**Qn. What is JDK, JRE, JVM and JIT?**

1. **JDK (Java Development Kit)**:
   * The JDK is a software development kit used for developing Java applications.
   * It includes tools for compiling, debugging, and running Java programs.
   * The JDK contains the JRE (Java Runtime Environment) and additional development tools such as the Java compiler (**javac**), debugger (**jdb**), and other utilities.
2. **JRE (Java Runtime Environment)**:
   * The JRE is a set of software tools used to run Java applications.
   * It includes the Java Virtual Machine (JVM), class libraries, and other supporting files required to execute Java bytecode.
   * The JRE does not contain development tools such as the compiler or debugger; it is used only for running Java applications.
3. **JVM (Java Virtual Machine)**:
   * The JVM is a virtual machine that provides an environment in which Java bytecode can be executed.
   * It interprets the bytecode and translates it into machine code that can be understood by the host operating system and processor.
   * The JVM is platform-independent, meaning that Java bytecode can run on any system with a compatible JVM installed.
4. **JIT (Just-In-Time Compiler)**:
   * The JIT compiler is a component of the JVM that improves the performance of Java applications.
   * It dynamically compiles Java bytecode into native machine code at runtime, allowing the application to execute more quickly.
   * The JIT compiler identifies frequently executed code paths and optimizes them for better performance.

Now, let's link these concepts together with an example:

Suppose you have a Java program (**HelloWorld.java**) that prints "Hello, world!" to the console:

*public class HelloWorld {*

*public static void main(String[] args) {*

*System.out.println("Hello, world!");*

*}*

*}*

Here's how these concepts are involved:

* **JDK**: You use the JDK to compile the **HelloWorld.java** source code into bytecode using the **javac** compiler tool.
* **JRE**: When you run the compiled bytecode (e.g., **HelloWorld.class**), you need the JRE installed on your system. The JRE provides the JVM and class libraries necessary to execute the bytecode.
* **JVM**: When you execute the **HelloWorld** class, the JVM loads the bytecode, interprets it, and executes the instructions on the host system. It manages memory, performs garbage collection, and ensures platform independence.
* **JIT**: When a Java program is executed, the JVM loads the bytecode instructions from the .class files.
  + Instead of interpreting the bytecode directly, the JVM passes it through the JIT compiler.
  + The JIT compiler analyses the bytecode and identifies frequently executed portions of code (hot spots).
  + For these hot spots, the JIT compiler generates***optimized native machine code*** specific to the underlying hardware architecture and operating system.
  + The native machine code is then executed directly by the CPU, which results in faster execution compared to interpreting bytecode.

**Qn. Is Java call by value or call by reference?**

Java is often said to be "call by value" because when you pass arguments to a method in Java, you're passing the value of the variable, not the variable itself. However, this *confused* especially for objects.

Let's clarify:

1. **Primitive types**: When you pass a primitive type (e.g., **int**, **char**, **boolean**) to a method in Java, you're passing a copy of the actual value stored in the variable. Any changes made to the parameter within the method are made to this copy, not to the original variable.

public void modifyPrimitive(int x) {  
 x = 10; // Changes made to the copy of 'x'  
 }  
  
public static void main(String[] args) {  
 int num = 5;  
 modifyPrimitive(num);  
 System.out.println(num); // Output: 5 (unchanged)  
 }

**Objects**: When you pass an object reference to a method in Java, you're passing a copy of the reference, not the actual object. This means that changes made to the object's state within the method affect the original object.

class MyClass {  
 int value;  
 MyClass(int value) {  
 this.value = value;  
 }  
}  
 public void modifyObject(MyClass obj) {  
 obj.value = 10; // Changes made to the original object  
 }  
  
 public static void main(String[] args) {  
 MyClass obj = new MyClass(5);  
 modifyObject(obj);  
 System.*out*.println(obj.value); // Output: 10 (changed)  
 }  
}

In summary, while Java is referred to as "call by value" it's essential to understand that this terminology applies differently to primitive types and objects. For primitive types, you're passing a copy of the value, while for objects, you're passing a copy of the reference to the object. In both cases, changes made within the method affect the copied value or reference, not the original variable or object.

**Qn. Type of Relationships in Java?**

**Is-A Relationship**:

The is-a relationship represents inheritance or specialization, where one class is a type of another class.

**class Animal { ... }**

**class Dog extends Animal { ... }**

Here, **Dog** is-a **Animal**, indicating that **Dog** inherits from **Animal**.

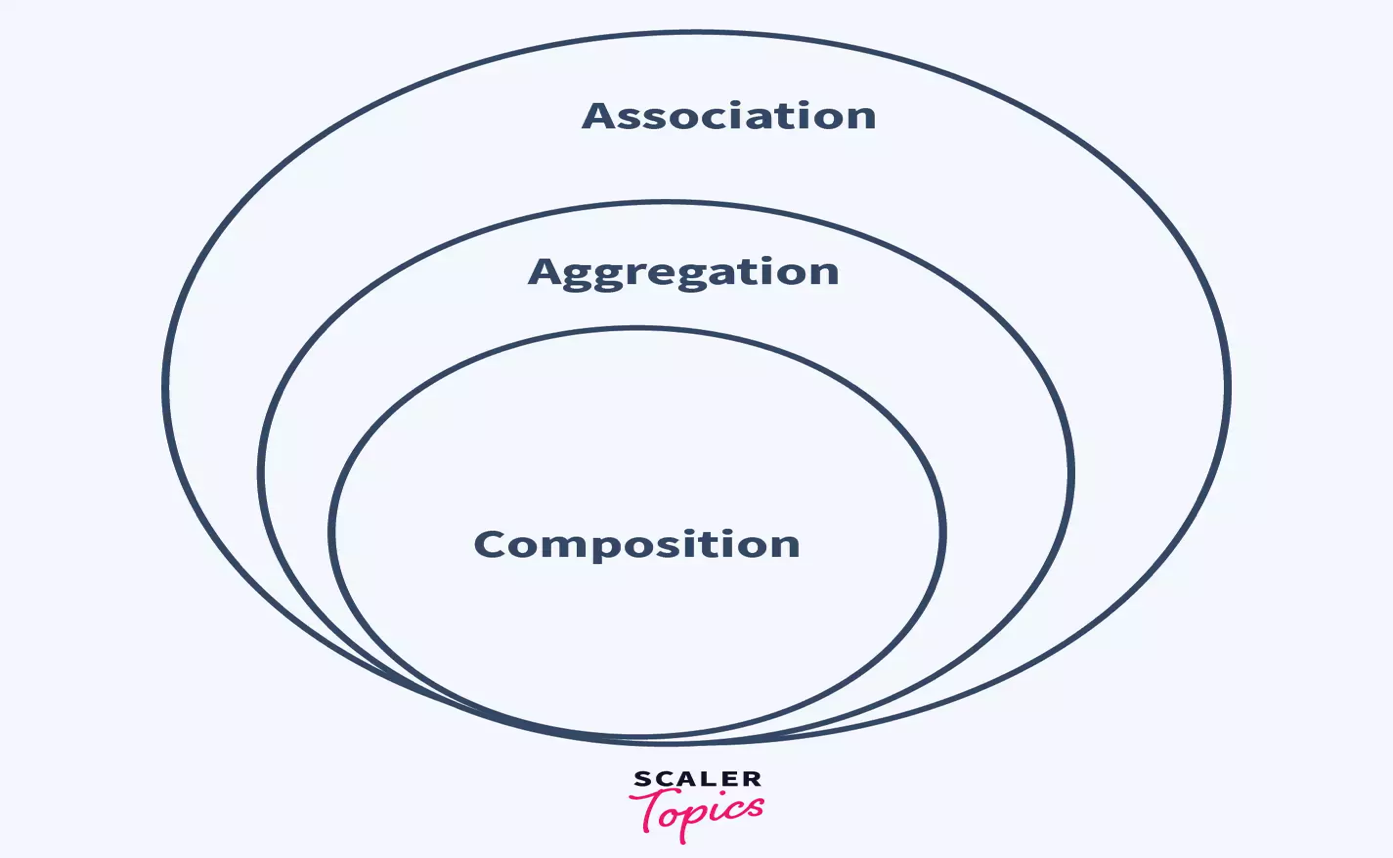
**Association: [Aggregation, Composition]**

Association in java is [a relationship between two classes where one class is aware of the other class].

It has two forms, Aggregation(HAS-A) and Composition(Belongs-to).

Aggregation is a relatively weak association, whereas Composition is a strong association. Composition can be called a more restricted form of Aggregation.

Aggregation can be called the superset of Composition, since all Compositions can be Aggregations but, not all Aggregations can be called Composition.



Aggregation Example:

import java.util.ArrayList;  
import java.util.List;  
  
public class Library {  
 private List<Book> books;  
   
 public Library() {  
 this.books = new ArrayList<>();  
 }  
  
 public void addBook(Book book) {  
 this.books.add(book);  
 }  
}  
  
public class Book {  
 private String title;  
  
 public Book(String title) {  
 this.title = title;  
 }  
}

Here Book and Library can grow independently.

Composition Example:

public class Car {  
 private Engine engine;  
  
 public Car() {  
 this.engine = new Engine();  
 }  
  
 // Other methods...  
}  
  
public class Engine {  
 // Engine properties and methods...  
}

Here changes done in Engine are directly affecting Car.

Below are the primary differences between the forms of Association, Composition and Aggregation in java:

| **Aggregation** | **Composition** |
| --- | --- |
| Weak Association | Strong Association |
| Classes in relation can exist independently | One class is dependent on Another Independent class. The Dependent class cannot exist independently in the event of the non-existence of an independent class. |
| One class has-a relationship with another class | Once class belongs-to another class |
| Helps with code reusability. Since classes exist independently, associations can be reassigned or new associations created without any modifications to the existing class. | Code is not that reusable as the association is dependent. Such Associations once established will create a dependency, and these associations cannot be reassigned or new associations like aggregation, etc cannot be created without changing the existing class. |

**Qn. OOPS in java.**

**Inheritance:**

Inheritance allows a class (subclass/derived class) to inherit properties and behavior from another class (superclass/base class). It promotes code reusability and establishes an "is-a" relationship between classes.

// Superclass  
class Vehicle {  
 void move() {  
 System.*out*.println("Vehicle is moving...");  
 }  
}  
  
// Subclasses inheriting from Vehicle  
class Car extends Vehicle {  
 void honk() {  
 System.*out*.println("Car is honking...");  
 }  
}  
  
class Bike extends Vehicle {  
 void ringBell() {  
 System.*out*.println("Bike is ringing bell...");  
 }  
}  
  
// Using inheritance and polymorphism  
public class Main {  
 public static void main(String[] args) {  
 Vehicle myCar = new Car();  
 Vehicle myBike = new Bike();  
  
 myCar.move(); // Calls Vehicle's move method  
 myBike.move(); // Calls Vehicle's move method  
  
 // Specific methods of subclasses  
 ((Car) myCar).honk();  
 ((Bike) myBike).ringBell();  
 }  
}

**Polymorphism:**

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables methods to be defined in a superclass and overridden by subclasses.

// Superclass  
class Shape {  
 void draw() {  
 System.*out*.println("Drawing a shape");  
 }  
}  
  
// Subclasses overriding the draw method  
class Circle extends Shape {  
 @Override  
 void draw() {  
 System.*out*.println("Drawing a circle");  
 }  
}  
  
class Rectangle extends Shape {  
 @Override  
 void draw() {  
 System.*out*.println("Drawing a rectangle");  
 }  
}  
   
// Using polymorphism  
public class Main {  
 public static void main(String[] args) {  
 Shape circle = new Circle();  
 Shape rectangle = new Rectangle();  
  
 circle.draw(); // Calls Circle's draw method  
 rectangle.draw(); // Calls Rectangle's draw method  
 }  
}

**Encapsulation:**

Encapsulation is the bindling of data and methods into a single unit (class). It hides the internal state of an object and only exposes necessary functionality through methods.

class Student {  
 private String name;  
 private int age;  
  
 // Getter and setter methods  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}  
  
// Using encapsulation  
public class Main {  
 public static void main(String[] args) {  
 Student student = new Student();  
 student.setName("John");  
 student.setAge(20);  
  
 System.*out*.println("Name: " + student.getName());  
 System.*out*.println("Age: " + student.getAge());  
 }  
}

**Abstraction:**

Abstraction is the concept of hiding the implementation details and showing only the essential features of an object. It focuses on "what" an object does rather than "how" it does it.

abstract class Shape {  
 abstract void draw();  
}  
  
class Circle extends Shape {  
 @Override  
 void draw() {  
 System.*out*.println("Drawing a circle");  
 }  
}  
  
class Rectangle extends Shape {  
 @Override  
 void draw() {  
 System.*out*.println("Drawing a rectangle");  
 }  
}  
  
// Using abstraction  
public class Main {  
 public static void main(String[] args) {  
 Shape circle = new Circle();  
 Shape rectangle = new Rectangle();  
  
 circle.draw(); // Calls Circle's draw method  
 rectangle.draw(); // Calls Rectangle's draw method  
 }  
}

**Qn. What is Garbage Collection? How does it work in Java?**

Garbage Collection (GC) is the process of automatically reclaiming memory occupied by objects that are no longer in use or reachable by the program. In Java, GC is an integral part of memory management and helps prevent memory leaks and efficiently manages memory resources.

### How Garbage Collection Works in Java:

1. **Identification of Unreachable Objects:**
   * The Garbage Collector starts by identifying objects in memory that are no longer reachable from any live thread. An object is considered unreachable if there are no references to it from the root set, which includes global variables, local variables, and active threads.
2. **Marking Phase:**
   * Once unreachable objects are identified, the Garbage Collector performs a marking phase. During this phase, it traverses the object graph starting from the root set and marks all reachable objects as live. Objects not marked during this traversal are considered garbage.
3. **Sweeping Phase:**
   * After marking live objects, the Garbage Collector proceeds to the sweeping phase. In this phase, it deallocates memory occupied by unreachable objects and adds it back to the heap's free space.

public class GarbageCollectionExample {  
 public static void main(String[] args) {  
 // Create objects  
 Object obj1 = new Object();  
 Object obj2 = new Object();  
  
 // Assign obj1 to null, making it eligible for garbage collection  
 obj1 = null;  
  
 // Request garbage collection (not guaranteed to be executed immediately)  
 System.*gc*(); // This is just a hint to the JVM, not a command  
  
 // Continue program execution  
 // ...  
 }  
}

### Explanation:

* In this example, two **Object** instances are created and assigned to **obj1** and **obj2**.
* After setting **obj1** to **null**, it no longer has any references pointing to it, making it eligible for garbage collection.
* The **System.gc()** method is used to request garbage collection, but it's only a hint to the JVM, and the GC may not be executed immediately.
* Garbage collection occurs automatically in the background as needed by the JVM.

### Benefits of Garbage Collection in Java:

* Automatically manages memory, reducing the risk of memory leaks.
* Simplifies memory management for developers by eliminating the need for manual memory deallocation.
* Enhances application reliability and stability by preventing memory-related errors.

In general, it's not recommended to use **System.gc()**

Instead of explicitly invoking **System.gc()**, it's better to rely on the JVM's automatic garbage collection mechanism.

