# My Introduction:

Hi, My name is Huy. I’m an associate engineer at NAB (Own My Home) who constantly seeks out the solution for everyday problems. In my three years in this 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 in my squad.

# 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:

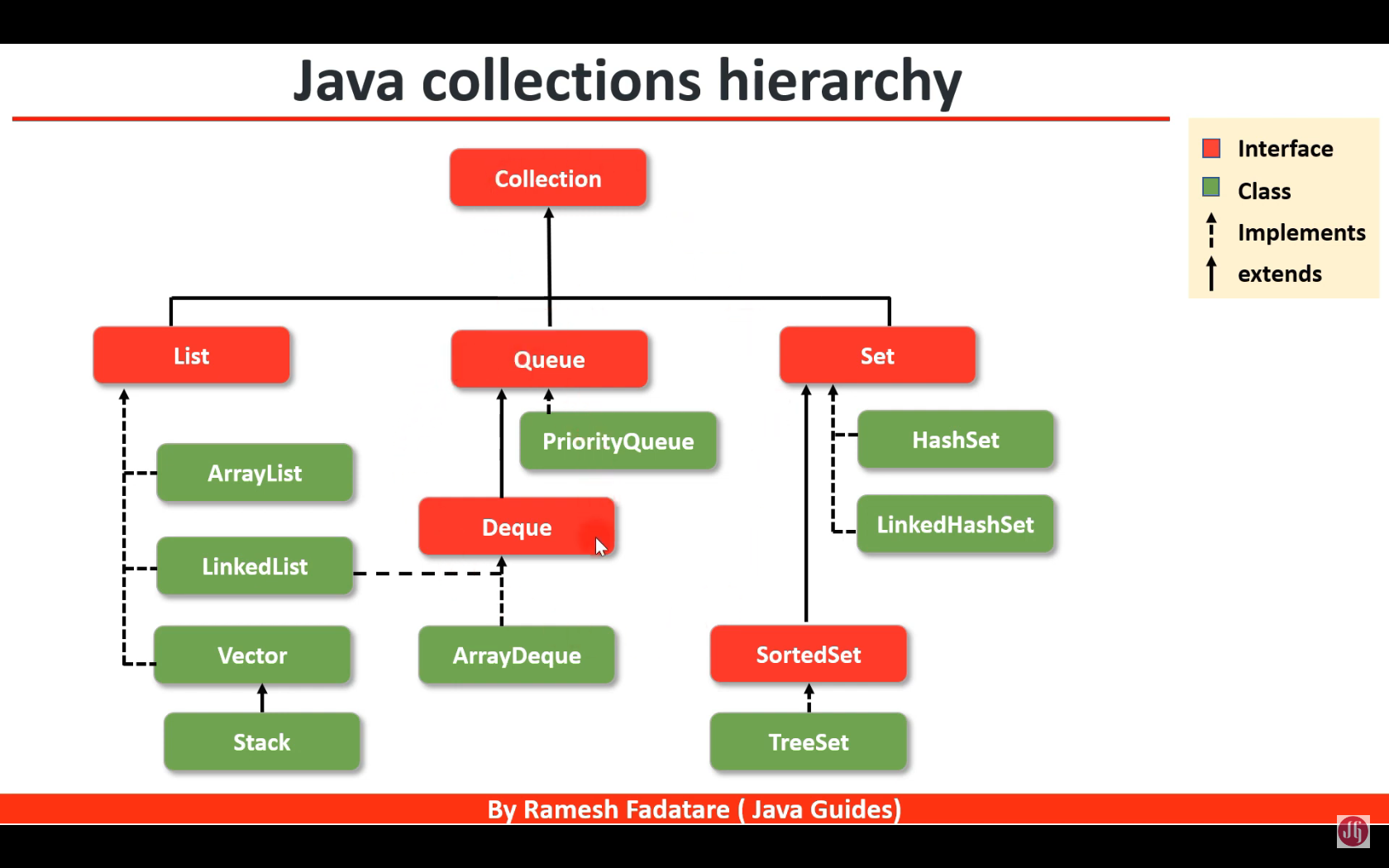
The Optional class in Java 8 is a container object which is used to contain a value that might or might not be present. It was introduced as a way to help reduce the number of NullPointerExceptions that occur in Java code. It is a part of java. util package and was added to Java as part of Java 8

