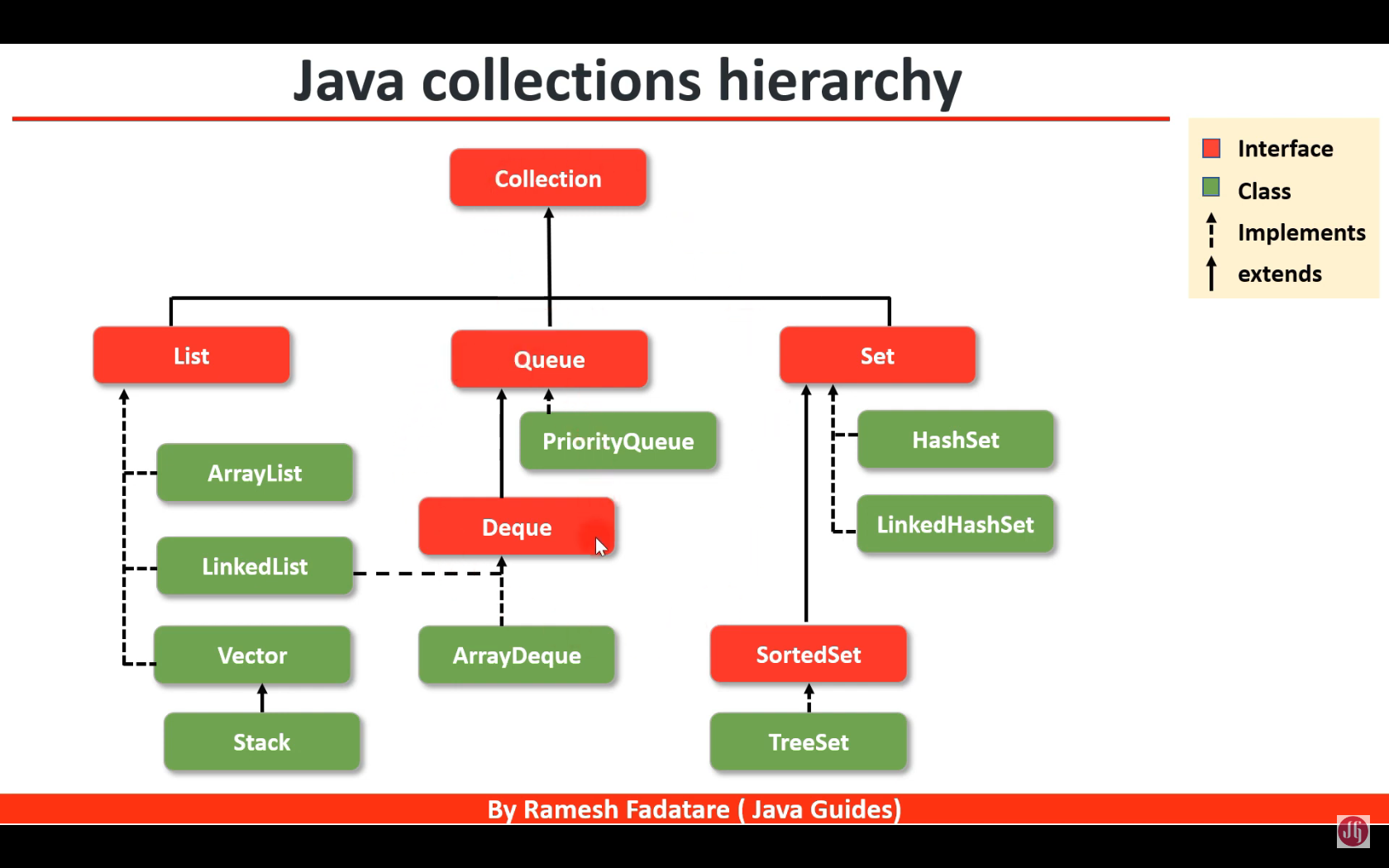
The Optional class in Java 8 is a container object which is used to contain a value that might or might not be present. It was introduced as a way to help reduce the number of NullPointerExceptions that occur in Java code. It is a part of java. util package and was added to Java as part of Java 8