**Qn. changes done in garbage collector in java8?**

In Java 8, several changes were made to the garbage collector, particularly with the introduction of the **G1 (Garbage-First)** garbage collector as the default garbage collector. Here are some key changes and features introduced in Java 8 related to garbage collection:

1. **Garbage-First (G1) Garbage Collector**: Java 8 introduced the G1 garbage collector as the default garbage collector. G1 is designed to provide *more* *predictable garbage collection pauses* compared to other garbage collectors like the Parallel GC. It divides the heap into smaller regions and collects the regions with the most garbage first, hence the name "Garbage-First."

In traditional garbage collectors like the Parallel GC), garbage collection pauses can vary in duration and frequency, leading to unpredictable application performance. These pauses occur when the JVM halts the execution of application threads to perform garbage collection activities, such as identifying and reclaiming unreachable objects.

1. **Predictable Garbage Collection Pauses**: by limiting the length of the pauses. This is achieved by performing garbage collection incrementally on smaller regions of the heap.
2. **Concurrent Marking**: G1 performs most of the marking phase concurrently with the application threads, reducing the pause times associated with full garbage collection.
3. **Adaptive Sizing**: G1 dynamically adjusts the size of the heap and the regions based on the application's behavior and the garbage generation rate. This adaptive sizing helps improve the efficiency of garbage collection.
4. **Efficient Mixed Collections**: G1 performs mixed collections that handle both young and old generation garbage concurrently, reducing the need for separate collection cycles for young and old generations.
5. **Metaspace**: Java 8 introduced Metaspace to replace the Permanent Generation (PermGen) for class metadata storage. Metaspace is part of the native memory and is automatically resized as needed. This change helps prevent OutOfMemoryError issues related to class metadata.

Overall, the garbage collector improvements in Java 8, particularly the introduction of the G1 garbage collector, aimed to provide better performance, more predictable garbage collection pauses, and improved scalability for modern applications.

**Qn. what is finalize() in java [deprecated]**

In Java, the **finalize()** method is a special method provided by the **Object** class. It's called by the garbage collector when an object is about to be garbage collected, i.e., when there are no more references to the object and it's eligible for garbage collection.

Here are some key points about the **finalize()** method:

1. **Signature**:
   * The **finalize()** method has the following signature:

protected void finalize() throws Throwable

It's declared as **protected**, which means it's not directly accessible outside the class and its subclasses.

1. **Automatic Invocation**:
   * The **finalize()** method is automatically invoked by the garbage collector before reclaiming the memory occupied by an object.
2. **Finalization Process**:
   * When an object becomes unreachable and is eligible for garbage collection, the JVM schedules its **finalize()** method to be executed at some point in the future.
   * The actual execution timing of **finalize()** is non-deterministic and depends on various factors such as garbage collector implementation, JVM settings, and system load.
3. **Performance Impact**:
   * The use of **finalize()** can have performance implications, as it adds overhead to the garbage collection process. Therefore, it's generally recommended to avoid relying on **finalize()** for resource cleanup and instead use explicit resource management techniques such as **try-with-resources** or manual resource cleanup methods.
4. **Deprecation**:
   * Starting from Java 9, the **finalize()** method has been deprecated. This is because it has several drawbacks, including performance issues, unpredictability, and potential for misuse. It's recommended to use other mechanisms for resource cleanup and finalization.

Here's an example demonstrating the usage of the **finalize()** method:

public class MyResource {

// Resource cleanup logic

protected void finalize() throws Throwable {

try {

// Release any resources held by this object

// Close files, sockets, database connections, etc.

// ...

} finally {

super.finalize();

}

}

}

In modern Java programming, it's preferred to use explicit resource management techniques such as **try-with-resources** or manual cleanup methods (**close()** methods) for efficient and deterministic resource cleanup.

import java.io.FileInputStream;  
import java.io.IOException;  
  
public class TryWithResourcesExample {  
 public static void main(String[] args) {  
 // Define the path to the file  
 String filePath = "example.txt";  
  
 // Using try-with-resources to automatically close the FileInputStream  
 try (FileInputStream fis = new FileInputStream(filePath)) {  
 // Read and print each byte from the file  
 int byteRead;  
 while ((byteRead = fis.read()) != -1) {  
 System.*out*.print((char) byteRead);  
 }  
 } catch (IOException e) {  
 // Handle any IO exceptions  
 e.printStackTrace();  
 }  
 }  
}

**Qn. What are the different generations in the Java heap?**

In Java's garbage collection mechanism, the heap memory is divided into several generations, each serving a different purpose and having its own garbage collection strategy. The main generations in the Java heap are:

1. **Young Generation:**
   * This is where newly created objects are allocated.
   * It consists of two parts: **Eden space and Survivor space**.
   * Initially, objects are allocated in the Eden space. When the Eden space fills up, a minor garbage collection (also known as a young generation garbage collection) is triggered, during which live objects are moved to one of the Survivor spaces or promoted to the Old Generation.
2. **Old Generation:**
   * This is where long-lived objects are stored.
   * Objects that survive multiple garbage collection cycles in the Young Generation are eventually promoted to the Old Generation.
   * Major garbage collections are performed in the Old Generation, which typically involve longer pauses compared to minor garbage collections in the Young Generation.
3. **Permanent Generation (until Java 7):**
   * This generation is used to store metadata required by the JVM, such as class metadata, method information, and interned strings.
   * Starting from Java 8, the Permanent Generation was replaced by the Metaspace, which is not part of the heap but is allocated from native memory.
4. **Metaspace (from Java 8 onwards):**
   * Metaspace is a native memory area that stores class metadata, including ***bytecode, static fields, method information, and references to other classes.***
   * Unlike the Permanent Generation, Metaspace dynamically adjusts its size based on the application's demand and is garbage collected like regular heap memory.

The use of multiple generations allows the garbage collector to apply different collection algorithms and tuning parameters to each generation, optimizing memory management and reducing pause times. For example, young generation collections are typically fast and frequent, while old generation collections are less frequent but may cause longer pauses.

**Qn. Kindly give brief idea about JAVA memory model?**

**First we can discuss about heap, stack and metaspace then below concepts.**

The Java Memory Model (JMM) defines how threads interact through memory when executing Java programs. It specifies the rules and guarantees about how changes made by one thread become visible to other threads and how concurrent access to shared variables is handled.

In simple terms, the Java Memory Model ensures that:

1. **Visibility**: Changes made by one thread to shared variables are eventually visible to other threads. This means if one thread updates a variable, other threads will eventually see the updated value. volatile
2. **Atomicity**: Certain operations on shared variables are atomic, meaning they are performed as a single, indivisible unit. [can be achieved by synchronization, atomic classes, locksx]
3. **Ordering**: The order in which actions occur in one thread is consistent with the order seen by other threads. This means that if one thread performs actions A, B, and C, other threads will see those actions occur in the same order.

The Java Memory Model provides these guarantees while allowing the Java Virtual Machine (JVM) to optimize the execution of code for performance. It defines rules for how threads interact with memory, synchronization, and the behavior of certain types of variables like **volatile** and **final**.

Qn. **Explain the significance of the -Xms and -Xmx JVM options.**

* -Xms sets the initial heap size when the JVM starts, while -Xmx sets the maximum heap size that the JVM can use. These options control the minimum and maximum amount of memory allocated to the Java heap, allowing you to tune the JVM's memory usage according to your application's requirements.

**Qn. what is memory leak in java, how to identify it and how to fix it.**

A memory leak in Java occurs when objects are no longer needed by the application but are still held in memory, preventing them from being garbage collected. Over time, this can lead to the consumption of excessive memory, causing performance issues such as out-of-memory errors or increased garbage collection overhead.

Memory leaks can be challenging to identify because Java's garbage collector automatically reclaims memory for objects that are no longer referenced. However, if objects continue to be held in memory due to unintended references, memory leaks can occur.

some common reasons for memory leaks in Java along with examples:

1. **Unclosed Resources:**

**Reason**: Failure to close resources like files, database connections, or network sockets after using them can lead to memory leaks.

Solution : use try with resource

1. **Static Collections**:
   * **Reason**: Storing objects in static collections that live for the lifetime of the application can prevent them from being garbage collected, even when they are no longer needed.
   * **Example**:

public class Cache {  
 private static List<Object> *cache* = new ArrayList<>();  
  
 public static void addToCache(Object obj) {  
 *cache*.add(obj);  
 }  
}

1. **Circular References**:
   * **Reason**: Objects referencing each other in a circular manner can prevent them from being garbage collected even when they are no longer reachable.
   * **Example**:

public class Node {  
 private Node next;  
  
 public void setNext(Node next) {  
 this.next = next;  
 }  
}  
  
 Node node1 = new Node();  
 Node node2 = new Node();  
node1.setNext(node2);  
 node2.setNext(node1); // Circular reference

Here's how to identify and fix memory leaks in Java:

### **Identify Memory Leaks:**

1. **Use Profiling Tools:** Use memory profiling tools like ***VisualVM***, to analyze memory usage and identify potential leaks.
2. **Monitor Heap Usage:** Monitor heap memory usage over time using tools like ***jstat or jconsole*** to detect abnormal increases in memory consumption.
3. **Analyze Heap Dumps:** Analyze heap dumps generated by the JVM when an out-of-memory error occurs to identify objects that are consuming excessive memory.

### **Fix Memory Leaks:**

1. **Review Code:** Review the code to identify places where objects are being created and ensure they are properly dereferenced when no longer needed.
2. **Close Resources:** Always close resources such as files, streams, or database connections after use to release associated memory.
3. **Avoid Object Caching:** Avoid caching large objects unnecessarily, especially in long-lived caches, as they can lead to memory bloat. Use bounded caches with eviction policies to limit memory usage.
4. **Optimize Data Structures:** Review data structures and algorithms to ensure they are efficient and not inadvertently holding onto references longer than necessary.
5. **Test and Monitor:** Thoroughly test the application under different load conditions and monitor memory usage regularly to detect and address memory leaks proactively.

**Qn. what is the difference between stackoverflow and memoryOutofbound with an example?**

In Java, both **StackOverflowError** and **OutOfMemoryError** are runtime exceptions that occur when certain conditions are met, but they represent different types of issues related to memory management.

1. **StackOverflowError**:
   * This error occurs when the stack memory allocated for a thread's method calls exceeds its limit. Each thread in Java has its own stack, which is used to store method invocations and local variables.
   * Typically, this error happens due to excessive recursion, where a method calls itself recursively without reaching a base case to terminate the recursion.

public class StackOverflowExample {  
 public static void main(String[] args) {  
 *recursiveMethod*(1);  
 }  
  
 public static void recursiveMethod(int i) {  
 System.*out*.println(i);  
 *recursiveMethod*(i + 1); // Recursive call  
 }  
}

In this example, the **recursiveMethod** is called recursively without a base case, leading to **StackOverflowError** eventually because the stack memory is exhausted due to the increasing number of method calls.

**OutOfMemoryError**:

* This error occurs when the Java Virtual Machine (JVM) cannot allocate enough memory for new objects or when it runs out of heap space.
* It can happen for various reasons, such as memory leaks, excessive object creation, or insufficient heap size configuration.

public class OutOfMemoryExample {  
 public static void main(String[] args) {  
 List<Object> list = new ArrayList<>();  
 while (true) {  
 list.add(new Object()); // Keep adding objects to the list  
 }  
 }  
}

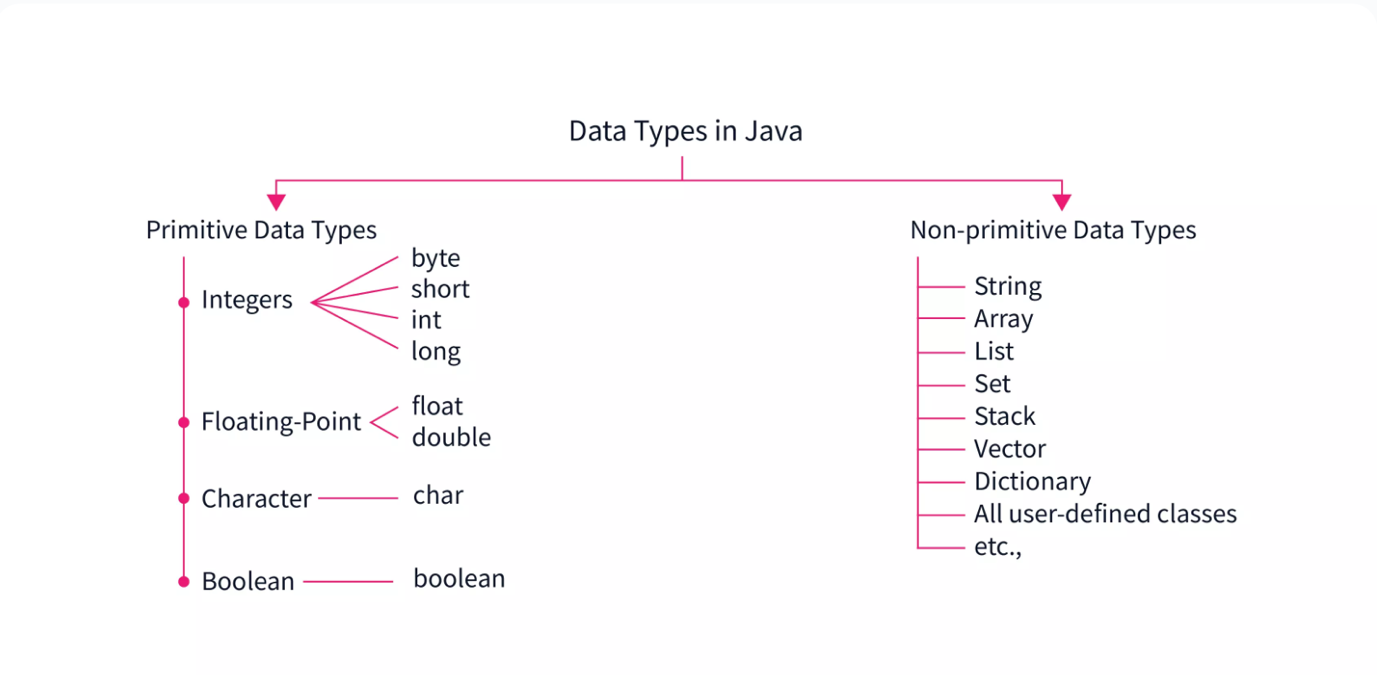
In this example, the **OutOfMemoryError** occurs because the **ArrayList** keeps adding objects indefinitely, consuming all available heap memory until it's exhausted.

In summary, **StackOverflowError** occurs when the stack memory is exhausted due to excessive method calls, typically caused by recursive methods. On the other hand, **OutOfMemoryError** occurs when the heap memory is exhausted, usually due to excessive object creation or insufficient heap space. Both errors indicate issues related to memory management but in different parts of the JVM memory.

**Qn. Static variables.**

* Static variable in Java are shared across all instances of a class. Static variables are associated with the class rather than with the objects.
* Static variables are loaded at the time of class compilation.
* Static variables can be used in any type of methods: **static** or **non-static**.
* **Non-static** variables cannot be used inside static methods. It will throw a compile-time error.
* Static variables are memory efficient as they are created only once per class. They are not created separately for each instance like instance variables.
* **Static variables** can be accessed using either the class name or instance name. However, it is recommend to access static variables using the class name.