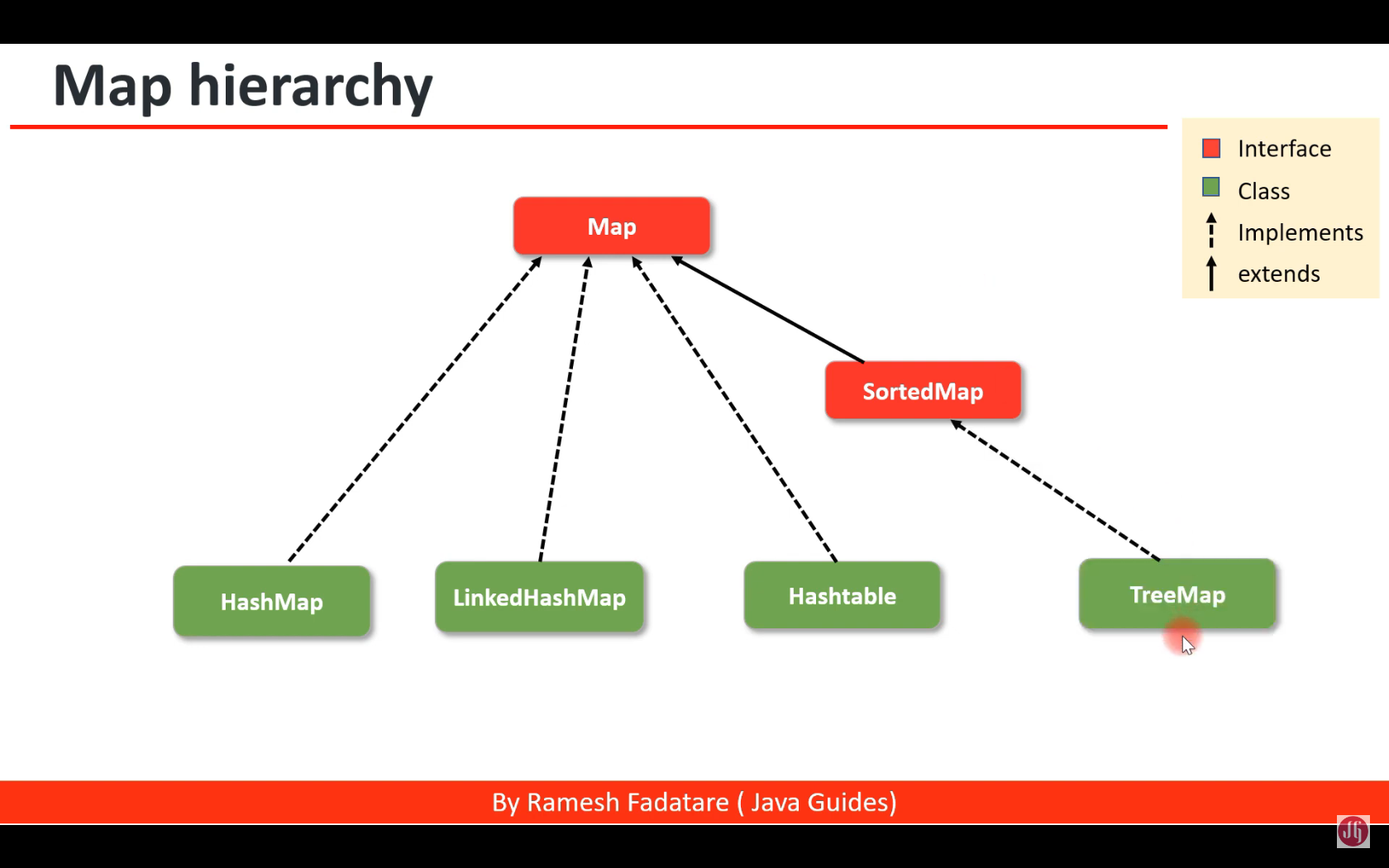
| class Customer {  private Long id;  private String firstName;  private String lastName;   public Long getId() {  return id;  }   public String getFirstName() {  return firstName;  }   public String getLastName() {  return lastName;  }   public Customer(Long id, String firstName, String lastName) {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