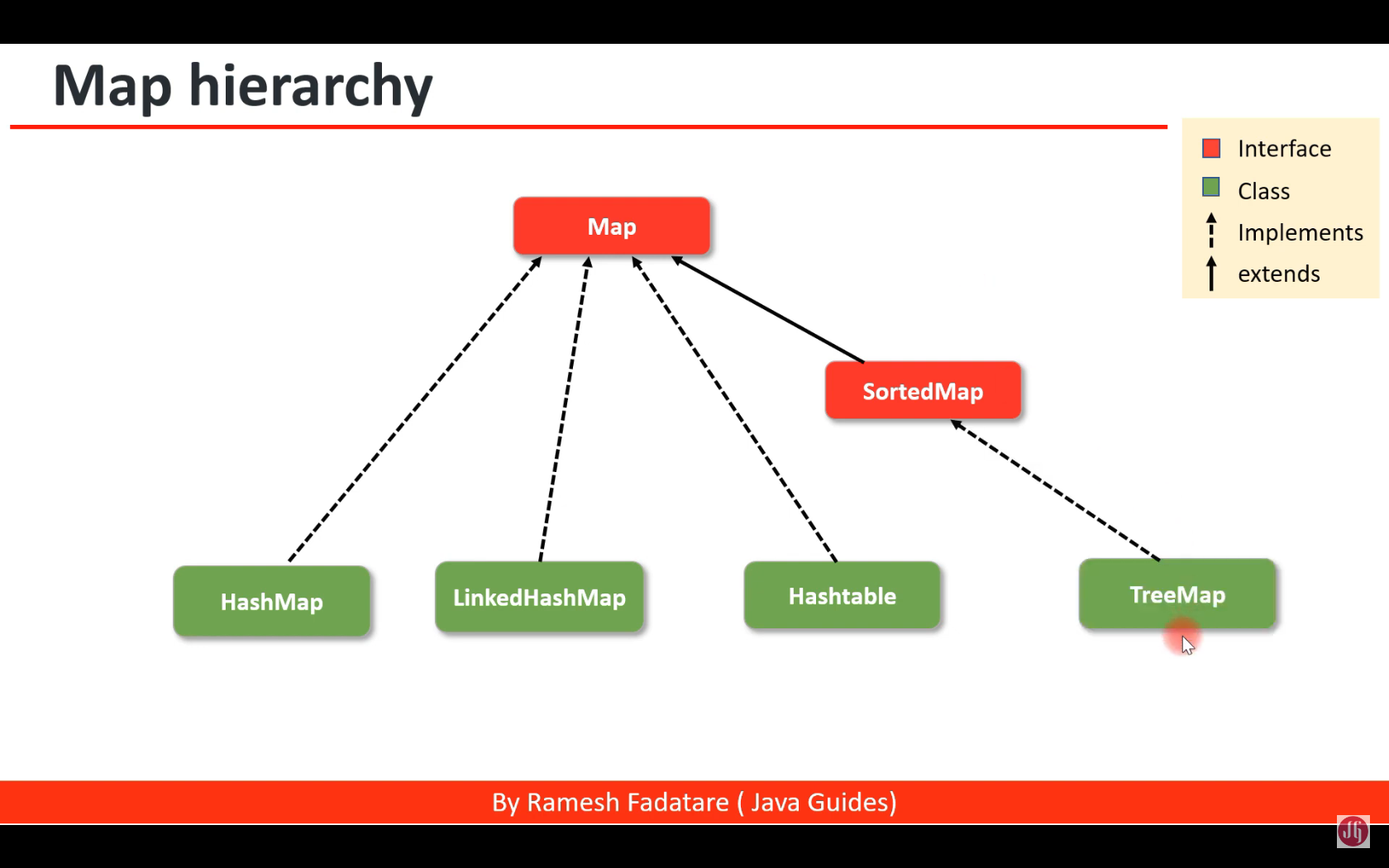
| class Customer {  private Long id;  private String firstName;  private String lastName;   public Long getId() {  return id;  }   public String getFirstName() {  return firstName;  }   public String getLastName() {  return lastName;  }   public Customer(Long id, String firstName, String lastName) {  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:

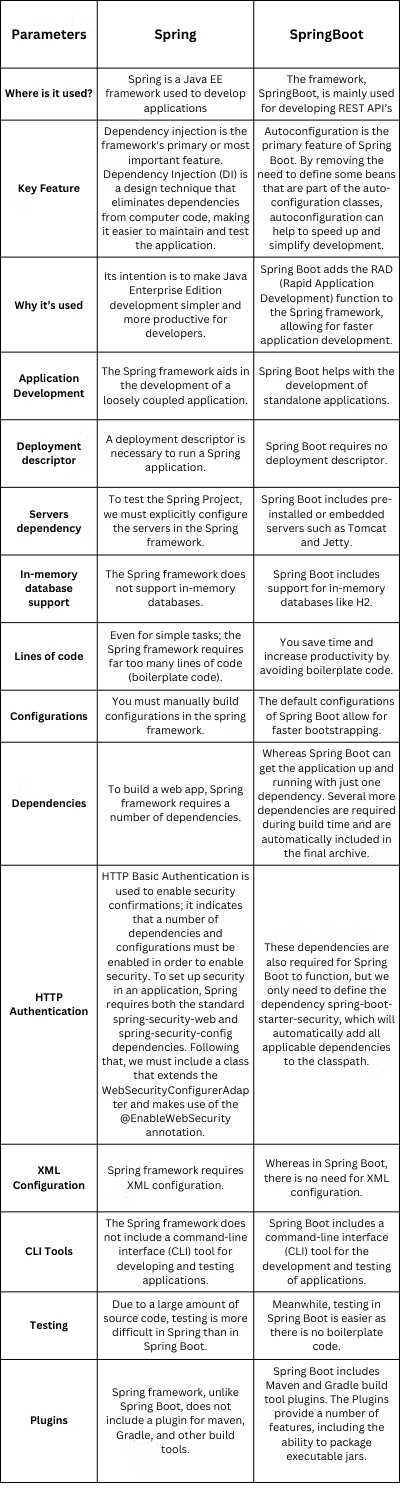
Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

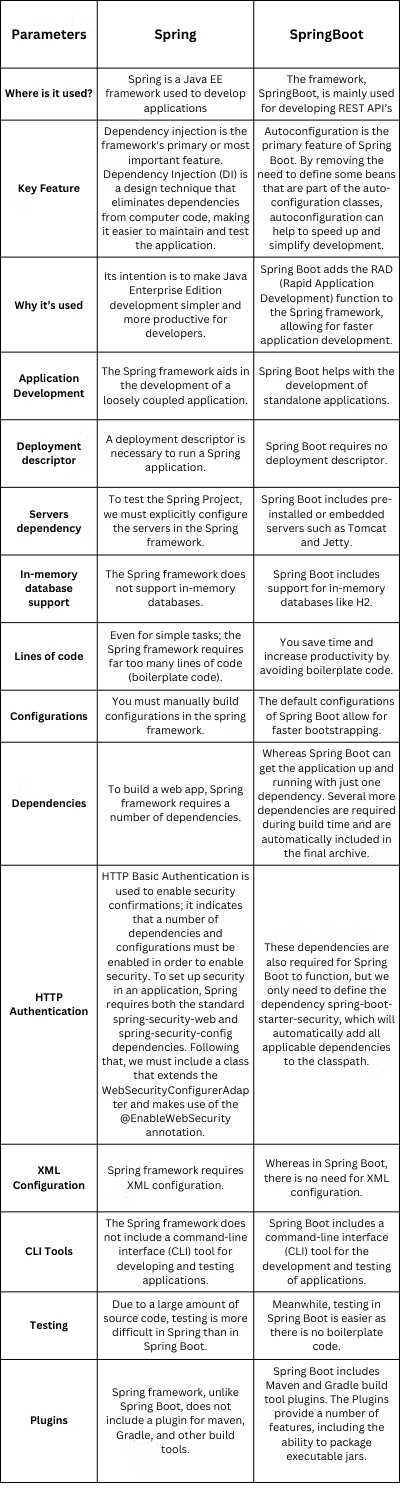
Examples of how you, as an application developer, can use the Spring platform advantage:

* 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:



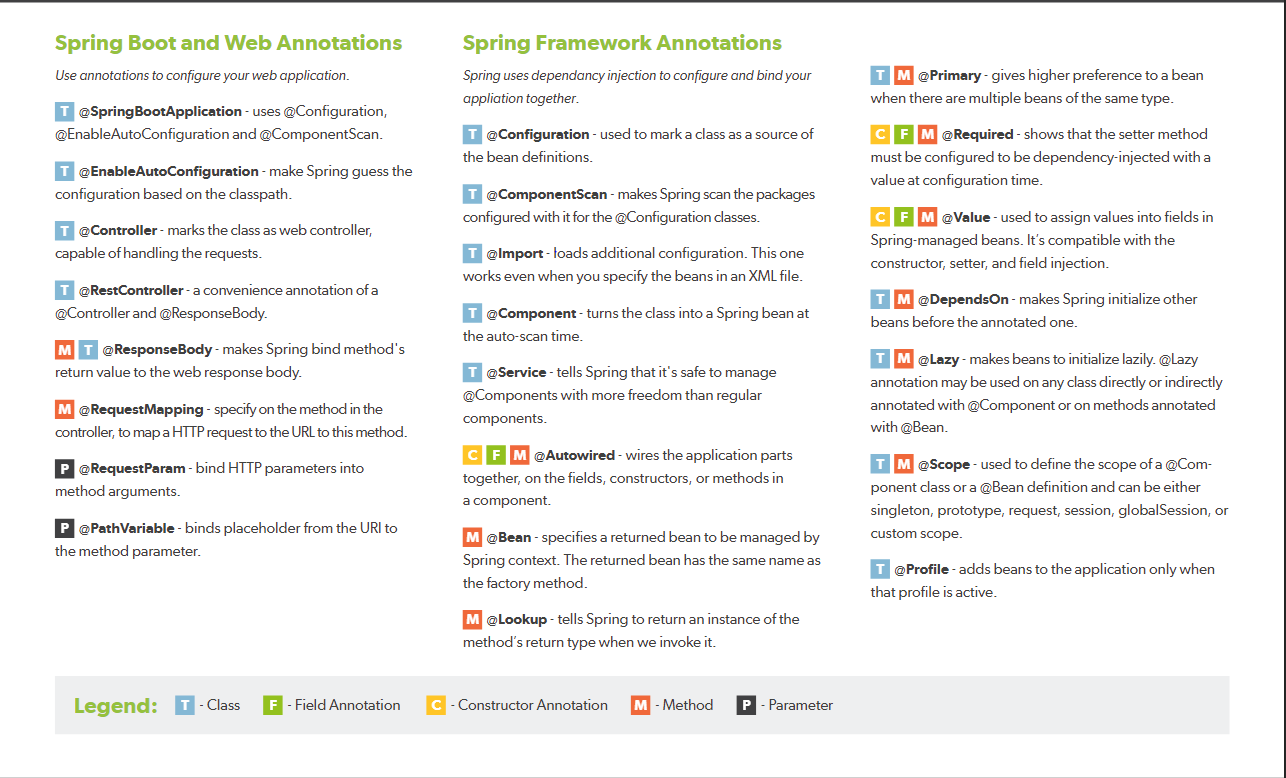


The three most basic parts of the Java ecosystem are the Java Virtual Machine (JVM), the Java Runtime Environment (JRE), and the Java Development Kit (JDK), which are *stock* parts that are supplied by Java implementations.

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.