**Qn. Datatypes in java**



**Qn. Wrapper classes in java**

Object classes for primitive data type. This is useful in scenarios where objects are required, such as when using collections, generics, or dealing with methods that expect objects rather than primitives. Wrapper classes provide methods to convert primitive data types into objects and vice versa, as well as additional utility methods for handling the data.

Here are the wrapper classes for primitive data types in Java:

1. **Byte**: **java.lang.Byte**
2. **Short**: **java.lang.Short**
3. **Integer**: **java.lang.Integer**
4. **Long**: **java.lang.Long**
5. **Float**: **java.lang.Float**
6. **Double**: **java.lang.Double**
7. **Character**: **java.lang.Character**
8. **Boolean**: **java.lang.Boolean**

These wrapper classes provide constructors and static methods for creating instances from primitive values, as well as methods to convert objects back to their corresponding primitive values. Additionally, they offer utility methods for tasks such as parsing strings into primitive values, comparing objects, and performing arithmetic operations.

Here's a simple example demonstrating the usage of wrapper classes:

public class WrapperExample {  
 public static void main(String[] args) {  
 Integer intValue = Integer.*valueOf*(10);  
 Double doubleValue = Double.*valueOf*(3.14);  
  
 // Converting objects back to primitive values  
 int primitiveInt = intValue.intValue();  
 double primitiveDouble = doubleValue.doubleValue();  
  
 // Performing arithmetic operations using wrapper classes  
 int sum = intValue + 20; // Automatically unboxing intValue to perform addition  
  
 // Using utility methods provided by wrapper classes  
 String numString = "100";  
 int parsedInt = Integer.*parseInt*(numString);  
 boolean isEqual = intValue.equals(parsedInt);  
 }  
}

**Qn. Access modifier in java.**

A screen shot of a white grid

Description automatically generated

The following steps can be followed to choose an appropriate access modifier based on the desired level of visibility.

* Java has four access modifiers: Public, Private, Default, and Protected.
* Private offers the most restricted access, while public offers the least.
* Private members are accessible only within their class.
* Public members are accessible from anywhere.
* Default, or package-private, access allows access within the same package.
* Protected access allows access within the same package or by subclasses in different packages.
* Private doesn't permit method overriding, but other modifiers do, both within the same package and across packages.

**Multithreading**

Multithreading in Java allows multiple threads of execution to run concurrently within a single Java program. Each thread represents a separate flow of control, allowing tasks to be performed asynchronously and concurrently. This enables better utilization of available CPU resources and can lead to improved performance and responsiveness in Java applications.

Here are some key points about multithreading in Java:

**Thread Vs Process:**

* **Process**: In Java, a process refers to an instance of a running program. Each Java application runs within its own process, which contains one or more threads. Processes are managed by the operating system and have their own memory space, resources, and runtime environment. Processes are typically created using the java.lang.ProcessBuilder class or by executing external programs using Runtime.exec() or ProcessBuilder.
* **Thread**: In Java, a thread is a lightweight unit of execution within a process. It shares the same memory space and resources as other threads within the same process. Threads are managed by the Java Virtual Machine (JVM) and are created and controlled using the java.lang.Thread class or the java.util.concurrent.Executor framework.

**Creating Threads**:

* + Threads can be created by extending the **Thread** class and overriding its **run()** method.
  + Alternatively, threads can be created by implementing the **Runnable** interface and passing an instance of the **Runnable** object to a **Thread** constructor.
  + Using the **Executor** framework and **ExecutorService** interface is a preferred way to manage threads in Java applications.

**Thread States**:

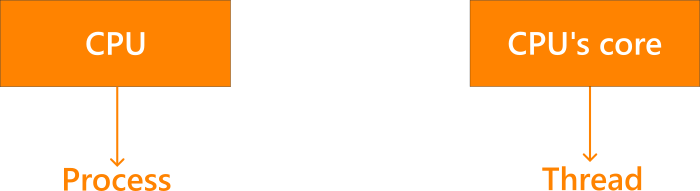
* + Threads in Java can be in various states, including:
    - New: When a thread is created but not yet started.
    - Runnable: When a thread is ready to run and waiting for CPU time.
    - Running: A thread is in the running state when it is actively executing its code on the CPU.
    - Blocked: When a thread is waiting for a monitor lock to enter a synchronized block or method.
    - Waiting and Timed Waiting: When a thread is waiting for another thread to perform a particular action or wait for a specified amount of time.
    - Terminated: When a thread completes its execution or is terminated forcibly.

**Thread Lifecycle**:

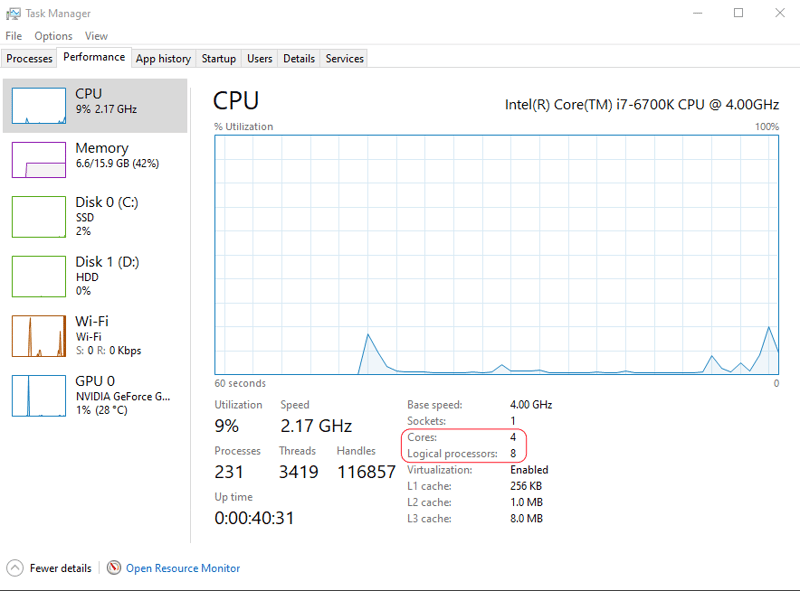
* + The lifecycle of a thread includes creation, starting, running, blocking, waiting, and termination.
  + Transitions between different thread states occur based on various factors like scheduling, synchronization, and I/O operations.

### Computer's interpretation of processes and threads

A process is a small program within an app, whereas a thread is a small program within a process. Your CPU (central processing unit) is responsible for executing them. A single CPU can run several threads but only one process at a time. Meanwhile, a CPU has several cores, each of them is responsible for running a single thread.

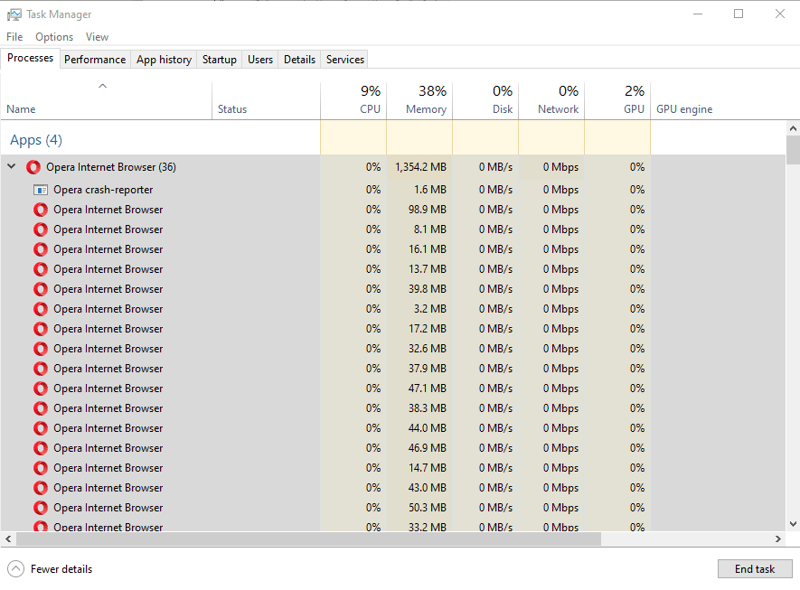
[](https://media.dev.to/cdn-cgi/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fo4vkm4q5iubm8dybmdhf.png)

For example, I have 8 CPUs and each of them contains 4 cores, so my computer can run 8 processes and 32 threads concurrently. To check how many you have open Task Manager, Performance tab.

[](https://media.dev.to/cdn-cgi/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fsj1zqhnu6eo9zhdb153w.png)

### Processes

An example of a process can be a single web application run by Opera browser. Since there are lots of web apps opened on different tabs running simultaneously, it’s appropriate for a browser to split them into different processes.

[](https://media.dev.to/cdn-cgi/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Ffye8wnwh2j81p1nxmu2j.png)

Although I have 17 tabs opened in my web browser, the number of processes is 35. How is my system not overloaded? The answer is in multitasking. Each CPU can switch rapidly between running processes. When some tasks get done, it switches to another one and this is done so fast that it manages to run all the applications and processes. That's why CPUs actually appear to run several processes at the same time, but in fact they don't.

**Now we are going to look at 5 ways to create threads in Java and the differences between them.**

## **1. By extending Thread class**

The most apparent (but in many cases not a proper) way to create a thread is to extend Thread class and override run() method. Use it only when you want to expand the functionality of Thread.

public class MyThread extends Thread {  
 @Override  
 public void run() {  
 System.*out*.println("Hello from a new thread!");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 MyThread myThread = new MyThread();  
 myThread.start();  
 }  
}

## **2. By implementing Runnable interface**

If you don’t want to expand Thread functionality but just want to do some stuff in a new thread, implementing Runnable is the best way to do it. But creating it, you only specify what should be done in a separate thread. To actually launch it you should create a new Thread, pass the Runnable as a parameter and run the start() method.

Thread thread = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("Hello from a new thread!");  
 }  
 });  
thread.start();

If you are going to use the Runnable several times, it’s worth making a separate interface instead of using an anonymous class. Otherwise, you should use lambdas to make it more concise (I didn’t use lambdas for clarity).

## **☠️ How does a Java thread die?**

When a thread has done all the operations it was programmed to do, it dies. And after that happens, you will not be able to start it again.

package javaconcepts.multithreading;  
  
public class Main {  
 public static void main(String[] args) throws Exception {  
 Thread stopTestThread = new Thread(  
 () -> {  
 System.*out*.println("Hello from " + Thread.*currentThread*().getName());  
 for (int i = 0; i < 3; i++) {  
 System.*out*.println("I'm running " + (i + 1));  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 throw new Runtime Exception(e);  
 }  
 }  
 System.*out*.println("Now I'm going to stop");  
 });  
  
 stopTestThread. Start();  
 stopTestThread. Join();  
 System.*out*.println();  
 System.*out*.println("Hello from " + Thread.*currentThread*().getName());  
 System.*out*.println("Trying to invoke " + stopTestThread. GetName());  
 stopTestThread.start();  
 }  
  
}

**Output:**

Hello from StopTestThread

I'm running 1

I'm running 2

I'm running 3

Now I'm going to stop

Hello from main

Trying to invoke StopTestThread

Exception in thread "main" java.lang.IllegalThreadStateException

In this code, we create a new thread and call it StopTestThread using the 2-param constructor. In the first param, we use lambda to create a new Runnable telling the thread what to do. In particular, the thread is going to greet the user, run for 3 sec, and tell when it's going to stop.

Then the newly created StopTestThread gets executed. While the separate thread does its work, the main thread, which executes every Java program by default, continues its work. It prints some statements and then tries to start the StopTestThread again.

You may notice that despite StopTestThread’s death we can call its getName() and isAlive() methods. However, it doesn’t mean we can bring it back to life.

### **join()**

You may also note that unless we write stopTestThread.join() StopTestThread and main will work simultaneously and the lines will not be printed sequentially, which is unwanted in our situation.

This method tells the thread that calls it (in our case main thread), to wait until the thread on which it’s being called (stopTestThread) finishes its work.

Another option is to specify a time interval as an argument for join. This way, the thread will pause for that duration and then resume its task regardless of whether another thread has completed the operation or not.

## **3. By creating a new thread pool**

When you use several threads in your program, you probably do this to speed up a process. But a Java thread corresponds to a system thread, and as was mentioned in [the previous article](https://dev.to/danielrendox/multithreading-in-java-part-1-process-vs-thread-2glf), their number is limited to the number of CPUs and their cores. Also remember, there are other apps that require some threads too.

Creating too many threads, for example, 100, is not efficient because only some of them will be scheduled. The rest will wait until the ones that are executing finish their work and die. Only then will they take their place. In addition, many threads consume lots of time and resources being born and dying.

That’s why, you should prefer thread pools over multiple instances of Thread. This way allows us to create a reasonable number of threads that will not die as long as they have done an operation, they will switch to another one instead. And this is how to do it:

ExecutorService executorService = Executors.newFixedThreadPool(threadsNumber);  
for (int i = 0; i < 20; i++) {  
 executorService.submit(  
 () -> System.*out*.println("Hello from: " + Thread.*currentThread*().getName())  
 );  
}  
executorService.submit(  
 () -> System.*out*.println("Another task executed by " + Thread.*currentThread*().getName())  
);  
  
executorService.shutdown();

Output:

Hello from: pool-1-thread-3

Hello from: pool-1-thread-8

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-6

Hello from: pool-1-thread-7

Hello from: pool-1-thread-5

Hello from: pool-1-thread-3

Hello from: pool-1-thread-2

Hello from: pool-1-thread-1

Hello from: pool-1-thread-4

Another task executed by pool-1-thread-6

For this, you just do the same but use ExecutorService. In the code above a new ThreadPool is created with the number of threads equal to threadsNumber. It’s usually a good decision to make as many threads as there are processors available for the JVM:

*int threadsNumber = Runtime.getRuntime().availableProcessors();*

The value of this number depends on the machine and its current state, such as the available resources and the apps that are running. For example, my computer had 8 available CPU cores at that time.

So the code above just creates an X number of threads, which do some action, 20 times, in this case, print their names. Mind you, the action is performed 20 times in total, not by each thread individually. You can also submit other tasks as I did below the for loop.

submit() method tells a thread what should be done and starts it. We should also shut down the executor to stop the program from running forever.

## **4. By using ExecutorService with Callable and Future**

All this time we’ve created new Threads using Runnable. But you may have noticed that it has a downside - it can’t return a value.

Of course, you can create a data class and use Runnable to keep some values. But there is a problem. In single-thread programs, you can use a variable immediately after it has been populated. In multi-thread programs, a variable may be assigned a value in a different thread. How do you determine if the variable has a value or not?

So this approach isn’t effective and may lead to errors, but the problem can be solved by using Callable instead of Runnable:

import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.\*;  
  
public class Main {  
 public static void main(String[] args) {  
 // Create an ExecutorService with multiple threads  
 ExecutorService executorService = Executors.*newFixedThreadPool*(3);  
  
 // List to store Future objects  
 List<Future<String>> futures = new ArrayList<>();  
  
 // Submit tasks to ExecutorService and obtain Future objects  
 for (int i = 0; i < 5; i++) {  
 Future<String> future = executorService.submit(new Task(i));  
 futures.add(future);  
 }  
  
 // Retrieve results from Future objects  
 for (Future<String> future : futures) {  
 try {  
 String result = future.get(); // Blocking call to get the result  
 System.*out*.println("Result: " + result);  
 } catch (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
  
 // Shutdown the ExecutorService  
 executorService.shutdown();  
 }  
  
 static class Task implements Callable<String> {  
 private final int taskId;  
  
 Task(int taskId) {  
 this.taskId = taskId;  
 }  
  
 @Override  
 public String call() {  
 try {  
 Thread.*sleep*(1000); // Simulate computation  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "Task " + taskId + " completed";  
 }  
 }  
}

In this example, we create an **ExecutorService** with three threads using **Executors.newFixedThreadPool(3)**. We then submit five tasks to the **ExecutorService**, each represented by a **Callable** object. The **submit()** method returns a **Future** object for each task, which we add to the list **futures**.