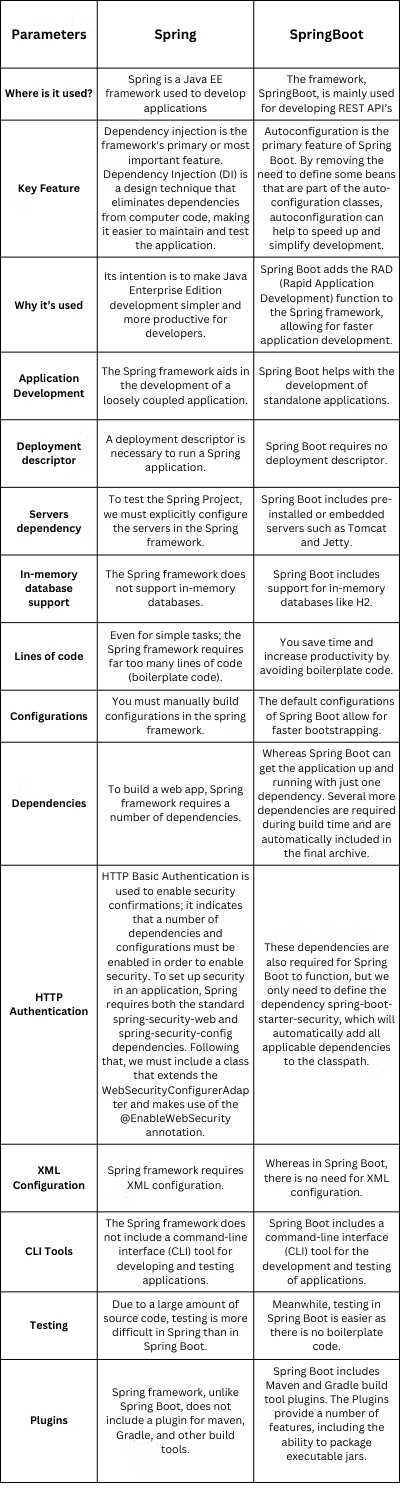
Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

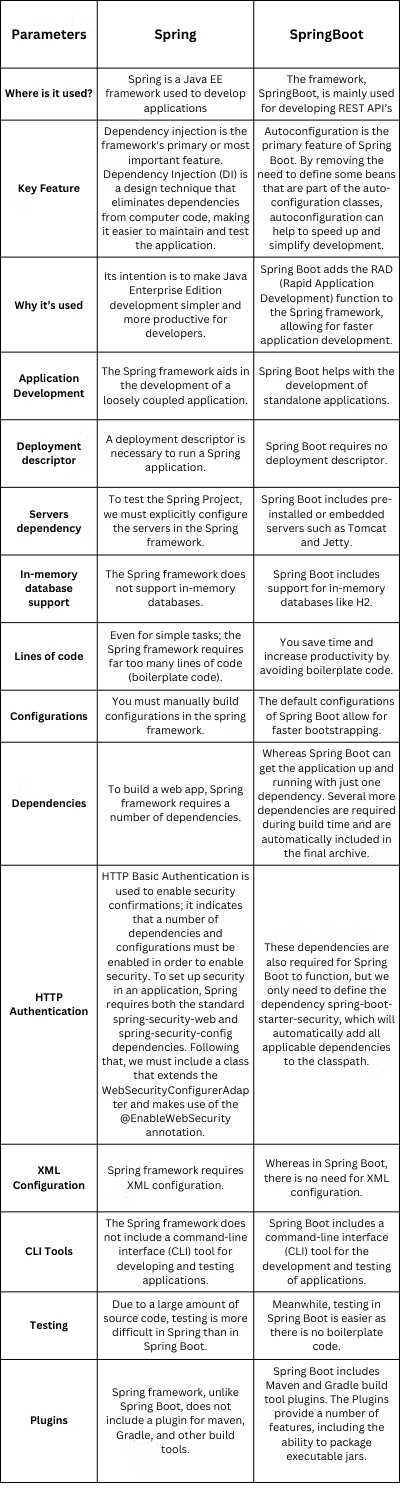
Examples of how you, as an application developer, can use the Spring platform advantage:

* 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:



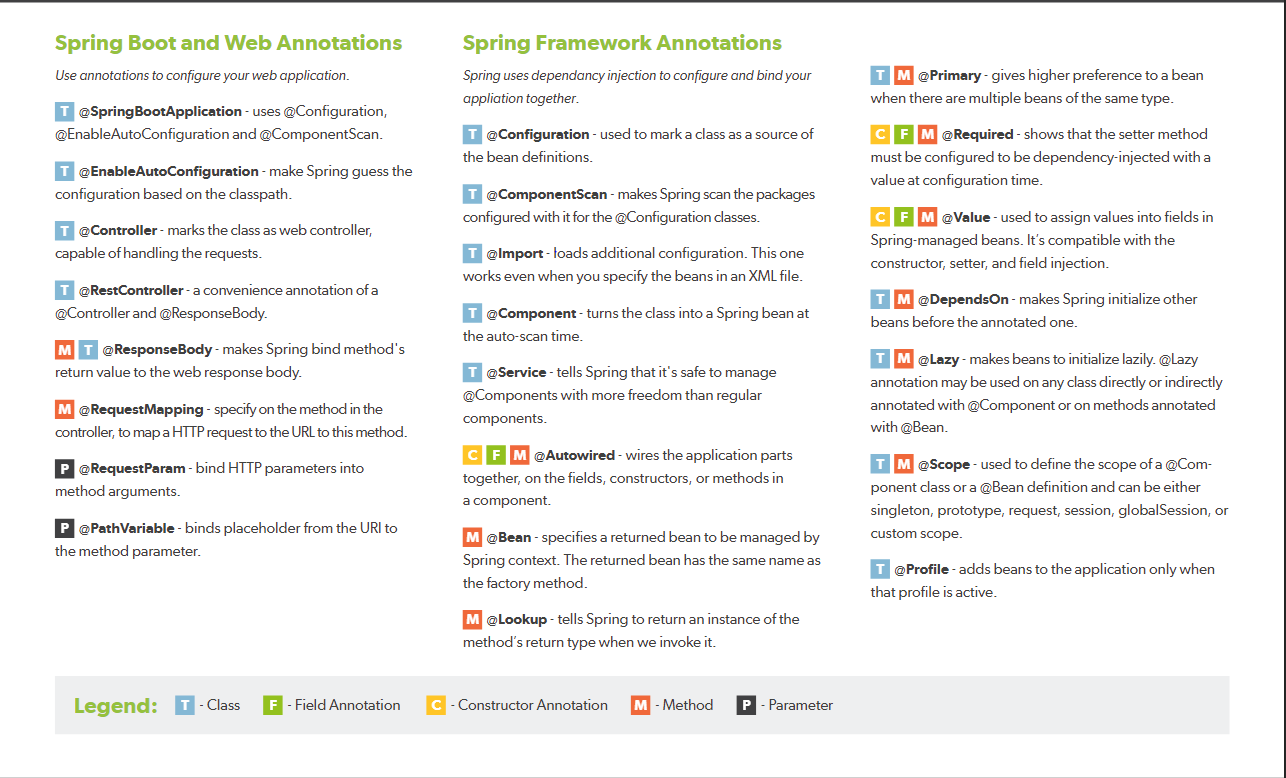


The three most basic parts of the Java ecosystem are the Java Virtual Machine (JVM), the Java Runtime Environment (JRE), and the Java Development Kit (JDK), which are *stock* parts that are supplied by Java implementations.