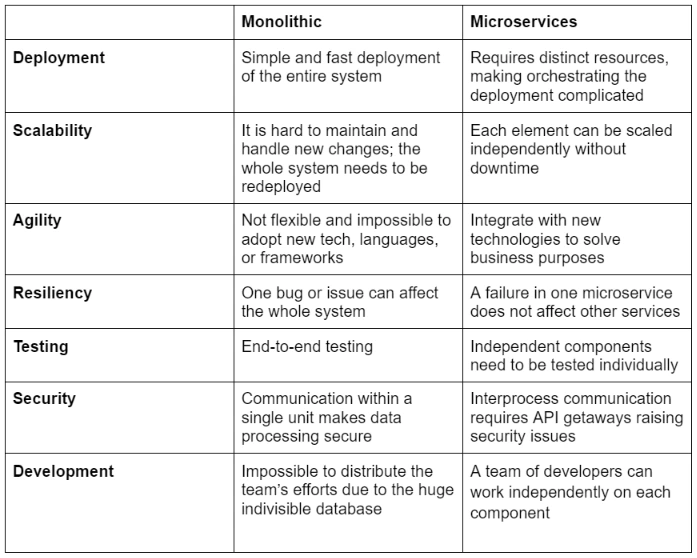
**Command:**

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.

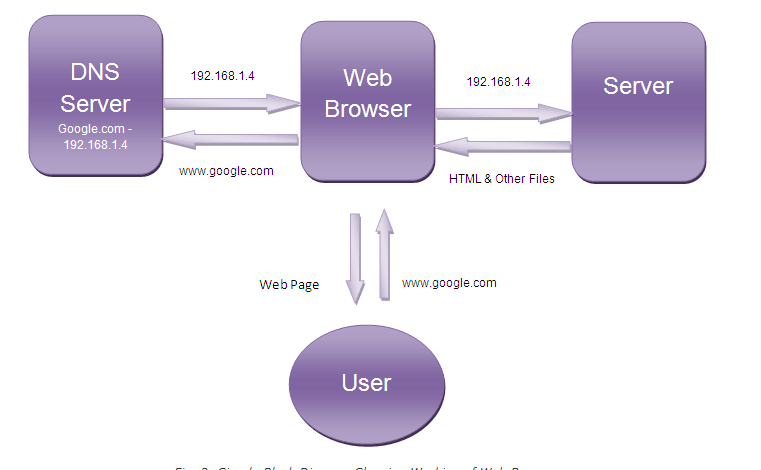
**Iterator:**

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:

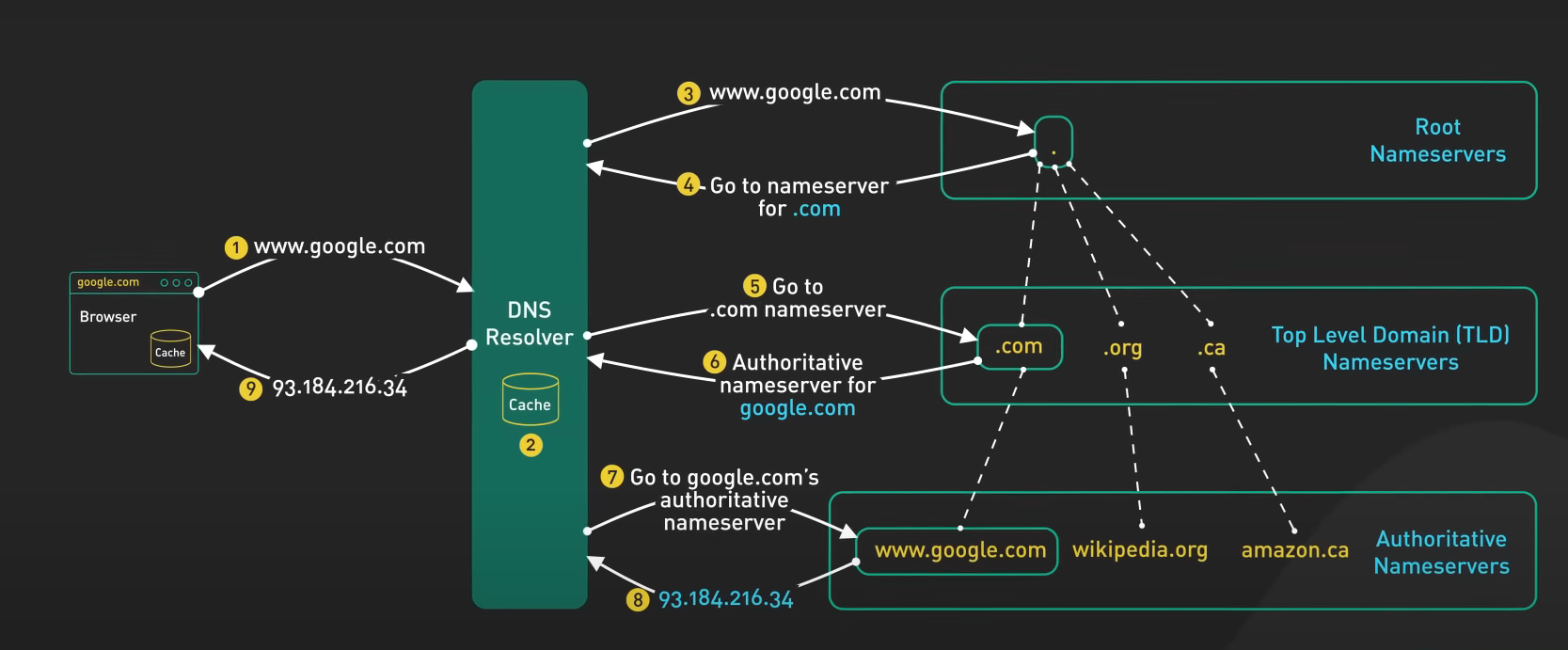


# Web Browser:



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

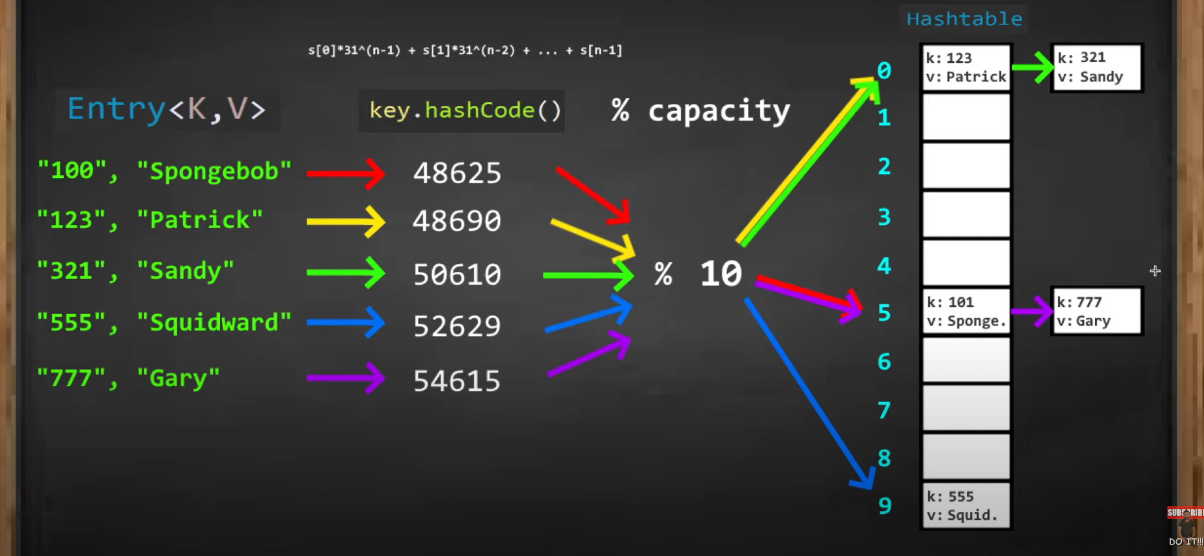
1. Check cache in browser. if not, call DNS resolver
2. Check cache in DNS resolver, if not, call Root name server (Root DNS)
3. Call Root name server
4. Return TLD .com
5. Call TLD (top level domain)
6. Return authoritative nameserver
7. Call Authoritative name server
8. Return IP address for DNS resolver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