Finally, we iterate over the **Future** objects in the list and call the **get()** method to retrieve the result of each task. This operation is blocking, so it will wait until the task completes and returns its result.

**5 using CompletableFuture**

**CompletableFuture** is a class introduced in Java 8 that represents a future result of an asynchronous computation. It provides a way to perform operations asynchronously and handle the result when it becomes available.

Here's a simple example of how to use **CompletableFuture**:

public static void main(String[] args) {  
 // Create a CompletableFuture that will asynchronously compute a result  
 CompletableFuture<Integer> futureResult = CompletableFuture.*supplyAsync*(() -> {  
 // Simulate a long-running computation  
 try {  
 Thread.*sleep*(2000); // Sleep for 2 seconds  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return 42; // Return the result  
 });  
  
 // Attach a callback to handle the result when it becomes available  
 futureResult.thenAccept(result -> {  
 System.*out*.println("Result: " + result);  
 });  
  
 System.*out*.println("Main thread continues to execute while CompletableFuture runs asynchronously...");  
  
 // Wait for the CompletableFuture to complete (not recommended in real applications)  
 futureResult.join(); // This blocks the main thread until the CompletableFuture completes  
}

In this example:

* We create a **CompletableFuture** using the **supplyAsync** method, which takes a **Supplier** representing the computation to be performed asynchronously.
* Inside the **Supplier**, we simulate a long-running computation by sleeping for 2 seconds and then returning a result.
* We attach a callback to the **CompletableFuture** using the **thenAccept** method, which will print the result when it becomes available.
* The main thread continues to execute while the **CompletableFuture** runs asynchronously in the background.
* Finally, we use the **join** method to block the main thread and wait for the **CompletableFuture** to complete (not recommended in real applications).

**CompletableFuture** provides various methods for composing asynchronous computations, handling exceptions, and combining multiple **CompletableFutures** together. It's a powerful tool for asynchronous programming in Java.

**Qn. How to synchronize threads?**

Well, why should we synchronize them if the essence of multithreading is to perform operations asynchronously? Because with the benefits of multithreading come challenges. In this article, we’ll learn about problems that occur and ways to solve them.

Let’s consider this code:

class DonutStorage {  
 private int donutsNumber;  
 public DonutStorage(int donutsNumber) {  
 this.donutsNumber = donutsNumber;  
 }  
  
 public int getDonutsNumber() {  
 return donutsNumber;  
 }  
  
 public void setDonutsNumber(int donutsNumber) {  
 this.donutsNumber = donutsNumber;  
 }  
}  
  
class Consumer {  
 private final DonutStorage donutStorage;  
  
 public Consumer(DonutStorage donutStorage) {  
 this.donutStorage = donutStorage;  
 }  
  
 public void consume(int numberOfItemsToConsume) {  
 donutStorage.setDonutsNumber(donutStorage.getDonutsNumber() - numberOfItemsToConsume);  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 int consumersNumber = 10;  
 DonutStorage donutStorage = new DonutStorage(20);  
 ExecutorService executor = Executors.*newFixedThreadPool*(Runtime.*getRuntime*().availableProcessors());  
 for (int i = 0; i < consumersNumber; i++) {  
 executor.submit(() -> new Consumer(donutStorage).consume(1));  
 }  
 executor.shutdown();  
  
 System.*out*.println("Number of remaining donuts: " + donutStorage.getDonutsNumber());  
 }  
}

Here we have a simple program for a donut shop that counts the number of donuts. This shop probably has a server, which gets data from clients. As there are many clients who consume donuts, and the server must serve them simultaneously, it will have different threads each dedicated to one user.

To realize this in code, we create an ExecutorService with the number of threads equal to the available number of cores. Then we want it to serve the consumerNumber of clients, so there is a for loop that submits tasks to create a new Consumer that will consume items from the DonutStorage, and print the number of remaining donuts on the console.

Let’s assume that there are 10 consumers, each buys 1 donut, and we have 20 donuts in total. To prevent errors, if a user wants more donuts than there are in stock, the remaining donuts will be sold (the donutsNumber will be set to 0).

## Problem 1

So we expect the number of remaining donuts to be 10, but I have the following printed (you may have a different result):

Number of remaining donuts: 19

### Question

Why is that so? This problem is not connected with race condition.

### Answer

Because we didn’t call join so the main thread doesn’t wait for the others to finish.

Don’t get confused by executor.shutdown(). It doesn’t mean that the code below it is executed after the executor is shut down. Look, the main thread does 5 things:

* creates some variables;
* creates an executor;
* submits tasks to the executor;
* shuts down the executor;
* prints the result;

These operations are done sequentially, but, when it tells the executor to shut down, it doesn’t actually wait until it happens. According to the documentation of shutdown(), when this method is called,

ExecutorService initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

However, the main thread isn’t blocked and continues its work.

In other words, in my situation, only 1 of the 10 users had managed to get a donut, when the main thread printed the number of remaining ones. As threads can be scheduled in other ways, rerunning the program, we may get a different result.

## Solution

But how can we join the threads? With a simple Thread we would write myThread.join(). But how to do it with ExecutorService?

It’s not appropriate to use ExecutorService in this way. We should harness its Future power instead.  
Even though, executor.submit(()->{}) submits a Runnable, the method still returns a Future that never contains any value inside but can tell when the task gets finished.

To get the desired functionality, we should keep these futures from each submission in a list and then loop through them to make sure all the tasks are finished:

public static void main(String[] args) {  
 int consumersNumber = 1005;  
 DonutStorage donutStorage = new DonutStorage(2000);  
 ExecutorService executor = Executors.*newFixedThreadPool*(Runtime.*getRuntime*().availableProcessors());  
 List<Future<?>> futures = new ArrayList<>(consumersNumber);  
 for (int i = 0; i < consumersNumber; i++) {  
 futures.add(executor.submit(() -> new Consumer(donutStorage).consume(1)));  
 }  
 executor.shutdown();  
  
 // make the main thread wait for others to finish  
 for (Future<?> future: futures) {  
 try {  
 future.get();  
 } catch (InterruptedException | ExecutionException e) {  
 System.*out*.println("Exception while getting from future" + e.getMessage());  
 e.printStackTrace();  
 }  
 }  
  
 System.*out*.println("Number of remaining donuts: " + donutStorage.getDonutsNumber());  
}

This way the main thread gets blocked (waits) until the other threads finish their work.

BTW, it’s wrong to assume that getting only the final future is enough because the tasks aren’t done sequentially.

## Problem 2 — Multiple threads have access to the same variable at a time

You see, multiple threads have access to the private int donutsNumber in the DonutStorage at the same time. To change the value of this variable they should:

1. Get the variable
2. Set the variable

Because of the thread interleaving we may get something like this:

|  |  |
| --- | --- |
| The 5th thread reads the value of 15 |  |
|  | The 6th thread reads the value of 15 (as the 5th user hasn’t yet changed the value) |
| The 5th thread changes the value to 14 |  |
|  | The 6th thread changes the value to 14 |

Blank in this table means that the thread for whatever reason sits idle, which is completely normal.

So everything goes OK until the 5th and the 6th threads read the same value simultaneously and one item is automatically not counted. And it may not be the only case. **This behaviour is called a race condition.**

[](https://media.dev.to/cdn-cgi/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fss99qvwvlwbd3kdhodzu.jpg)

## Solution — **AtomicInteger**

To solve a synchronization problem we first should check if the solution is already written for us. In this case, there is one - AtomicInteger.

An atomic operation is an operation that is done at once. As we’ve seen, subtracting a number from a variable is not an atomic operation. Similarly, when you write i++, it seems to be atomic, but it’s not because the value is read and then written (so two operations not one).

There are classes like AtomicInteger, AtomicLong, AtomicBoolean, etc. They internally make these operations atomic and in this way solve the problem.

Here is the improved code:

class DonutStorage {  
 private final AtomicInteger donutsNumber;  
  
 public DonutStorage(int donutsNumber) {  
 this.donutsNumber = new AtomicInteger(donutsNumber);  
 }  
  
 public AtomicInteger getDonutsNumber() {  
 return donutsNumber;  
 }  
}  
  
class Consumer {  
 private final DonutStorage donutStorage;  
  
 public Consumer(DonutStorage donutStorage) {  
 this.donutStorage = donutStorage;  
 }  
 public void consume(int numberOfItemsToConsume) {  
 donutStorage.getDonutsNumber().addAndGet(-numberOfItemsToConsume);  
 }  
}

Firstly, we change the donutsNumber’s type to AtomicInteger, modify the respective getter, and delete the setter because we no longer need it. Then, the Consumer’s consume method should be modified to work with AtomicInteger. Let’s use its addAndGetmethod, which atomically adds the given number to the current value or subtracts from it if the given param is negative.

Mind you, it wouldn’t work if we’d modified getters and setters for donutsNumber to return int instead of making the consume method work with AtomicInteger. That’s because different threads would’ve ended up using int and bypassing the atomic nature of AtomicInteger. So be careful with this and make sure that you are using it correctly. However, this problem will be explained in more detail in the following section.

## Problem 3 — AtomicInteger may be not enough

Now consumers are allowed to take as many items as they want, but the number of donuts is limited. Let’s improve the program to satisfy this requirement.

So we’ll add an if-statement to the consume method that checks if the given number is bigger than the number of donuts in stock. When so, the number of donuts in stock will be set to 0. But it’ll be also good to know how many donuts a consumer actually consumes so let’s make the consume method return this number. We’ll also change the lambda in the Main to print the results in the form of “**Thread’s name consumed number of items**”. To test how it works, let’s also make the consumers try to consume more than is available, say 3 items each. The changes may look like this:

futures.add(executor.submit(() -> {

Consumer consumer = new Consumer(donutStorage);

System.out.println(Thread.currentThread().getName() + " consumed " +

consumer.consume(3)); // changed the number from 1 to 3

}));

/\*\*

\* Subtracts the given number from the DonutStorage's donutsNumber. If the given number is bigger

\* than the number of donuts in stock, sets the donutsNumber to 0.

\* @param numberOfItemsToConsume Number that will be subtracted from the donutsNumber

\* @return the number of consumed items

\*/

public int consume(int numberOfItemsToConsume) {

AtomicInteger donutsNumber = donutStorage.getDonutsNumber();

// if there aren't enough donuts in stock, consume as many as there are

if (numberOfItemsToConsume > donutsNumber.get()) {

int result = donutsNumber.get();

donutsNumber.set(0);

return result;

}

donutStorage.getDonutsNumber().addAndGet(-numberOfItemsToConsume);

return numberOfItemsToConsume;

}

You can compare the old and the updated versions [here](https://github.com/DanielRendox/ThreadSynchronizationInJava/commit/e00c502bc0471dfaf43ab358d04e43ac91ad3917?diff=unified).

Running the code, we can get the desired result!

pool-1-thread-1 consumed 3

pool-1-thread-3 consumed 3

pool-1-thread-6 consumed 3

pool-1-thread-8 consumed 0

pool-1-thread-7 consumed 2

pool-1-thread-1 consumed 0

pool-1-thread-2 consumed 3

pool-1-thread-5 consumed 3

pool-1-thread-4 consumed 3

pool-1-thread-7 consumed 0

Number of remaining donuts: 0

You see how threads interleave. The one which gets only 2 items doesn’t go after the others that get 3 items. It’s completely OK. What’s not OK is that if we add a simple action that takes some time into the consume method, for example, printing some random text it will break the program. Check this out:

public int consume(int numberOfItemsToConsume) {

AtomicInteger donutsNumber = donutStorage.getDonutsNumber();

// if there aren't enough donuts in stock, consume as many as there are

if (numberOfItemsToConsume > donutsNumber.get()) {

int result = donutsNumber.get();

donutsNumber.set(0);

return result;

}

// printing some random text breaks the program

System.out.println("The UFO flew in and left this inscription here.");

donutStorage.getDonutsNumber().addAndGet(-numberOfItemsToConsume);

return numberOfItemsToConsume;

}

Output:

The UFO flew in and left this inscription here.

The UFO flew in and left this inscription here.

The UFO flew in and left this inscription here.

The UFO flew in and left this inscription here.

The UFO flew in and left this inscription here.

The UFO flew in and left this inscription here.

The UFO flew in and left this inscription here.

The UFO flew in and left this inscription here.

pool-1-thread-4 consumed 3

pool-1-thread-4 consumed -4

pool-1-thread-4 consumed 0

pool-1-thread-6 consumed 3

pool-1-thread-5 consumed 3

pool-1-thread-7 consumed 3

pool-1-thread-3 consumed 3

pool-1-thread-2 consumed 3

pool-1-thread-1 consumed 3

pool-1-thread-8 consumed 3

Number of remaining donuts: 0

Despite the resulting number still being 0, we have issues with the resulting value of consume method. Why is that so?

That’s because printing some text introduces a delay resulting in the interleaving of the threads in a way that causes another **race condition**. it doesn’t mean that we shouldn’t just write some random text and the program will work fine. On another person’s computer, it may return an unexpected output without the random text, because their computer may be slower/faster/less or more busy, etc.

The question is why the program breaks despite AtomicInteger. Another question is why the race condition doesn’t happen in the previous section’s code. And you are about to get the answers.

### Explanation

That’s because AtomicInteger only provides atomic access to the value it holds and uses a non-blocking algorithm. And that means that multiple threads may access the value at the same time. Whereas, our program contains consume method that gets executed by multiple threads simultaneously.

Here is a possible scenario:

|  |  |
| --- | --- |
| The 6th thread reads the value of 5 and gets false in the if-condition |  |
|  | The 7th thread reads the value of 5 and gets false in the if-condition either |
| The 6th thread atomically reads 5 and writes 2 |  |
|  | The 7th thread atomically reads 2 and writes -1 |

You see that the 7th user may make the value negative because it has already passed the check and moved from the if-statement when it updates the value.

## Solution — synchronized keyword

Luckily, the solution to this is quite simple — just use the synchronized keyword for the critical section. In our case, it’s the whole consume method. This will tell the JVM that only one thread at a time can execute the critical section.

Here is some info you need to know regarding this:

In Java, this keyword [can be used with (both static and non-static) methods and code blocks](https://stackoverflow.com/questions/2056243/java-synchronized-block-for-class):

public synchronized void m1(){}

public static synchronized void m2(){}

synchronized (this) {}

synchronized (MyClass.class) {}

### Synched non-static method

This means that when there is an Object, for instance, donutStorage that has the following method:

public synchronized void setDonutsNumber(int donutsNumber) {

this.donutsNumber = donutsNumber;

}

**only 1 thread will be allowed to execute it (set the variable) at a time.**

### Another synched non-static method

And when there is another synchronized method in this class:

public synchronized int getDonutsNumber() {

return donutsNumber;

}

**another thread will not be allowed to execute it**, while the first one is executing setDonutsNumber(), even though this method does not depend on setDonutsNumber()

However, if another synchronized method is called within a synchronized method or code block it will not cause any issues and the thread executing it will freely run that method.

