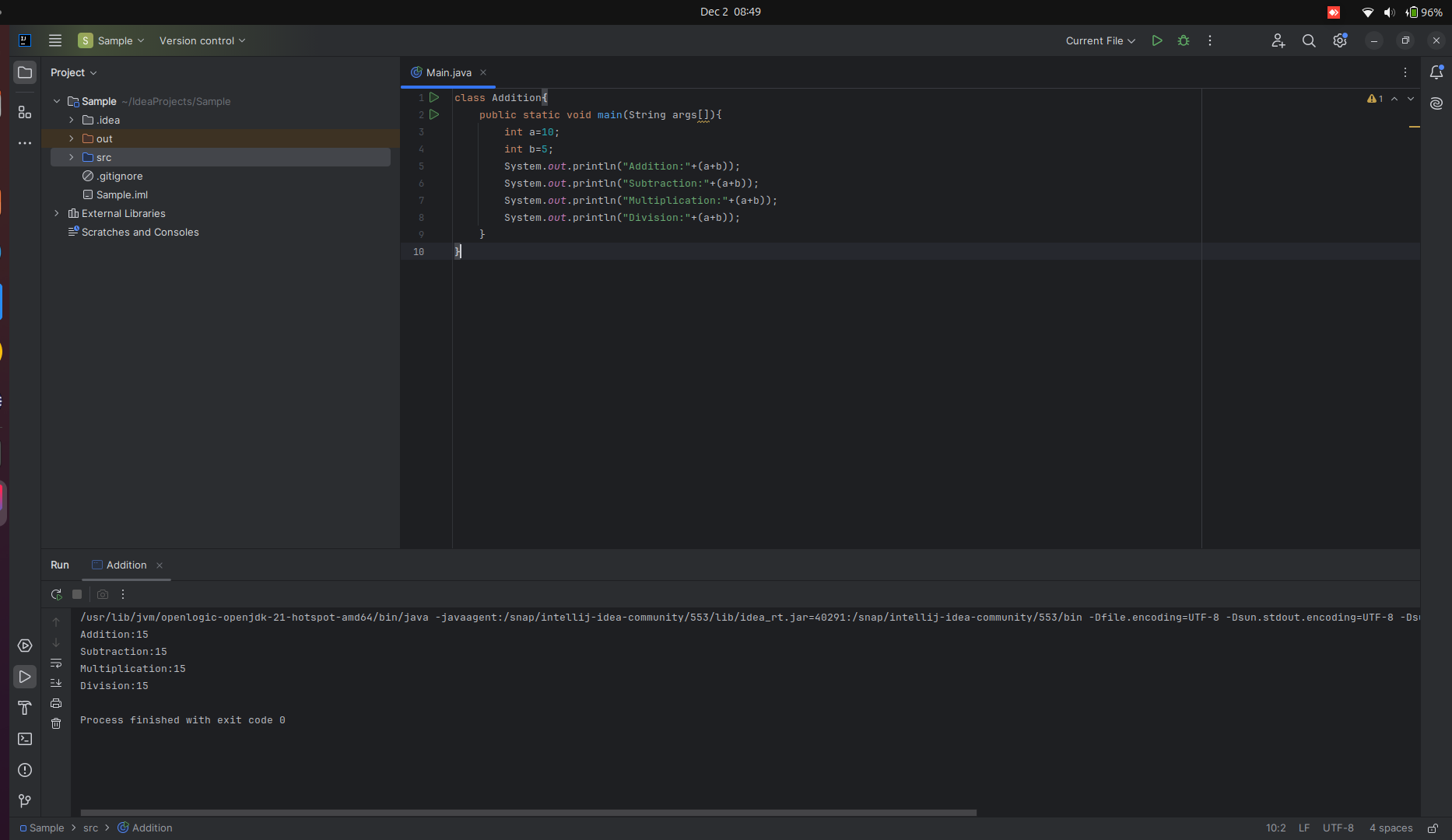
Java:

Sample code



Features of java:

1. [Simple](https://www.javatpoint.com/features-of-java#Simple) - Easy to understand
2. [Object-Oriented](https://www.javatpoint.com/features-of-java#Object-Oriented) - combination of different types of objects that incorporate both data and behavior.
3. [Portable](https://www.javatpoint.com/features-of-java#Portable) - carry the Java bytecode to any platform
4. [Platform independent](https://www.javatpoint.com/features-of-java#Platform-independent) - write once, run anywhere
5. [Secured](https://www.javatpoint.com/features-of-java#Secured) - No Explicit Pointers and Runs in a Virtual Machine