Moreover, IDE and libraries are also Java ecosystem

# Annotations:

Annotations in [Java](https://www.simplilearn.com/tutorials/java-tutorial/what-is-java) provide additional information to the compiler and [JVM](https://www.simplilearn.com/jvm-and-the-implications-of-sandbox-model-rar41-article). An annotation is a tag representing metadata about [classes](https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects), [interfaces](https://www.simplilearn.com/tutorials/java-tutorial/java-interface), variables, [methods](https://www.simplilearn.com/tutorials/java-tutorial/methods-in-java), or fields. Annotations do not impact the execution of the code that they annotate.



**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 |

# 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.

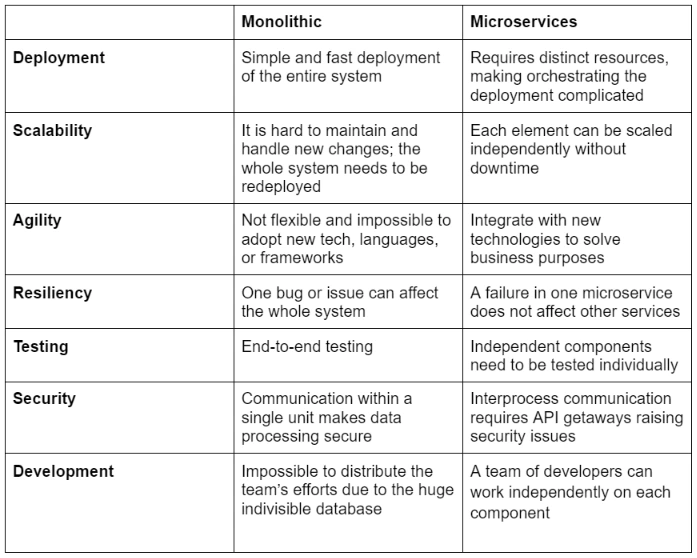
**Command:**

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.

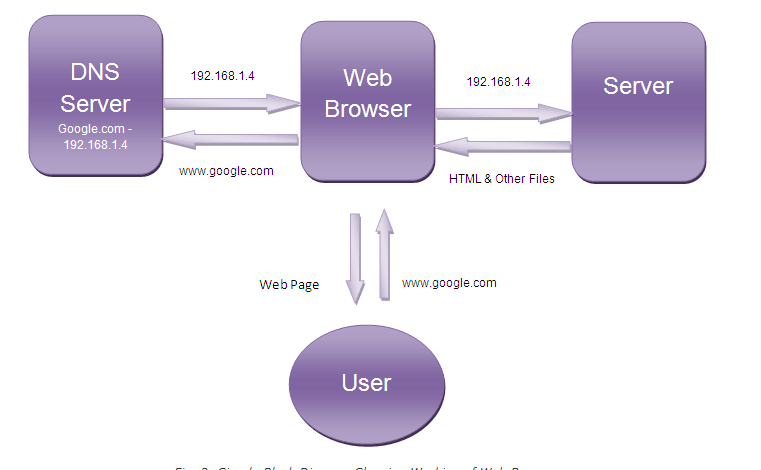
**Iterator:**

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:

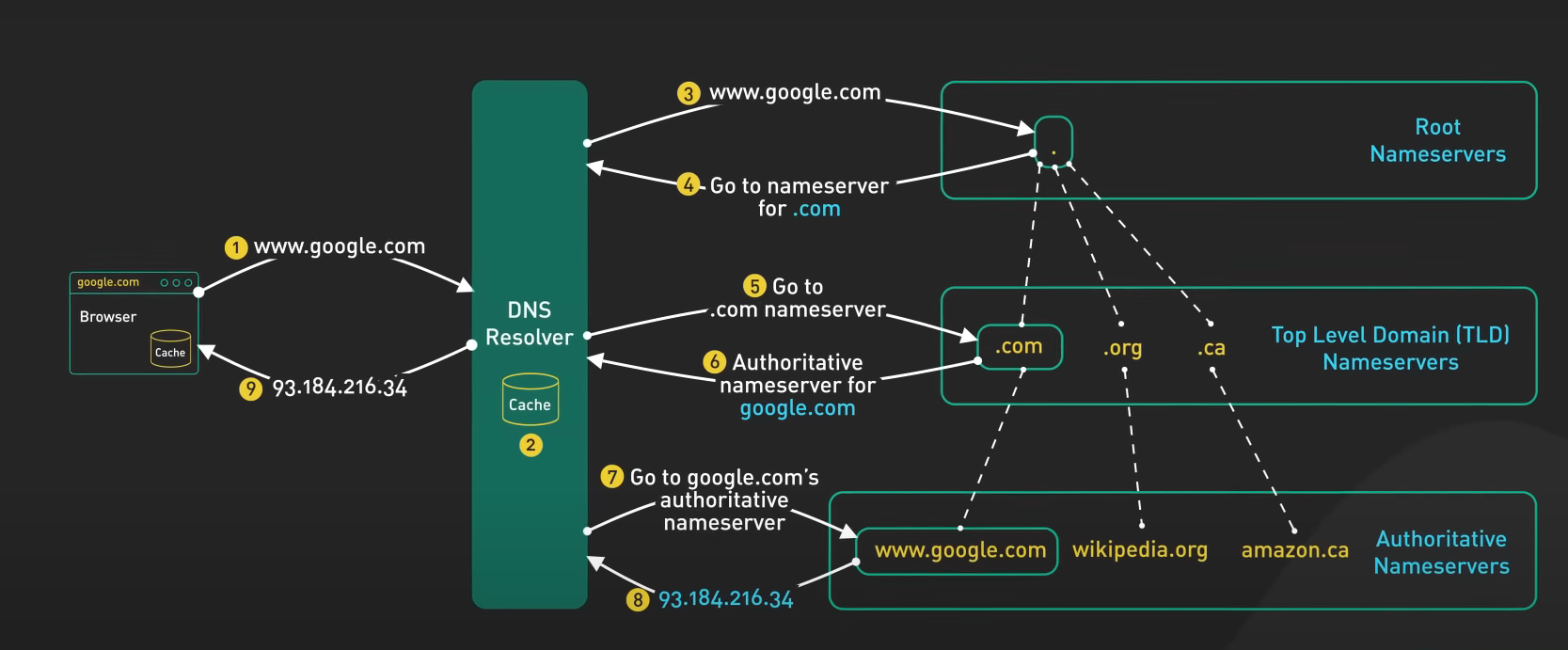


# Web Browser:



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

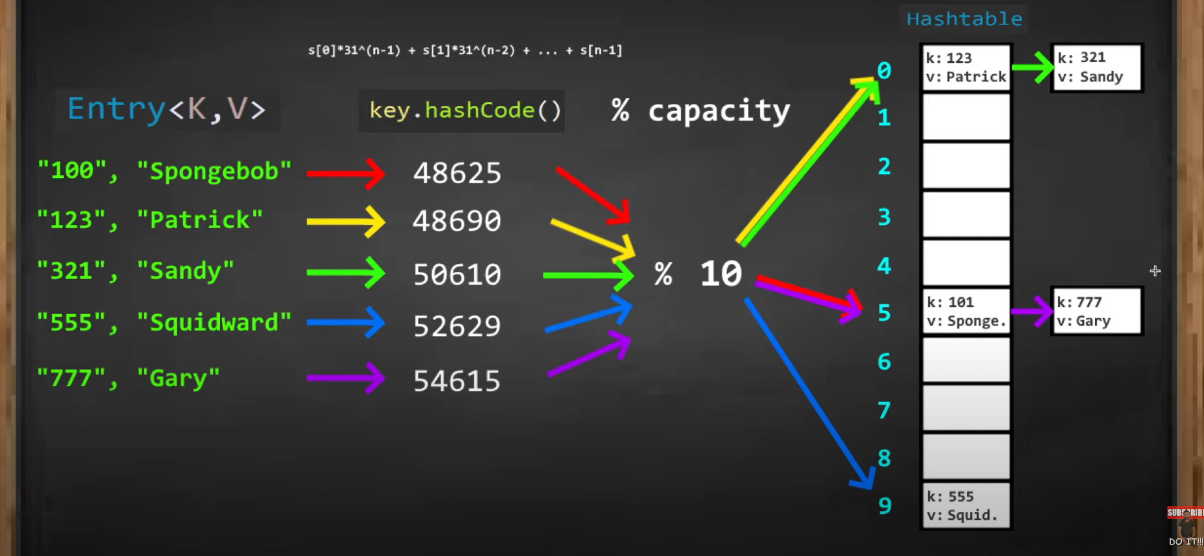
1. Check cache in browser. if not, call DNS resolver
2. Check cache in DNS resolver, if not, call Root name server (Root DNS)
3. Call Root name server
4. Return TLD .com
5. Call TLD (top level domain)
6. Return authoritative nameserver