// the thread that is executing m1 will freely get access to m2

// and execute it, even though m2 is synchronized

public synchronized void m1() {

m2();

}

public synchronized void m2() {}

### Simple unsynched method

However, if there is an unsynchronized method in this class, for example:

public void optimizeDatabase() {}

**multiple other threads will be allowed to execute it**

### Static synched method

In addition, if there is a static synchronized method in this class, for example:

public static synchronized double calculateAverageDonutWeight(Donut[] donuts) {}

**another thread will be allowed to execute it**, even though the first one is executing setDonutsNumber because one of them is static and another one is non-static.

Similarly to non-static, if there is another static synchronized method\*,\* other threads will be restricted from executing it, while this one is being executed.

### Sounds overwhelming. How to understand it?

Sounds overwhelming, but in fact, this is pretty easy to understand.

A synchronized method tells the thread executing it to acquire a lock, which doesn’t allow any other threads to pass if they don’t have the lock.

By default, there are only two locks associated with a class: class-level lock (for static methods), and object-level lock (for non-static methods).

Synchronized blocks require an explicit indication of what lock they use.

synchronized (this) {} // object-level lock

synchronized (MyClass.class) {} // class-level lock

There is a good visual explanation in [this video](https://youtu.be/IIgHG_YHXPE). It is also a good idea to play around with code to understand the topic and put the necessary info into your long-term memory.

### Updated code

Mind you, in this case, when we use synchronized, AtomicInteger is no longer needed. That’s because it’s only consume method that causes the race condition and no one else. Keeping AtomicInteger will only cause performance degradation. However, if you eventually decide to get multithreaded access to the DonutStorage in other parts of your code, you will need to use synchronized either (or find another solution).

Here’s the updated code:

// DonutStorage is the same as it was in the beginning

public class DonutStorage {

private int donutsNumber;

public DonutStorage(int donutsNumber) {this.donutsNumber = donutsNumber;}

public int getDonutsNumber() {return donutsNumber;}

public void setDonutsNumber(int donutsNumber) {this.donutsNumber = donutsNumber;}

}

public int consume(int numberOfItemsToConsume) {

synchronized (donutStorage) {

int donutsNumber = donutStorage.getDonutsNumber();

// if there aren't enough donuts in stock, consume as many as there are

if (numberOfItemsToConsume > donutsNumber) {

donutStorage.setDonutsNumber(0);

return donutsNumber;

}

donutStorage.setDonutsNumber(donutsNumber - numberOfItemsToConsume);

return numberOfItemsToConsume;

}

}

You can compare the old and the updated versions [here](https://github.com/DanielRendox/ThreadSynchronizationInJava/commit/695039618cd7bc14d298c7cd58ae5b8c31ee4a40).

Also, note that making the consume method synchronized wouldn’t work. That’s because a thread entering it would have to acquire a lock for the Consumer object, while we want it to get a lock for the donutStorage.

There is no point in getting a lock for the Consumer object because each thread has its own Consumer object therefore each Consumer is used by only one thread. On the other hand, donutStorage is a single instance of the DonutStorage class throughout our code. Getting a lock for it before executing the consume method forces other threads to wait until the current one releases that lock.

Output:

pool-1-thread-3 consumed 0

pool-1-thread-7 consumed 3

pool-1-thread-5 consumed 3

pool-1-thread-8 consumed 3

pool-1-thread-1 consumed 3

pool-1-thread-2 consumed 3

pool-1-thread-4 consumed 2

pool-1-thread-6 consumed 3

pool-1-thread-8 consumed 0

pool-1-thread-3 consumed 0

Number of remaining donuts: 0

You can also add that print-random-text command to make sure that it works properly.

**Qn. difference between AtomicInteger vs synchronization?**

**AtomicInteger** and synchronization are both mechanisms used in Java to ensure thread safety when accessing shared mutable variables. However, they have different characteristics and are suitable for different scenarios.

1. **AtomicInteger:**
   * **AtomicInteger** is a part of the **java.util.concurrent.atomic** package and provides atomic operations on an integer value.
   * It uses lock-free algorithms internally, making it more efficient than synchronization in many cases, especially in scenarios with high contention.
   * It offers methods such as **get**, **set**, **getAndIncrement**, **getAndDecrement**, **getAndAdd**, etc., which are executed atomically without the need for explicit synchronization.
   * It is typically used for simple operations on single variables where fine-grained control over atomicity is required.
2. **Synchronization:**
   * Synchronization in Java is achieved using **synchronized** blocks or methods, or by using **Lock** objects from the **java.util.concurrent.locks** package.
   * Synchronization ensures **mutual exclusion** *[only one thread can access a shared resource at a time, preventing conflicts and maintaining data integrity]* by allowing only one thread to execute a synchronized block of code or method at a time.
   * It is more flexible than atomic operations and can be used for protecting multiple variables or executing more complex sequences of operations atomically.
   * Synchronization introduces more overhead compared to atomic operations, especially in scenarios with low contention, as it involves acquiring and releasing locks.

**When to Use Which:**

* Use **AtomicInteger** when you need to perform simple operations (like increments, decrements, additions) on a single integer variable in a highly concurrent environment.
* Use synchronization when you need to protect multiple variables or execute complex sequences of operations atomically. Synchronization is more flexible and can be applied to a wider range of scenarios, but it may introduce more overhead compared to atomic operations.

In general, if you're dealing with simple operations on a single variable and performance is critical, **AtomicInteger** is often preferred. If you need more complex synchronization or coordination between threads, synchronization mechanisms like **synchronized** blocks or locks may be more appropriate.

**Executors**

The **java.util.concurrent.Executors** class provides factory and utility methods for creating different types of thread pools in Java. Each type of executor serves different use cases and has its own advantages and limitations. Here's an explanation of some common types of executors along with examples and recommendations for their usage:

1. **SingleThreadExecutor**:
   * This executor uses a single worker thread to execute tasks sequentially.
   * It's suitable for background tasks that need to be executed sequentially or for offloading non-blocking I/O operations.
   * Example:

ExecutorService executor = Executors.newSingleThreadExecutor(); executor.submit(() -> {  
 // Task 1 });  
 executor.submit(() -> {  
// Task 2 });  
 executor.shutdown();

1. **FixedThreadPool**:
   * This executor maintains a fixed number of threads in the pool.
   * It's useful when you have a specific number of tasks to execute concurrently and want to limit resource usage.
   * Example:

ExecutorService executor = Executors.newFixedThreadPool(5);  
for (int i = 0; i < 10; i++) {  
 executor.submit(() -> { // Task   
 });  
}  
executor.shutdown();

1. **CachedThreadPool**:
   * This executor creates new threads as needed and reuses existing ones if available.
   * It's suitable for executing a large number of short-lived asynchronous tasks.
   * Example:

ExecutorService executor = Executors.newCachedThreadPool();  
for (int i = 0; i < 100; i++) {  
 executor.submit(() -> { // Task  
 });  
}  
executor.shutdown();

1. **ScheduledExecutorService:**
   * This executor is capable of scheduling tasks to run after a certain delay or at regular intervals.
   * It's useful for tasks that need to be executed periodically, such as polling or cleanup tasks.
   * Example:

ExecutorService executor = Executors.newScheduledThreadPool(1); executor.scheduleAtFixedRate(() -> {  
 // Task to be executed periodically   
}, 0, 1, TimeUnit.SECONDS);  
// Initial delay of 0 seconds, repeat every 1 second

**Recommendations**:

* Use **SingleThreadExecutor** when tasks need to be executed sequentially or you want to dedicate a single thread to the tasks.
* Use **FixedThreadPool** when you have a fixed number of tasks to execute concurrently and want to limit resource usage.
* Use **CachedThreadPool** when you have a large number of short-lived asynchronous tasks and want to minimize thread creation overhead.
* Use **ScheduledExecutorService** when tasks need to be scheduled to run after a delay or at regular intervals.

Choose the appropriate executor based on the specific requirements and characteristics of your application.

**Synchronization, Atomicity, Visibility:**

1. **Synchronized:**
   * **synchronized** keyword is used to provide exclusive access to a shared resource or critical section of code among multiple threads.
   * It ensures that only one thread can execute the synchronized block or method at a time, preventing concurrent access and maintaining data consistency.
   * It can be applied at the method level (**synchronized** methods) or block level (**synchronized** blocks).
   * Example:

javaCopy code

public class SynchronizedExample {  
 private int count = 0;   
 public synchronized void increment() {  
 count++;  
 }  
}

* + Use synchronized when you need to ensure thread safety by preventing concurrent access to shared resources, especially in scenarios where multiple threads may read and write to the same data simultaneously.

1. **Atomicity:**
   * Atomicity refers to the property of an operation where it appears to execute in a single, indivisible step, without interruption or interference from other threads.
   * In Java, atomic operations are typically achieved using classes from the **java.util.concurrent.atomic** package, such as **AtomicInteger**, **AtomicBoolean**, etc.
   * These classes provide atomic versions of common operations like increment, decrement, compare-and-set, etc., ensuring that these operations execute atomically without the need for explicit synchronization.
   * **AtomicInteger** class itself already ensures visibility guarantees for its operations without the need for **volatile**.
   * Example:

public class AtomicityExample {  
 private AtomicInteger count = new AtomicInteger(0);  
  
 public int incrementAndGet() {  
 return count.incrementAndGet();  
 }  
}

* + Use atomic operations when you need to perform simple operations atomically without the overhead of synchronization, especially in high-concurrency scenarios.

1. **Volatile:**
   * **volatile** keyword is used to indicate that a variable's value may be modified by multiple threads asynchronously.
   * It ensures visibility of changes to the variable across threads, meaning that changes made by one thread are immediately visible to other threads.
   * Unlike **synchronized**, **volatile** does not provide mutual exclusion or atomicity; it only ensures visibility.
   * Example:

public class VolatileExample {  
 private volatile boolean flag;  
  
 public void setFlag(boolean value) {  
 this.flag = value;  
 }  
  
 public boolean isFlag () {  
 return flag;  
 }  
}

* + Use **volatile** when you need to ensure that changes made to a variable by one thread are immediately visible to other threads, especially for flags or status variables.

**Relationships and Differences:**

* **synchronized** and atomic operations (**Atomic** classes) both ensure atomicity, but they achieve it through different mechanisms. **synchronized** uses locks to provide mutual exclusion, while atomic operations use compare-and-swap (CAS) operations for atomicity without locking.
* **volatile** ensures visibility of changes to a variable across threads but does not provide atomicity or mutual exclusion.
* All three concepts are related in that they address different aspects of concurrent programming: **synchronized** ensures mutual exclusion, atomicity ensures indivisibility of operations, and **volatile** ensures visibility of changes.
* The choice of which to use depends on the specific requirements of your application:
  + Use **synchronized** when you need mutual exclusion and synchronization between threads.
  + Use atomic operations (**Atomic** classes) when you need atomicity for simple operations without the overhead of synchronization.
  + Use **volatile** when you need visibility of changes to a variable across threads without synchronization, especially for simple flags or status variables.

**What are differences between Sleep and wait in java?**

|  |  |  |
| --- | --- | --- |
| **Parameter** | **wait** | **sleep** |
| Synchronized | wait should be called from synchronized context i.e. from block or method, If you do not call it using synchronized context, it will throw IllegalMonitorStateException | It need not be called from synchronized block or methods |
| Calls on | wait method operates on Object and defined in Object class | Sleep method operates on current thread and is in java.lang.Thread |
| Release of lock | wait release lock of object on which it is called and also other locks if it holds any | Sleep method does not release lock at all |
| Wake up condition | until call notify() or notifyAll() from Object class | Until time expires or calls interrupt() |
| static | wait is non static method | sleep is static method |

**Why wait(), notify() And notifyAll() methods are in Object Class**

a monitor refers to a synchronization mechanism that is associated with an object. Every object in Java has an associated monitor, which is used to control access to the object's synchronized methods or blocks.

When a thread enters a synchronized method or block associated with an object, it automatically acquires the monitor of that object. This means that no other thread can enter any synchronized method or block associated with the same object until the first thread releases the monitor by exiting the synchronized code.

In Java, thread ***waits*** on **monitor** assigned to the object and when you want to send a signal to another thread who is waiting for the same monitor, you call ***notify()*** method to ***wake*** one thread and notifyAll() to wake up all the threads.

If wait, notify and notifyAll method are in thread class, then each [thread](https://java2blog.com/java-thread-example/) should be aware of the status of another thread.

For example: Let’s say you have two threads, T1 and T2. Now T1 must know that T2 was waiting for this particular resource which I have just freed because T1 will need to call T2.notify().

In Java, the **object itself is shared** among the threads and facilitates [inter-thread communication](https://java2blog.com/wait-notify-and-notifyall-method-in/). **Threads have no specific knowledge of each other**. They can run asynchronously and are independent. They just run, lock, wait and get notified. They do not need to know about the status of other threads. They just need to call notify method on an object, so whomever thread is waiting on that resource will be notified.

As stated before,

* When you call ***wait()*** method on the object, then it gives up monitor and go to [sleep](https://java2blog.com/java-thread-sleep-example/)
* When you call ***notify()*** method, then the single thread which is waiting for the object’s monitor will be notified.

Hence wait, notify() And notifyAll() work at object’s monitor level. If thread which is currently holding the monitor, wants to give up the monitor then it will call wait method on the object and if it want to notify other thread, then it will call notify method on the object.

Shared objects allow threads to communicate by calling ***wait()***, ***notify()*** And ***notifyAll()*** Methods, so these

**Why sleep() and yield() are static methods in Thread class?**

In Java, both **sleep()** and **yield()** are static methods in the **Thread** class because they are intended to affect the behavior of the currently executing thread, rather than any specific instance of **Thread**.

1. **sleep(long millis)**: This method causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. Since it affects the thread calling the method, it is static to indicate that it operates on the current thread.

javaCopy code

Thread.sleep(1000); // Sleep for 1 second

1. **yield()**: This method is a hint to the scheduler that the current thread is willing to yield its current use of a processor. It's a way of saying to the thread scheduler that the current thread is not in a state of running, and other threads can have a chance to run. It's also static because it's meant to affect the current executing thread.

javaCopy code

Thread.yield(); // Hint to the scheduler to allow other threads to run

Both methods operate on the current thread's execution and are static to reflect this behavior. They provide a convenient way for a thread to control its execution without needing to obtain a reference to itself.

Top of Form

Bottom of Form

**Qn. What is Object and class level locking?**

**Object level locking:**

Object level locking means you want to synchronize non static method or block so that it can be accessed by only one thread at a time for that instance. It is used if you want to protect non static data.

You can achieve Object level locking by following.

**Make method synchronized:**

public synchronized **int** incrementCount()

{

}

**Using synchronized block and lock on this:**

public **int** incrementCount() {

  synchronized (**this**) {

   count++;

**return** count;

  }

**Using synchronized block and lock on some other object:**

private final **Object** lock = **new** **Object**();

public **int** incrementCount() {

  synchronized (lock) {

   count++;

**return** count;

  }

**Class level locking:**

Class level locking means you want to synchronize static method or block so that it can be accessed by only one thread for whole class. If you have 10 instances of class, only one thread will be able to access only one method or block of any one instance at a time. It is used if you want to protect static data.