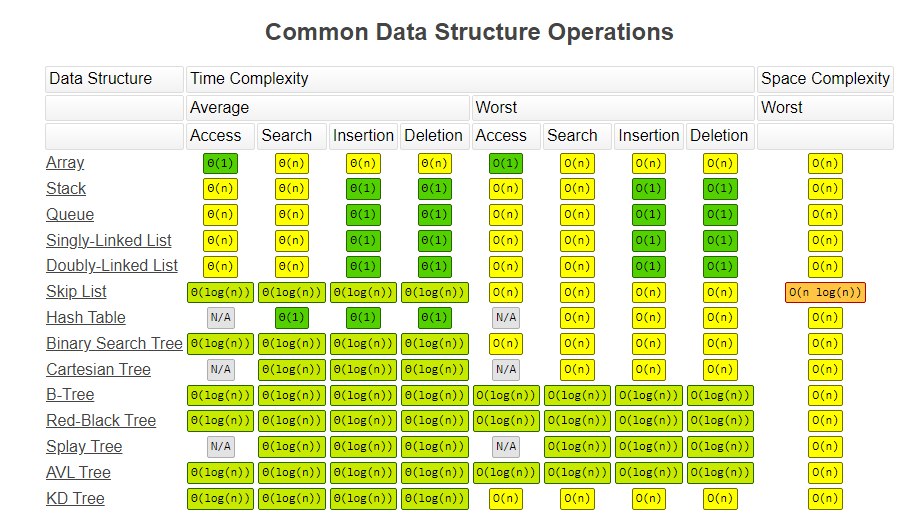
# Data Structure:

## Hashtable:

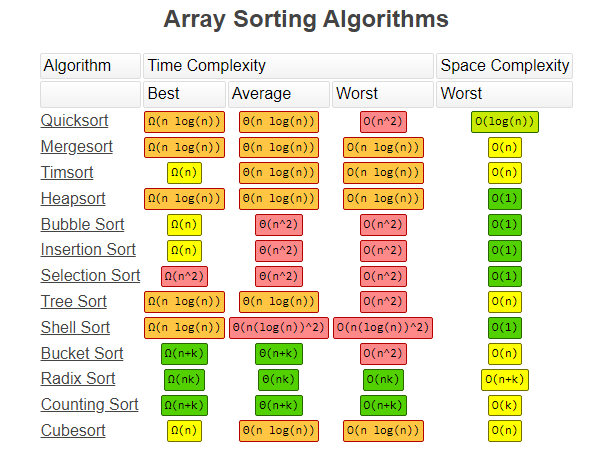
is an array, the key will be hash by modulus the capacity. But got problem with Collision, to solve that we need will store the linked list