7. Call Authoritative name server
8. Return IP address for DNS resolver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



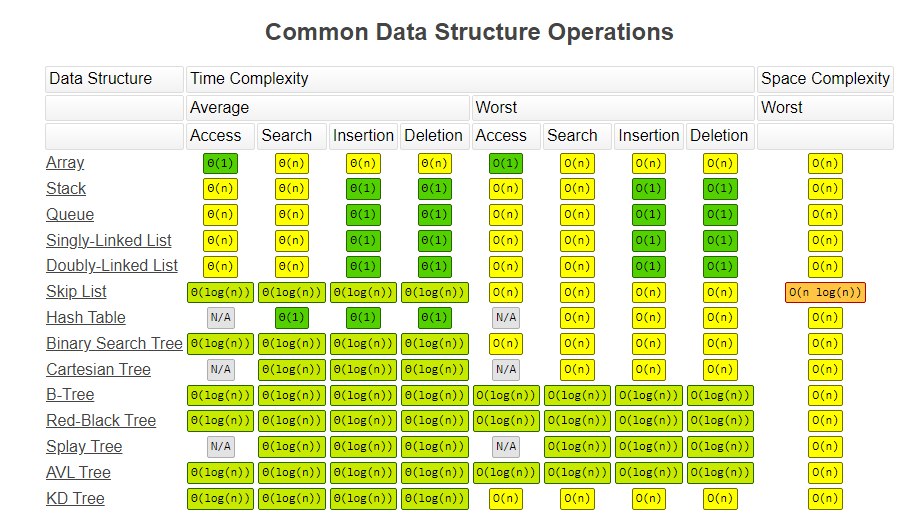
# Data Structure:

## Hashtable:

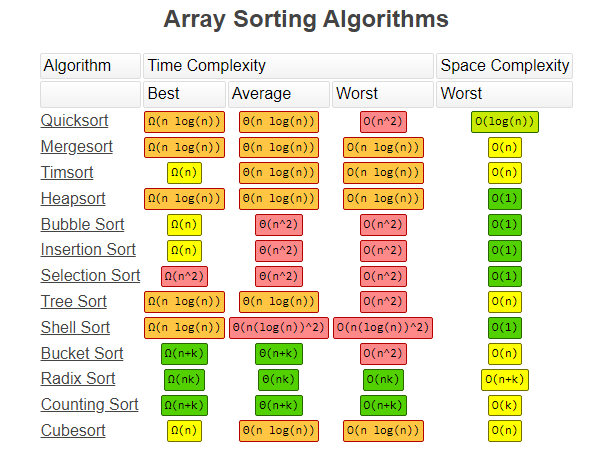
is an array, the key will be hash by modulus the capacity. But got problem with Collision, to solve that we need will store the linked list