This can be achieved by following:

**Make static method synchronized:**

public static synchronized **int** incrementCount()

{}

**Using synchronized block and lock on .class:**

public **int** incrementCount() {

  synchronized (RequestCounter.**class**) {

   count++;

**return** count;

  }

**Using synchronized block and lock on some other static object:**

private final static **Object** lock=**new** **Object**();

public **int** incrementCount() {

  synchronized (lock) {

   count++;

**return** count;

  }

Qn**. How will you simulate concurrent modification through stream API ?**

Ans: To simulate concurrent modification using the Stream API in Java, you can use parallel streams along with side-effect operations that modify the underlying data structure. Here's an example:

package javaconcepts.inbuildDS;  
import java.util.ArrayList;  
import java.util.List;  
public class ConcurrentExceptionAndPrevention {  
 public static void main(String[] args) {  
 //below code will work  
 List<Integer> numbers = new ArrayList<>();  
 for (int i = 0; i < 10; i++) {  
 numbers.add(i);  
 }  
 List<Integer> modifiedNumbers =  
 numbers.parallelStream()  
 .flatMap(  
 num -> {  
 List<Integer> temp = new ArrayList<>();  
 if (num % 2 == 0) {  
 temp.add(num \* 10);  
 }  
 return temp.stream();  
 })  
 .toList();  
  
 // Print the modified list  
 System.*out*.println("Modified numbers: " + modifiedNumbers);  
 // Simulate concurrent modification using parallel stream  
 List<Integer> numbers1 = new ArrayList<>();  
 for (int i = 0; i < 10; i++) {  
 numbers1.add(i);  
 }  
  
 // Simulate concurrent modification using parallel stream  
 numbers1.parallelStream()  
 .forEach(num -> {  
 if (num % 2 == 0) {  
 numbers1.add(num \* 10); // This will throw ConcurrentModificationException  
 }  
 });  
 }  
}

In this example, we have a list of integers numbers, and we are attempting to modify it concurrently using a parallel stream. Inside the stream operation, we are trying to add elements to the list based on a condition (num % 2 == 0). Since the stream operates concurrently, it may attempt to modify the list while other modifications are in progress, leading to a concurrent modification exception (ConcurrentModificationException). This exception occurs because the underlying list is modified while it is being iterated over by the parallel stream, violating the fail-fast behavior of the collections framework.

**Qn. How many threads will open for parallel stream and how parallel stream internally works?**

Ans. The number of threads used by a parallel stream depends on the characteristics of the underlying **ForkJoinPool** that manages the parallel execution. By default, the common pool is used, which is initialized with a number of threads equal to the number of available processors on the system.

When you create a parallel stream, the stream operations are divided into smaller tasks that can be executed concurrently. These tasks are then submitted to the ForkJoinPool for execution. The ForkJoinPool manages a pool of worker threads, and tasks are distributed among these threads for parallel execution.

Internally, parallel streams use the fork/join framework provided by the java.util.concurrent package. This framework divides the tasks into smaller subtasks recursively until they are small enough to be executed independently. These subtasks are then executed in parallel by multiple worker threads in the ForkJoinPool. Once the subtasks are completed, the results are combined to produce the final result.

Overall, parallel streams provide a convenient way to leverage multi-core processors and achieve parallelism for stream operations, potentially improving the performance of processing large datasets.

**Qn. How does Executor make or check, number of threads are active or dead, in other word what is internal working of thread pool executor? Explain with an example.**

The ThreadPoolExecutor in Java manages a pool of worker threads and executes tasks concurrently. It internally maintains several components to efficiently manage the execution of tasks. Let's explore its internal working with an example:

package javaconcepts.multithreading;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
public class ThreadPoolExample {  
 public static void main(String[] args) {  
 // Create a fixed-size thread pool with 3 threads  
 ExecutorService executor = Executors.*newFixedThreadPool*(3);  
 // Submit tasks to the thread pool  
 for (int i = 0; i < 5; i++) {  
 final int taskNumber = i;  
 executor.submit(() -> {  
 String threadName = Thread.*currentThread*().getName();  
 System.*out*.println("Task " + taskNumber + " executed by thread: " + threadName);  
 try {  
 Thread.*sleep*(2000); // Simulate task execution time  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
 }  
 // Shutdown the executor  
 executor.shutdown();  
 }  
}

We create a ThreadPoolExecutor using Executors.newFixedThreadPool(3), which creates a fixed-size thread pool with 3 threads.

We submit 5 tasks to the thread pool using executor.submit(Runnable task). Each task is a lambda expression that prints a message indicating the task number and the thread executing it, simulates some task execution time, and then completes.

After submitting all tasks, we call executor.shutdown() to shut down the executor once all tasks have completed.

Now, let's understand the internal workings of the ThreadPoolExecutor:

**Worker Threads**: When tasks are submitted to the executor, it assigns them to worker threads from the thread pool. In our example, since we have 5 tasks and a pool size of 3, some tasks will wait in the task queue until a worker thread becomes available.

**Task Queue**: Tasks waiting to be executed are stored in a task queue. In our example, tasks that cannot be immediately executed due to the limited number of threads in the pool are queued until a thread becomes available.

**Thread Management**: The ThreadPoolExecutor dynamically manages the number of active threads based on the workload. It creates new threads up to the core pool size (3 in our example) if there are pending tasks in the queue. If the workload increases beyond the core pool size, additional threads may be created up to the maximum pool size. Once the workload decreases, excess threads may be terminated to conserve resources.

**Thread States**: Each worker thread in the thread pool can be in different states, such as running, waiting, or terminated. The ThreadPoolExecutor keeps track of the state of each thread and manages their lifecycle accordingly.

Overall, the ThreadPoolExecutor efficiently manages the execution of tasks by optimizing the utilization of available resources and adjusting the number of active threads dynamically based on the workload, ensuring smooth execution of concurrent tasks.

**Qn. How to start on Parallel execution using thread pools in java.**  
Let’s take an example with a service that accepts input from multiple sources (API, Kafka), performs some calculations, and interacts with a database. We'll consider the same hypothetical machine specifications and apply the concepts mentioned earlier:

CPU Cores: Our machine has 4 CPU cores.

Memory: We have 8 GB of memory.

**Task Characteristics:**

**Input Sources:** The service receives inputs from both API and Kafka. These inputs are I/O-bound operations and can benefit from concurrency to overlap I/O operations.

**Calculations:** The service performs calculations, which may include both CPU-bound and I/O-bound tasks.

**Database Interaction:** Interacting with the database involves I/O-bound operations.

**Workload Variability:** The workload may vary based on the incoming requests from API and Kafka. It's essential to have enough concurrency to handle peak loads efficiently.

Considering these factors, let's determine an example number of threads in the thread pool:

Input Sources: Since the inputs from API and Kafka are I/O-bound, having more threads than CPU cores could help overlap I/O operations and improve throughput. Let's allocate 2 threads for handling input from API and 2 threads for processing messages from Kafka.

Calculations: Depending on the complexity of calculations, we may allocate additional threads to handle CPU-bound tasks efficiently. Let's allocate 2 threads for CPU-bound calculations.

Database Interaction: Interacting with the database involves I/O-bound operations. We can reuse the threads allocated for handling input sources and calculations for interacting with the database.

Based on these considerations, we allocate a total of 6 threads in the thread pool:

2 threads for handling input from API

2 threads for processing messages from Kafka

2 threads for CPU-bound calculations and database interaction

This allocation provides enough concurrency to handle I/O-bound tasks efficiently while also addressing occasional CPU-bound tasks. However, it's essential to monitor system metrics and adjust the thread pool size based on actual performance and workload characteristics to optimize resource utilization and throughput.

**Qn. Parallelism Vs concurrency ?**

Parallelism and concurrency are related concepts in the context of computer science and multi-threaded programming, but they represent different aspects of program execution:

1. **Concurrency**:
   * Concurrency refers to the ability of a system to execute multiple tasks or processes concurrently. These tasks may start, run, and complete in overlapping periods, but they don't necessarily execute simultaneously.
   * Concurrency is often achieved through techniques like multitasking, where a single processor switches between different tasks rapidly, or through multi-threading, where multiple threads within a single process execute concurrently.
   * Concurrency is useful for improving resource utilization and responsiveness in systems that handle multiple tasks simultaneously, such as web servers, operating systems, and database systems.
2. **Parallelism**:
   * Parallelism refers to the simultaneous execution of multiple tasks or processes, where each task runs independently and may be executed by separate processing units (e.g., CPU cores).
   * Parallelism enables tasks to be divided into smaller sub-tasks that can be executed concurrently, allowing for faster execution and improved performance.
   * Parallelism is often achieved by distributing tasks across multiple processors, cores, or machines, and coordinating their execution to achieve a common goal.
   * Parallelism is commonly used in high-performance computing, scientific simulations, data processing, and other computationally intensive tasks.

In summary, concurrency focuses on the ability of a system to handle multiple tasks simultaneously, while parallelism focuses on the actual simultaneous execution of tasks across multiple processing units. Concurrency is more about the design and structure of programs to handle multiple tasks concurrently, while parallelism is about utilizing hardware resources to execute tasks simultaneously for improved performance.

The choice between concurrency and parallelism depends on the specific requirements and characteristics of the application:

1. **Concurrency**:
   * Use concurrency when the application needs to handle multiple tasks simultaneously, but these tasks can share resources or don't require truly simultaneous execution.
   * Concurrency is suitable for applications with I/O-bound tasks (e.g., network communication, file I/O) where tasks spend a significant amount of time waiting for external resources.
   * It's also beneficial for applications that need to maintain responsiveness and handle multiple user interactions concurrently, such as web servers, GUI applications, and real-time systems.
2. **Parallelism**:
   * Use parallelism when the application needs to perform computationally intensive tasks that can be divided into independent sub-tasks, and these tasks can be executed simultaneously to achieve faster processing.
   * Parallelism is suitable for applications with CPU-bound tasks (e.g., mathematical calculations, data processing) where tasks can be divided into smaller chunks and executed concurrently across multiple processing units.
   * It's also useful for applications that require high-performance computing, such as scientific simulations, machine learning, and data analysis.

In summary:

* Use **concurrency** for applications with many I/O-bound tasks or interactions that need to be handled concurrently, focusing on managing the execution flow efficiently and maintaining responsiveness.
* Use **parallelism** for applications with computationally intensive tasks that can be divided into independent units of work and executed simultaneously across multiple processing units, aiming to achieve faster processing and improved performance.

In some cases, you may need to use a combination of both concurrency and parallelism to meet the requirements of the application effectively.

**Qn. how to Optimize multithreaded code for better performance?**

Optimizing multithreaded code for better performance involves several strategies to ensure efficient use of resources, minimize contention, and maximize parallelism. Here are some tips to optimize multithreaded code:

1. **Minimize Synchronization**:
   * Reduce the use of synchronized blocks or locks whenever possible, as they can introduce contention and limit parallelism.
   * Use non-blocking algorithms, lock-free data structures, or thread-local variables to minimize the need for synchronization.
2. **Fine-Grained Locking**:
   * Use fine-grained locking to reduce the duration of lock acquisition and release, which can help mitigate contention and improve parallelism.
   * Instead of locking entire data structures, lock smaller portions or individual elements to allow concurrent access to unrelated parts of the data structure.
3. **Avoid Thread Contention**:
   * Identify and eliminate hotspots where multiple threads contend for the same resources, such as shared locks, critical sections, or global variables.
   * Use techniques like sharding, partitioning, or data replication to distribute contention across multiple resources and reduce contention points.
4. **Optimize Data Access**:
   * Minimize data sharing between threads to reduce the need for synchronization and contention.
   * Use thread-local variables or immutable data structures whenever possible to eliminate contention and improve cache locality.
5. **Use Thread Pools**:
   * Reuse threads from a thread pool instead of creating new threads for each task, as thread creation and destruction can be expensive.
   * Tune the size of the thread pool based on the characteristics of your workload and the available resources to maximize throughput and minimize overhead.
6. **Optimize Task Distribution**:
   * Use workload partitioning techniques like work stealing or dynamic task scheduling to distribute tasks evenly among threads and avoid load imbalance.
   * Balance the workload across threads to ensure that no thread is overloaded while others are idle, maximizing overall throughput.
7. **Cache Optimization**:
   * Minimize cache contention by optimizing data access patterns and reducing cache invalidations.
   * Align data structures and access patterns with the underlying hardware architecture to leverage cache locality and improve performance.
8. **Profile and Benchmark**:
   * Use profiling tools to identify performance bottlenecks and hotspots in your multithreaded code.
   * Benchmark different optimizations and configurations to measure their impact on performance and identify the most effective strategies.
9. **Optimize I/O Operations**:
   * Use asynchronous I/O operations or non-blocking I/O libraries to avoid blocking threads and improve concurrency.
   * Batch and coalesce I/O requests to reduce overhead and improve throughput when performing multiple I/O operations.
10. **Consider Hardware and Platform Characteristics**:
    * Take into account the characteristics of the underlying hardware (e.g., number of CPU cores, memory architecture) and the platform (e.g., JVM parameters, operating system settings) when optimizing multithreaded code.

**Qn. Tools and techniques for identifying and resolving concurrency issues.?**

Identifying and resolving concurrency issues can be challenging, but there are several tools and techniques available to help diagnose and address these issues effectively:

1. **Static Analysis Tools**:
   * Use static analysis tools like FindBugs, PMD, and SpotBugs to identify potential concurrency issues such as data races, deadlock-prone code, and improper synchronization.
   * These tools analyze the code statically and provide warnings or suggestions to address potential concurrency problems.
2. **Dynamic Analysis Tools**:
   * Utilize dynamic analysis tools like Thread Dump Analyzers (e.g., jstack, VisualVM) to capture and analyze thread dumps of running applications.
   * Thread dump analysis tools help identify thread contention, deadlock conditions, thread starvation, and other concurrency issues at runtime.
3. **Concurrency Profilers**:
   * Use specialized profilers like YourKit Java Profiler, JProfiler, or VisualVM with Thread and Monitor profiling capabilities to analyze the behavior of concurrent applications.
   * These profilers provide insights into thread activity, synchronization overhead, lock contention, and other performance-related metrics.
4. **Memory Model Analysis Tools**:
   * Employ memory model analysis tools like JCStress or Concuerror to systematically test and verify the correctness of concurrent algorithms and data structures.
   * These tools help identify and diagnose subtle concurrency bugs related to memory visibility, reordering, and other memory model issues.
5. **Concurrency Testing Frameworks**:
   * Use concurrency testing frameworks like JUnit with ConcurrentUnit or TestNG with MultithreadedTC to write and execute concurrent unit tests.
   * Concurrency testing frameworks enable systematic testing of concurrent code under various thread scheduling scenarios, helping uncover race conditions and synchronization errors.
6. **Debugger**:
   * Debug concurrent applications using a debugger with support for multithreaded debugging (e.g., IntelliJ IDEA, Eclipse).
   * Use breakpoints, watchpoints, and thread-specific debugging features to inspect the state of threads, monitor variable values, and identify concurrency issues interactively.
7. **Concurrency Visualization Tools**:
   * Visualize the execution of concurrent programs using tools like Thread Weaver, Concutest, or Concurrency Explorer.
   * These tools provide graphical representations of thread interactions, lock acquisitions, and memory accesses, making it easier to understand and debug complex concurrency issues.
8. **Concurrency Best Practices**:
   * Follow concurrency best practices and design patterns to minimize the likelihood of concurrency issues.
   * Use thread-safe data structures, proper synchronization techniques, and immutable objects to reduce the risk of race conditions, deadlock, and other concurrency problems.
9. **Code Reviews and Pair Programming**:
   * Conduct code reviews and engage in pair programming to collaboratively review and analyze concurrent code.
   * Leverage collective knowledge and experience to identify potential concurrency issues, share best practices, and suggest improvements in code design and implementation.