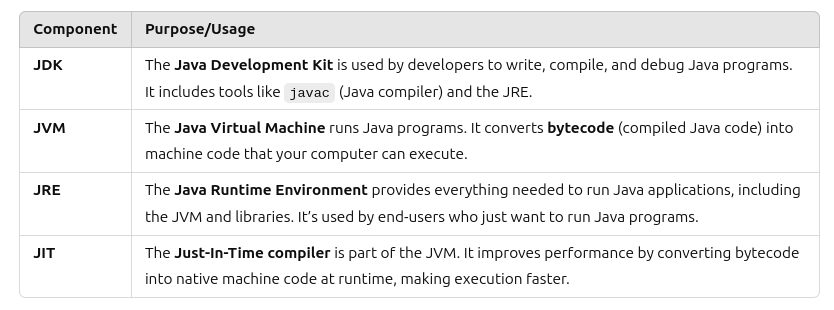
Java provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

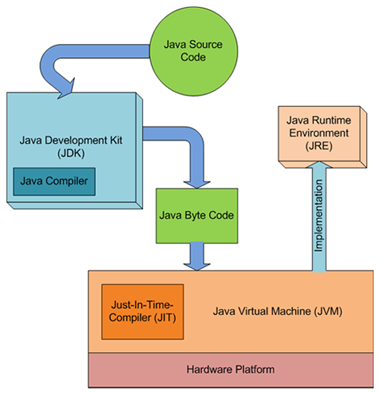
1. [Robust](https://www.javatpoint.com/features-of-java#Robust) - strong memory management.
2. [Architecture neutral](https://www.javatpoint.com/features-of-java#Architecture-neutral) - size of primitive types is fixed.
3. [Interpreted](https://www.javatpoint.com/features-of-java#Interpreted)
4. [High Performance](https://www.javatpoint.com/features-of-java#High-Performance) - Bytecode is Close to Native Code and Using JIT compiler

Java is faster than many interpreted languages because of its bytecode and JIT compilation.

However, it’s a bit slower than fully compiled languages like C++ since they directly produce machine code before execution.

1. [Multithreaded](https://www.javatpoint.com/features-of-java#Multithreaded) -A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area.
2. [Distributed](https://www.javatpoint.com/features-of-java#Distributed) - RMI (Remote Method Invocation)
3. This feature of Java makes us able to access files by calling the methods from any machine on the internet.
4. [Dynamic](https://www.javatpoint.com/features-of-java#Dynamic) - Java is **dynamic** because it can adapt to changing environments by linking code dynamically at runtime. It also supports features like reflection, which allows inspecting and modifying a program's structure during execution.

**Usage:** 



**Example of How They Work Together:**

1. The developer uses the JDK to write and compile a Java program into bytecode.
2. The end-user runs the program using the JRE, which includes the JVM.
3. The JVM executes the bytecode, and the JIT compiler optimizes it for better performance.

### **Summary:**

* JDK: For developers (write and compile Java).

Compiler: compile the code and into bytecode

Debugger:debugs the bytecode

* JVM: Runs Java code.

Load Bytecode: Reads the .class file (Java bytecode).

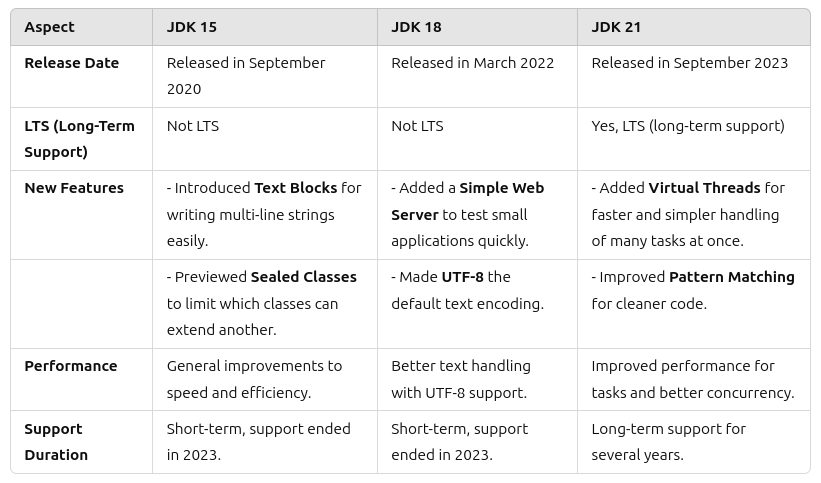
Interpret Bytecode: Converts bytecode into ma-chine code specific to the host system.

Memory Management: Manages memory allocation (Heap and Stack) and performs garbage collection to clean up unused objects.