## 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.

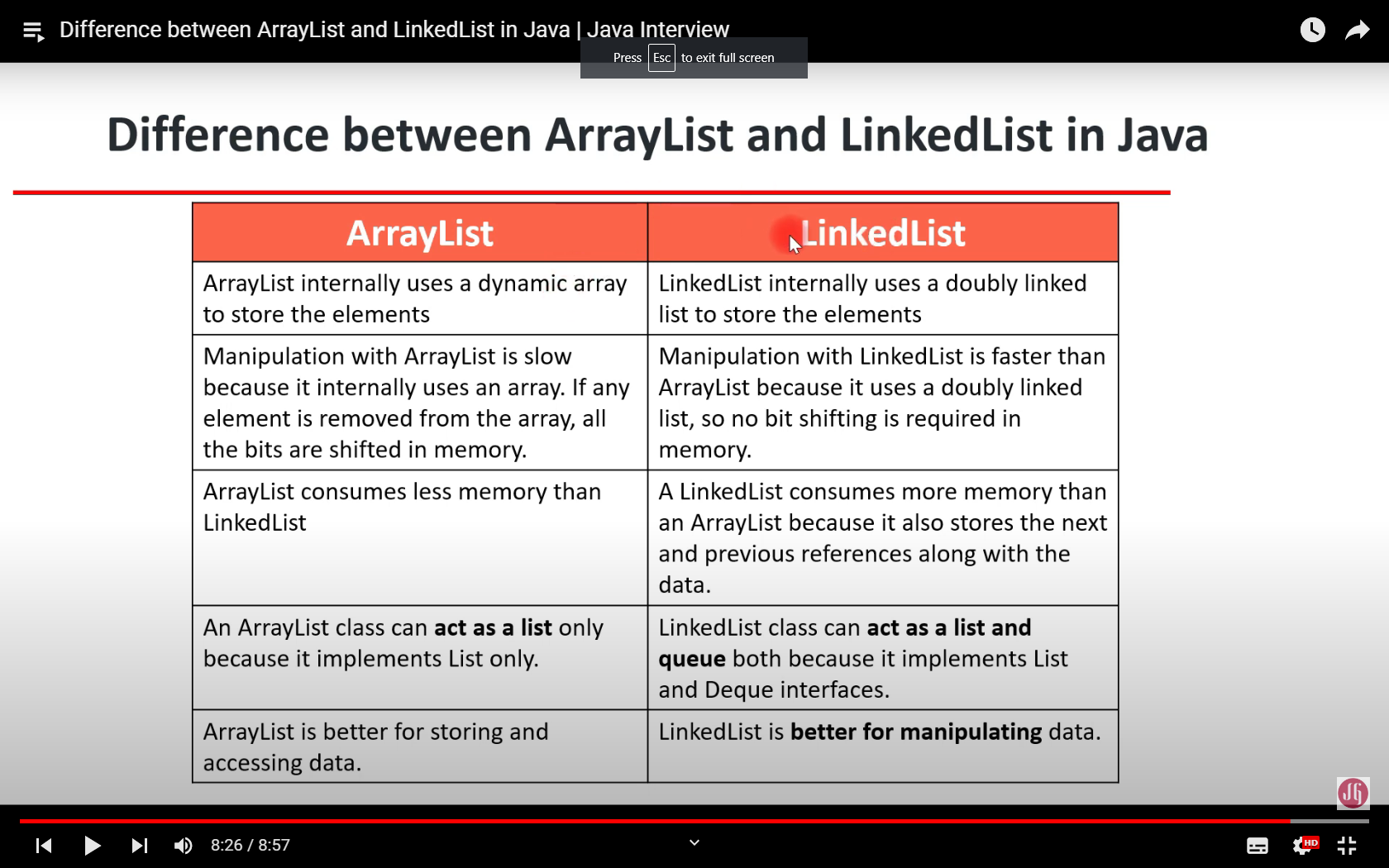
Improving backend performance is an ongoing process, and adopting best practices like regular monitoring, refactoring, and scaling will ensure your project runs efficiently.

# 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:



## 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. |

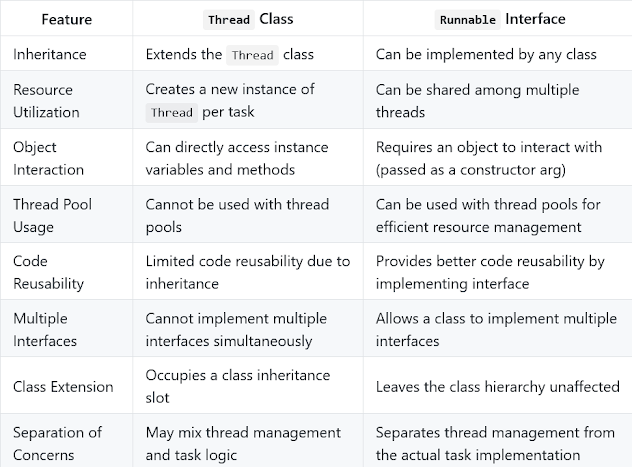
## Final keyword:

* 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.  
sql  
Copy code  
SELECT \* FROM authors;

For each author, you execute another query to get their books:  
sql  
Copy code  
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.

Solutions by ORM:

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

In Django ORM: Use .select\_related() or .prefetch\_related().  
python  
Copy code  
# Without eager loading (N+1 problem)

authors = Author.objects.all()

for author in authors:

books = author.books.all() # N+1 queries

# With eager loading (fixed)

authors = Author.objects.prefetch\_related('books') # 2 queries: one for authors, one for books

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

SQL Example:  
sql  
Copy code  
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.

**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:

# Git Flow vs Trunk-Based

## 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.