**Qn. How will you simulate concurrent modification through stream API ?**

Ans: To simulate concurrent modification using the Stream API in Java, you can use parallel streams along with side-effect operations that modify the underlying data structure. Here's an example:

package designpatterns.creational;  
  
import java.util.ArrayList;  
import java.util.List;  
public class ConcurrentModificationExample {  
 public static void main(String[] args) {  
 List<Integer> numbers = new ArrayList<>();  
 for (int i = 0; i < 10; i++) {  
 numbers.add(i);  
 }  
  
 // Simulate concurrent modification using parallel stream  
 numbers.parallelStream().forEach(num -> {  
 if (num % 2 == 0) {  
 numbers.add(num \* 10); // This will throw ConcurrentModificationException  
 }  
 });  
 }  
}

In this example, we have a list of integers numbers, and we are attempting to modify it concurrently using a parallel stream. Inside the stream operation, we are trying to add elements to the list based on a condition (num % 2 == 0). Since the stream operates concurrently, it may attempt to modify the list while other modifications are in progress, leading to a concurrent modification exception (ConcurrentModificationException). This exception occurs because the underlying list is modified while it is being iterated over by the parallel stream, violating the fail-fast behavior of the collections framework.

Top of Form

**Qn. How many thread will open for parallel stream and how parallel stream internally works ?**

The number of threads used by a parallel stream depends on the characteristics of the underlying **ForkJoinPool** that manages the parallel execution. By default, the common pool is used, which is initialized with a number of threads equal to the number of available processors on the system.

When you create a parallel stream, the stream operations are divided into smaller tasks that can be executed concurrently. These tasks are then submitted to the **ForkJoinPool** for execution. The **ForkJoinPool** manages a pool of worker threads, and tasks are distributed among these threads for parallel execution.

Internally, parallel streams use the fork/join framework provided by the **java.util.concurrent** package. This framework divides the tasks into smaller subtasks recursively until they are small enough to be executed independently. These subtasks are then executed in parallel by multiple worker threads in the **ForkJoinPool**. Once the subtasks are completed, the results are combined to produce the final result.

Overall, parallel streams provide a convenient way to leverage multi-core processors and achieve parallelism for stream operations, potentially improving the performance of processing large datasets.

Top of Form

Bottom of Form

Bottom of Form

**Qn. Concurrency Models?**

1. **Shared-Memory Concurrency Model**:
   * **Description**: Multiple threads or processes share a common address space and communicate by reading from and writing to shared memory locations.
   * **Pros**:
     + Simplifies communication and data sharing between threads/processes.
     + Efficient for small-scale concurrency where synchronization overhead is minimal.
   * **Cons**:
     + Prone to data races, race conditions, and other synchronization issues.
     + Difficult to reason about and debug due to shared mutable state.
     + Can suffer from contention and scalability limitations under high concurrency.
2. **Message Passing Concurrency Model**:
   * **Description**: Concurrent entities (e.g., threads, processes) communicate by sending and receiving messages through predefined channels or mailboxes.
   * **Pros**:
     + Enforces clear separation of concerns and encapsulation of state.
     + Facilitates loosely coupled and scalable systems by decoupling communication from computation.
     + Can be more fault-tolerant and resilient to failures compared to shared-memory models.
   * **Cons**:
     + Requires explicit serialization and deserialization of messages, which may introduce overhead.
     + Can be complex to implement and manage, especially in distributed systems.
     + Message delivery and synchronization overhead may impact performance in certain scenarios.
3. **Actor Model Concurrency Model**:
   * **Description**: Concurrency model based on the concept of actors, which are autonomous entities that communicate exclusively through asynchronous message passing.
   * **Pros**:
     + Provides a natural and intuitive model for concurrency, where each actor encapsulates its state and behavior.
     + Facilitates fine-grained parallelism and scalability by distributing actors across multiple threads or processes.
     + Promotes fault isolation and resilience by isolating state and failure within individual actors.
   * **Cons**:
     + Requires a runtime system or framework to manage actor lifecycle, message delivery, and actor scheduling.
     + Overhead associated with message passing and actor supervision may impact performance in certain scenarios.
     + Can be challenging to reason about and debug due to asynchronous message passing and potential message ordering issues.
4. **Dataflow Concurrency Model**:
   * **Description**: Concurrency model based on the flow of data through computational units (nodes), where nodes execute asynchronously and communicate by exchanging data tokens along directed channels.
   * **Pros**:
     + Simplifies coordination and synchronization by focusing on data dependencies rather than control flow.
     + Enables implicit parallelism and pipelining of computations based on data availability.
     + Supports dynamic task scheduling and load balancing, making it suitable for streaming and data-driven applications.
   * **Cons**:
     + Requires a dataflow execution engine or runtime system to manage task scheduling and data movement.
     + Complexity of managing data dependencies and ensuring correctness in asynchronous execution environments.
     + Limited tooling and adoption compared to more traditional concurrency models.

Each concurrency model has its strengths and weaknesses, and the choice depends on factors such as the nature of the problem domain, performance requirements, scalability goals, and developer familiarity with the model. It's often beneficial to combine multiple concurrency models judiciously to leverage their respective advantages and mitigate their drawbacks in complex systems.

**Qn. How will you simulate concurrent modification through stream API ?**

Ans: To simulate concurrent modification using the Stream API in Java, you can use parallel streams along with side-effect operations that modify the underlying data structure. Here's an example:

package designpatterns.creational;  
  
import java.util.ArrayList;  
import java.util.List;  
public class ConcurrentModificationExample {  
 public static void main(String[] args) {  
 List<Integer> numbers = new ArrayList<>();  
 for (int i = 0; i < 10; i++) {  
 numbers.add(i);  
 }  
  
 // Simulate concurrent modification using parallel stream  
 numbers.parallelStream().forEach(num -> {  
 if (num % 2 == 0) {  
 numbers.add(num \* 10); // This will throw ConcurrentModificationException  
 }  
 });  
 }  
}

In this example, we have a list of integers **numbers**, and we are attempting to modify it concurrently using a parallel stream. Inside the stream operation, we are trying to add elements to the list based on a condition (**num % 2 == 0**). Since the stream operates concurrently, it may attempt to modify the list while other modifications are in progress, leading to a concurrent modification exception (**ConcurrentModificationException**). This exception occurs because the underlying list is modified while it is being iterated over by the parallel stream, violating the fail-fast behavior of the collections framework.

Top of Form

**Qn. How many thread will open for parallel stream and how parallel stream internally works ?**

The number of threads used by a parallel stream depends on the characteristics of the underlying **ForkJoinPool** that manages the parallel execution. By default, the common pool is used, which is initialized with a number of threads equal to the number of available processors on the system.

When you create a parallel stream, the stream operations are divided into smaller tasks that can be executed concurrently. These tasks are then submitted to the **ForkJoinPool** for execution. The **ForkJoinPool** manages a pool of worker threads, and tasks are distributed among these threads for parallel execution.

Internally, parallel streams use the fork/join framework provided by the java.util.concurrent package. This framework divides the tasks into smaller subtasks recursively until they are small enough to be executed independently. These subtasks are then executed in parallel by multiple worker threads in the ForkJoinPool. Once the subtasks are completed, the results are combined to produce the final result.

Overall, parallel streams provide a convenient way to leverage multi-core processors and achieve parallelism for stream operations, potentially improving the performance of processing large datasets.

Bottom of Form

Bottom of Form

===========End of Multithreading=============

**Qn. Please give brief description about hashcode() and equals()?**

In Java, **hashCode()** and **equals()** are methods defined in the **Object** class, which is the superclass of all other classes in Java. These methods are fundamental for object comparison and are commonly used when working with collections like **HashMap**, **HashSet**, **Hashtable**, and others.

1. **hashCode()**:
   * The **hashCode()** method returns an integer value, known as the hash code, for an object.
   * This hash code is typically used by hash-based data structures like **HashMap** to determine the **bucket or index** where an object should be stored or retrieved.
   * It's important that equal objects produce the same hash code, but it's not necessary that unequal objects produce distinct hash codes. However, generating distinct hash codes for unequal objects helps to distribute objects evenly across hash table buckets, improving performance.
   * The general contract for **hashCode()** is:
     + If two objects are equal according to the **equals()** method, then calling **hashCode()** on each of the two objects must produce the same integer result.
     + It's not required that if two objects are unequal according to the **equals()** method, their hash codes must be different. However, unequal objects with different hash codes may improve the performance of hash-based collections.
   * It's recommended to override the **hashCode()** method whenever the **equals()** method is overridden, to ensure consistency.
2. **equals()**:
   * The **equals()** method is used to compare the equality of two objects.
   * It returns **true** if the objects are equal according to some logical criteria, and **false** otherwise.
   * The general contract for **equals()** is:
     + Reflexive: An object must be equal to itself. Hence, **x.equals(x)** must return **true**.
     + Symmetric: If **x.equals(y)** returns **true**, then **y.equals(x)** must also return **true**.
     + Transitive: If **x.equals(y)** and **y.equals(z)** both return **true**, then **x.equals(z)** must also return **true**.
     + Consistent: Multiple invocations of **equals()** on the same objects must consistently return either **true** or **false**, provided that the objects have not changed between invocations.
     + It's also recommended to override the **hashCode()** method whenever **equals()** is overridden, to ensure consistency in hash-based collections.

**Qn. Internal Working of HashMap and HashSet?**

HashMap and HashSet are both implementations of the Set and Map interfaces, respectively, in Java. They use hashing techniques for efficient storage and retrieval of elements. Here's a brief overview of their internal workings:

1. **HashMap**:
   * Internally, HashMap stores key-value pairs in ***an array of linked lists called buckets.***
   * When you put a key-value pair into a HashMap, it calculates the hash code of the key to determine the bucket index where the pair will be stored.
   * If there are no collisions (i.e., multiple keys with the same hash code), the pair is added to the bucket directly.
   * If there is a collision, meaning multiple keys have the same hash code, the elements are stored in a linked list within the bucket.
   * In Java 8 and later versions, when the number of elements in a bucket exceeds a certain threshold, the linked list is converted into a balanced tree for improved performance.
   * HashMap uses the hashCode() method of keys to calculate their hash codes.
2. **HashSet**:
   * HashSet internally uses a HashMap to store its elements.
   * When you add an element to a HashSet, it is inserted into the underlying HashMap as a key, with a fixed dummy value.
   * Since HashMap does not allow duplicate keys, HashSet effectively ensures that it contains only unique elements.
   * The add() method of HashSet delegates to the put() method of the underlying HashMap, with the element as the key and a dummy value as the associated value.
   * HashSet uses the hashCode() method of elements to calculate their hash codes.

In summary, HashMap and HashSet both leverage hashing techniques to achieve constant-time performance for insertion, deletion, and retrieval operations. They use the hashCode() method of elements to determine their storage locations and efficiently handle collisions when multiple elements have the same hash code.

**Qn. what is database sharding?**

Database sharding is a technique used to horizontally partition a large database into smaller, more manageable parts called shards. Each shard is stored on a separate physical or logical database server instance, allowing the database workload to be distributed across multiple machines. This approach helps to improve performance, scalability, and availability by reducing the load on individual database servers and allowing them to handle a subset of the overall data.

Key aspects of database sharding include:

1. **Horizontal Partitioning**: Sharding involves splitting the database horizontally, meaning that rows of data are divided into separate shards based on a specific criterion, such as a range of values, a hash function, or another logical partitioning scheme.
2. **Data Distribution**: Each shard contains a subset of the data, and different shards may be located on different physical servers or clusters. This distribution ensures that the database workload is spread across multiple nodes, reducing the burden on any single server.
3. **Partitioning Strategy**: Choosing an appropriate partitioning strategy is crucial for effective sharding. Common strategies include range-based partitioning (e.g., dividing data based on a key range), hash-based partitioning (e.g., using a hash function to determine shard placement), or composite partitioning (combining multiple partitioning methods).
4. **Data Consistency and Distribution**: Sharding introduces challenges related to data consistency and distribution. Distributed transactions, joins across shards, and maintaining referential integrity require careful consideration and may involve additional complexity.
5. **Shard Management**: Sharding requires mechanisms for managing shard placement, rebalancing data across shards, and handling shard failures or additions. Tools and techniques for shard management help automate these tasks and ensure the stability and availability of the distributed database system.

Database sharding is commonly used in large-scale web applications, cloud-based systems, and distributed data processing platforms where horizontal scalability and high availability are essential. By distributing the database workload across multiple shards, organizations can accommodate growing data volumes, handle increased user traffic, and achieve better performance and resilience. However, implementing and managing sharded databases requires careful planning, monitoring, and maintenance to ensure optimal performance and data consistency.

1. **SQL Databases (RDBMS)**:
   * Sharding is less common in traditional SQL databases due to their strong consistency guarantees and support for ACID transactions, which can make horizontal scaling more complex.
   * However, some SQL databases support sharding techniques for scaling out read-heavy workloads or distributing data across multiple nodes. Examples include MySQL Cluster, PostgreSQL with Citus extension, and SQL Server with distributed availability groups.
2. **NoSQL Databases**:
   * NoSQL databases are designed with horizontal scalability in mind and are more commonly associated with sharding.
   * Many NoSQL databases, such as MongoDB, Cassandra, and Couchbase, natively support sharding to distribute data across clusters of servers. Sharding in NoSQL databases enables linear scalability and improved performance for write-heavy and read-heavy workloads.
   * NoSQL databases often provide built-in mechanisms for automatic or manual sharding, allowing developers to define sharding keys and configure how data is distributed across shards.

In summary, while sharding can be implemented in both SQL (RDBMS) and NoSQL databases, it is more prevalent and integral to the architecture of many NoSQL databases. NoSQL databases are typically designed for horizontal scalability and distributed systems, making them well-suited for sharding to handle large volumes of data and high transaction rates.