## 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.

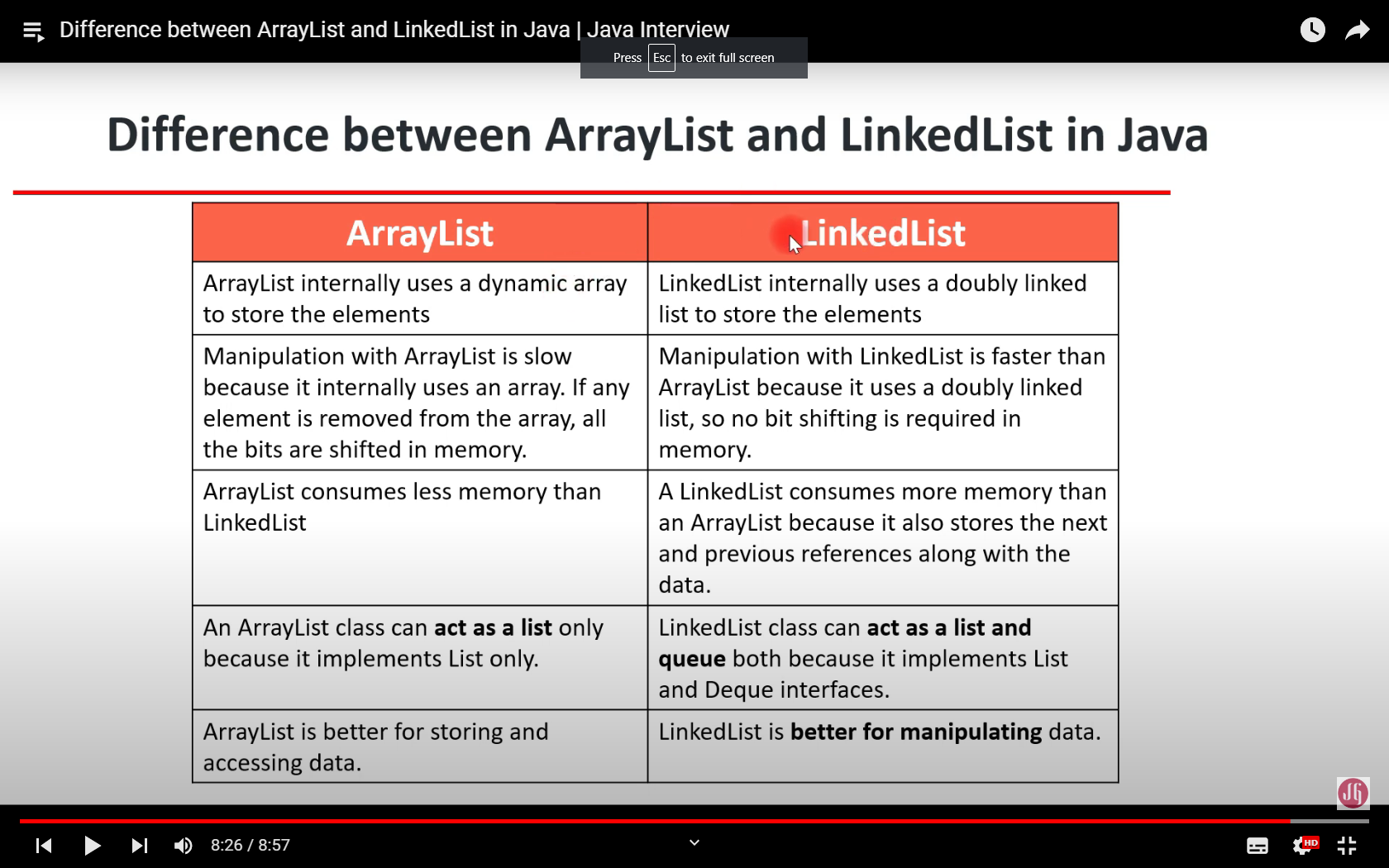
Each consumer need to have consumer group Id to know the current offset

# 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