**Qn. Difference between (RDMS) SQL and No SQL ?**

Relational Database Management Systems (RDBMS) and NoSQL databases are two types of database management systems that differ in their data models, scalability, and query languages. Here's a comparison between SQL (RDBMS) and NoSQL databases:

1. **Data Model**:
   * **SQL (RDBMS)**: RDBMS uses a tabular structure with rows and columns to organize data. It enforces a predefined schema where data must conform to a specific structure, typically defined by tables, columns, and relationships between tables.
   * **NoSQL**: NoSQL databases support various data models, including document-oriented, key-value, wide-column, and graph databases. They offer more flexibility in schema design, allowing for dynamic or schema-less data.
2. **Scalability**:
   * **SQL (RDBMS)**: Traditional RDBMS systems may face challenges with horizontal scalability, especially for large-scale distributed systems. Scaling typically involves vertical scaling by upgrading hardware or using replication for read-heavy workloads.
   * **NoSQL**: NoSQL databases are designed for horizontal scalability, making it easier to distribute data across multiple nodes or servers. They can handle large volumes of data and high transaction rates by scaling out horizontally.
3. **Query Language**:
   * **SQL (RDBMS)**: RDBMS systems use SQL (Structured Query Language) for data manipulation and querying. SQL provides a standardized syntax for performing CRUD operations (Create, Read, Update, Delete) and complex queries involving joins, aggregates, and transactions.
   * **NoSQL**: NoSQL databases may use various query languages or APIs, depending on the data model. For example, document-oriented databases like MongoDB use query languages such as MongoDB Query Language (MQL), while key-value stores like Redis provide APIs for data access and manipulation.
4. **Consistency and Transactions**:
   * **SQL (RDBMS)**: RDBMS systems typically offer strong consistency guarantees and support ACID transactions (Atomicity, Consistency, Isolation, Durability). Transactions ensure that database operations maintain data integrity and adhere to the defined constraints.
   * **NoSQL**: NoSQL databases vary in their consistency models, with some prioritizing availability and partition tolerance over strong consistency. Many NoSQL databases offer eventual consistency, where data consistency is eventually achieved after updates propagate across distributed nodes. Some NoSQL databases provide tunable consistency levels to balance consistency and performance based on application requirements.
5. **Use Cases**:
   * **SQL (RDBMS)**: RDBMS systems are well-suited for applications with structured data, complex relationships, and ACID transaction requirements. They are commonly used in traditional enterprise applications, financial systems, and applications requiring strict data integrity.
   * **NoSQL**: NoSQL databases are preferred for applications with unstructured or semi-structured data, high scalability requirements, and flexible schemas. They are commonly used in web applications, content management systems, real-time analytics, and IoT (Internet of Things) applications.

In summary, the choice between SQL (RDBMS) and NoSQL databases depends on factors such as data structure, scalability requirements, consistency needs, and application use cases. Each type of database system has its strengths and weaknesses, and the selection should be based on the specific requirements of the application.

**Qn. Difference between String and StringBuilder?Bottom of Form**

The main difference between **String** and **StringBuilder** lies in their mutability and performance characteristics:

1. **Mutability**:
   * **String**: Strings in Java are immutable, meaning once a String object is created, its value cannot be changed. Any operation that seems to modify a String actually creates a new String object with the modified value.
   * **StringBuilder**: **StringBuilder** is mutable, meaning its contents can be modified after creation. Operations like append(), insert(), delete(), etc., modify the contents of the StringBuilder instance directly without creating new objects.
2. **Performance**:
   * **String**: Due to immutability, every operation that appears to modify a String results in the creation of a new String object, which can lead to memory overhead, especially in scenarios with frequent string manipulations.
   * **StringBuilder**: StringBuilder offers better performance for string manipulations compared to String. Since it's mutable, it allows for efficient in-place modifications of strings without the need for creating new objects.

Usage scenarios:

* Use **String** when dealing with fixed strings or when immutability is desired for thread safety.
* Use **StringBuilder** when performing extensive string manipulations, concatenations, or modifications, especially in performance-critical code, as it offers better efficiency compared to repeatedly creating new **String** objects.

**Qn. What are the time/space complexities for collection classes and name their internal underline data structure?**

Here are the time and space complexities for some common collection classes in Java, along with their underlying data structures:

1. **ArrayList**:
   * Time complexities:
     + Access (get): O(1)
     + Search (contains, indexOf): O(n)
     + Insertion (add): O(1) amortized, O(n) worst-case when resizing is required
     + Deletion (remove): O(n) in worst-case, O(1) if removing from the end
   * Space complexity: O(n)
   * Underlying data structure: Array
2. **LinkedList**:
   * Time complexities:
     + Access (get): O(n)
     + Search (contains, indexOf): O(n)
     + Insertion (add): O(1)
     + Deletion (remove): O(1) for the first or last element, O(n) otherwise
   * Space complexity: O(n)
   * Underlying data structure: Doubly linked list
3. **HashSet**:
   * Time complexities:
     + Add: O(1) average-case, O(n) worst-case
     + Remove: O(1) average-case, O(n) worst-case
     + Contains: O(1) average-case, O(n) worst-case
   * Space complexity: O(n)
   * Underlying data structure: Hash table (based on HashMap)
4. **TreeSet**:
   * Time complexities:
     + Add: O(log n)
     + Remove: O(log n)
     + Contains: O(log n)
   * Space complexity: O(n)
   * Underlying data structure: Red-black tree
5. **HashMap**:
   * Time complexities:
     + Put: O(1) average-case, O(n) worst-case
     + Get: O(1) average-case, O(n) worst-case
     + Remove: O(1) average-case, O(n) worst-case
   * Space complexity: O(n)
   * Underlying data structure: Array of linked lists or balanced trees (for JDK 8 and later)
6. **TreeMap**:
   * Time complexities:
     + Put: O(log n)
     + Get: O(log n)
     + Remove: O(log n)
   * Space complexity: O(n)
   * Underlying data structure: Red-black tree

**Qn. Name concurrent collection class..**

\* ConcurrentHashMap

\* ConcurrentLinkedQueue

\* CopyOnWriteArrayList

**Qn. If any data base table has name, id column then please write a query to find even rows?**

SELECT \* FROM your\_table

WHERE id % 2 = 0;

**Qn. Name of algorithm use by Arrays.sort(..) and Collections.sort(..)?**

The algorithm used by **Arrays.sort(..)** and **Collections.sort(..)** in Java is called "Timsort." Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort.

Timsort provides excellent performance on many kinds of real-world data and has a guaranteed worst-case time complexity of O(n log n). It also performs well on partially sorted data or data with small runs.

Timsort works by dividing the array into small chunks, sorting them individually using insertion sort, and then merging them together using a modified form of merge sort.

**Qn. In Java, primitive types and non-primitive types (also known as reference types) differ in how they are stored in memory:**

1. **Primitive Types:**
   * Primitive types represent simple data types and are stored directly on the stack memory. [However, when primitive types are part of an object (i.e., as instance variables), they are stored indirectly in the heap memory.
   * They hold their values directly, without needing to reference another memory location.
   * Examples include **int**, **double**, **char**, **boolean**, etc.
   * Memory allocation and deallocation are straightforward since they are stored directly on the stack.
2. **Non-primitive Types (Reference Types):**
   * Non-primitive types, also known as reference types, store references (memory addresses) to objects in the heap memory.
   * They represent complex data types that are created using classes or interfaces.
   * Examples include arrays, objects, and strings.
   * Memory for the actual objects is allocated on the heap, and the reference to the object is stored on the stack.
   * Accessing non-primitive types involves an extra level of indirection through the reference, which can impact performance slightly compared to primitives.

**Qn. What are some things to consider when optimizing code:**

* Data types: The right data types are key to optimizing code performance.
* Code complexity: Reduce code complexity.
* Data structures: Use data structures similar to sorting and string concatenation.
* Algorithms: Use the most efficient algorithms.
* Code style: Write clean and maintainable code.
* Functions: Break your code down into small, reusable functions.
* Performance: Write code that runs fast, uses minimal resources, and is easy to understand and modify.

# **Qn. What is ClassLoader in Java?**

The **Java ClassLoader** is a part of the [**Java Runtime Environment**](https://www.geeksforgeeks.org/differences-jdk-jre-jvm/) that dynamically loads Java classes into the [**Java Virtual Machine**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). The Java run time system does not need to know about files and file systems because of classloaders. [Java classes](https://www.geeksforgeeks.org/classes-objects-java/) aren’t loaded into memory all at once, but when required by an application. At this point, the **Java ClassLoader** is called by the **JRE** and these ClassLoaders load classes into memory dynamically.

Ther e are three built-in class loaders in Java:

1. **Bootstrap Class Loader:** it is responsible for loading core Java classes located in the JDK's ***rt.jar*** and other system classes. It is implemented in native code.
2. **Extension Class Loader:** It is responsible for loading classes from the extensions directory (usually ***lib/ext***) of the JRE. It is a child of the bootstrap class loader.
3. **System Class Loader:** Also known as the application class loader, it loads classes from the ***classpath***, including user-defined classes and third-party libraries. It is a child of the extension class loader.

Java also provides a mechanism to create custom class loaders for loading classes from non-standard sources, such as network locations or databases. Custom class loaders must subclass the **java.lang.ClassLoader** class and override the **findClass** method to specify how classes are loaded.

Class loaders follow the delegation model, where a class loader first delegates the class loading request to its parent class loader before attempting to load the class itself. This hierarchical structure ensures that classes are loaded in a controlled and consistent manner, allowing for class reuse and preventing class duplication.

### **Qn. Delegation Model in Java Class loader?**

Class loaders follow the delegation model, where **on request to find a class or resource, a***ClassLoader***instance will delegate the search of the class or resource to the parent class loader**.

Let’s say we have a request to load an application class into the JVM. The system class loader first delegates the loading of that class to its parent extension class loader, which in turn delegates it to the bootstrap class loader.

Only if the bootstrap and then the extension class loader are unsuccessful in loading the class, the system class loader tries to load the class itself.

**Qn. how we can use multithreading in data processing Application?**

Multithreading can be very beneficial in data processing applications as it allows for parallel execution of tasks, leading to improved performance and throughput. Here's how you can use multithreading in a data processing application:

1. **Divide and Conquer:** Break down your data processing task into smaller, independent units of work. Each unit of work should ideally be capable of running independently of the others.
2. **Thread Pool:** Use a thread pool to manage multiple threads efficiently. Instead of creating new threads for each task, you can reuse existing threads from the pool, reducing overhead.
3. **Task Parallelism:** Assign each unit of work to a separate thread or a subset of threads. This allows multiple tasks to be executed concurrently, taking advantage of multicore processors and improving overall throughput.
4. **Synchronization:** Ensure proper synchronization mechanisms are in place if multiple threads need to access shared resources or modify shared data structures. This prevents data corruption and ensures thread safety.
5. **Producer-Consumer Pattern:** If your data processing involves producing and consuming data, consider using the producer-consumer pattern. One or more producer threads generate data, which is then consumed by one or more consumer threads. This pattern can help balance the load and optimize resource utilization.
6. **Parallel Stream Processing:** If you're working with collections or streams of data in Java, consider using parallel streams. Java's Stream API provides support for parallel execution of operations on streams, making it easy to leverage multicore processors for data processing tasks.

**Qn. What is Reflection in Java?**

Reflection in Java allows programs to inspect and manipulate their own structure and behavior at runtime. It provides a way to analyze classes, interfaces, fields, methods, and constructors, as well as to invoke methods, access fields, and create new instances dynamically.

**When to use reflection**:

1. **Dynamic Loading**: When the types of objects need to be determined dynamically at runtime, reflection enables loading and instantiation of classes, interfaces, and objects. For example, frameworks like Spring and Hibernate use reflection to instantiate beans and entities based on configuration.
2. **Introspection**: Reflection is useful for analyzing and discovering information about classes and their members. It's commonly used in libraries, frameworks, and tools that need to work with arbitrary types. For example, libraries like Jackson use reflection to serialize and deserialize objects based on their structure.
3. **Debugging and Testing**: Reflection allows for dynamic inspection and modification of objects during debugging or testing, providing flexibility in testing scenarios. For example, testing frameworks like Mockito use reflection to mock objects and verify method invocations.

**Disadvantages of reflection**:

1. **Performance Overhead**: Reflection operations often incur a performance overhead compared to direct method invocation or field access, due to runtime checks and method lookups. For performance-critical code, using reflection extensively may not be ideal.
2. **Encapsulation Violation**: Reflection can bypass access controls and access private methods and fields, leading to violations of encapsulation principles. This can make the code less maintainable and harder to understand.
3. **Code Complexity and Maintenance**: Code relying heavily on reflection can be harder to understand, debug, and maintain due to its dynamic nature. It may lack compile-time type safety, making it prone to errors and more challenging to maintain.

**Example**: Consider a scenario where you want to load and instantiate a class dynamically based on user input:

public class ReflectionExample {  
 public static void main(String[] args) throws ClassNotFoundException, IllegalAccessException, InstantiationException {  
 // User input specifying the class name  
 String className = "com.example.MyClass";  
  
 // Dynamically load the class  
 Class<?> clazz = Class.*forName*(className);  
  
 // Instantiate the class  
 Object instance = clazz.newInstance();  
  
 // Cast the instance to the appropriate type  
 MyClass myObject = (MyClass) instance;  
  
 // Use the instantiated object  
 myObject.doSomething();  
 }  
}

In this example, reflection is used to dynamically load and instantiate the **MyClass** based on the class name provided by the user at runtime. While this provides flexibility, it also comes with the drawbacks mentioned above, such as performance overhead and potential encapsulation violations. Therefore, reflection should be used judiciously and with caution.

**Qn. how to create unmodifiable list in java and what is it's use cases?**

In Java, you can create an unmodifiable list using the **Collections.unmodifiableList()** method. This method returns a view of the specified list that disallows any modifications to the underlying list. Any attempt to modify the returned list, such as adding, removing, or updating elements, will result in an **UnsupportedOperationException**.

Here's how you can create an unmodifiable list:

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
  
public class Main {  
 public static void main(String[] args) {  
 List<String> mutableList = new ArrayList<>();  
 mutableList.add("Apple");  
 mutableList.add("Banana");  
 mutableList.add("Orange");  
  
 List<String> unmodifiableList = Collections.*unmodifiableList*(mutableList);  
  
 // Attempting to modify the unmodifiable list will throw an UnsupportedOperationException  
 // unmodifiableList.add("Grapes"); // This will throw an exception  
  
 // You can still access elements in the unmodifiable list  
 System.*out*.println(unmodifiableList.get(0)); // Output: Apple  
 }  
}

Use cases for unmodifiable lists include:

1. **Immutable API**: You may want to expose a read-only view of a list from a method or class API to prevent callers from modifying the list.
2. **Thread Safety**: Unmodifiable lists are inherently thread-safe for reading, as they cannot be modified after creation. This can simplify concurrent programming by eliminating the need for explicit synchronization when accessing shared data structures.
3. **Protecting Data**: You can use unmodifiable lists to protect sensitive data or configuration settings from unintended modification.
4. **Functional Programming**: In functional programming paradigms, immutability is often preferred. Unmodifiable lists can be used to enforce immutability and facilitate functional programming practices.

**Qn. what is immutable in java?**

In Java, immutability refers to the characteristic of objects whose state cannot be modified after they are created. Once an immutable object is created, its state remains constant throughout its lifetime.

Immutable objects are inherently thread-safe by design. Since their state cannot change, multiple threads can access them concurrently without the risk of race conditions or unexpected behavior.

Here are some common types and data structures in Java that are immutable:

1. **String**: Strings in Java are immutable. Once a String object is created, its value cannot be changed.
2. **Wrapper Classes**: Wrapper classes such as **Integer**, **Double**, **Boolean**, etc., are immutable. The value of these wrapper objects cannot be modified after they are created.
3. **Collections**: Java provides immutable implementations of some collection types through the **Collections** utility class. For example:
   * **Collections.unmodifiableList()**
   * **Collections.unmodifiableSet()**
   * **Collections.unmodifiableMap()**

These methods return wrapper objects that prevent modification of the underlying collection.

1. **Immutable Classes**: You can create your own immutable classes by ensuring that the class:
   * Declares all fields as **final**.
   * Does not provide setter methods for modifying its state.
   * Ensures that any mutable fields are not exposed or are defensively copied.

Immutable data structures are suitable for caching, configuration, share DS for multithreaded env etc.