* JRE: For running Java apps (includes JVM).
* JIT: Speeds up JVM performance.

**JDK:**

The JDK (Java Development Kit) version represents a specific release of Java and determines the features, libraries, and tools available to developers. Each version introduces updates, bug fixes, performance improvements, and new capabilities for building and running Java applications.



### **Notes:**

1. JDK 15: Focused on improving Java language flexibility with features like records and sealed classes (preview).
2. JDK 18: Introduced practical tools like the Simple Web Server and UTF-8 as the default charset.
3. JDK 21: A major release with long-term support, featuring significant enhancements like virtual threads, making Java more modern and developer-friendly.

Here’s a simple explanation of each feature:

### **1. Text Blocks**

* A way to write multi-line strings easily in Java.

Example: Instead of adding quotes and line breaks manually, you can use triple quotes:  
 String text = """

This is a multi-line

string in Java.

""";

### **2. Simple Web Server**

* A small, built-in web server for testing or running small applications.
* Developers can quickly serve files or test HTTP requests without needing external tools.

### **3. Virtual Threads**

* Lightweight threads for running many tasks at once efficiently.
* Makes handling tasks like user requests or background processing faster and simpler.

### **4. Sealed Classes (Preview Feature)**

* Lets you control which classes can extend a specific class.

Example:  
 sealed class Shape permits Circle, Square {}

class Circle extends Shape {}

class Square extends Shape {}

* Only **Circle** and **Square** can extend **Shape**.

### **5. UTF-8 as Default Charset**

* Java now uses **UTF-8** as the default encoding for text.
* This means text files, strings, and data can handle multiple languages and special characters without issues.

### **6. Pattern Matching**

* A simpler way to check an object's type and use it immediately.

Example:  
 if (obj instanceof String str) {

System.out.println(str.toUpperCase());

}

* Here, Java checks if obj is a String and casts it to str in one step.

### **Summary:**

These features make Java easier to use, faster, and better at handling modern programming needs like text processing, web development, and multitasking.

**Note:**

**Why is java robust?**

1. **Garbage Collection**: Java automatically removes unused objects from memory to free up space, preventing memory leaks.
2. **Exception Handling**: Java uses try-catch-finally blocks to handle runtime errors gracefully, ensuring the program doesn't crash unexpectedly.
3. **Memory Allocation**: Java manages memory using **heap** (for objects) and **stack** (for method calls and local variables) automatically.

**Questions:**

**1.Emoji**

In **JDK 21**, **emojis** and **garbage collection** are two distinct concepts that play important roles in modern Java programming. While emojis are Unicode characters represented by code points, garbage collection is a memory management process where Java automatically frees up memory by removing objects that are no longer reachable or in use.

Emojis are special characters that use Unicode for representation. Many emojis are part of the **Supplementary Multilingual Plane (SMP)**, which is a range of Unicode codes beyond the basic characters.

Since Java's char type can only store a single 16-bit code unit (Basic Multilingual Plane), emojis require **two char values** (called a **surrogate pair**) to represent a single emoji.

### **Simple Explanation:**

1. **Basic characters** (like A, B, 1, @) use one char.
2. **Emojis** need two char values because their Unicode values are large.

**Example:**

String emoji = "\uD83D\uDE0A"; // Two char values together represent 😊

System.out.println(emoji); // Output: 😊

In this example:

* \uD83D and \uDE0A are the two char values forming the single emoji 😊.

### **1. Emojis in Java (Unicode Code Points)**

**Unicode characters** are represented using escape sequences like \u followed by four hexadecimal digits.

Emojis are part of the Unicode standard, and in Java, they are represented using **Unicode code points**. Starting from JDK 8 and later, Java supports Unicode characters, including emojis.

#### **Handling Emojis:**

Emojis are typically represented as **Unicode code points** and can be manipulated with the Character class or using **String.codePoints()** to work with emojis.

#### **Example Program: Working with Emojis**

public class EmojiExample {

public static void main(String[] args) {

// A string containing an emoji

String text = "Java is fun! 😊";

// Loop through the Unicode code points in the string

text.codePoints().forEach(cp -> {

System.out.println("Code Point: " + cp + " (Character: " + Character.toString(cp) + ")");

});

// Displaying an emoji directly

}

}

**Explanation**:

* **text.codePoints()**: Converts the string into a stream of Unicode code points, which includes any emoji characters.
* **Character.toString(cp)**: Converts the Unicode code point (cp) back to the corresponding character.
* This program loops through each character (including emojis) in the string, displaying its code point and character.

**Output**:

Code Point: 74 (Character: J)

Code Point: 97 (Character: a)

Code Point: 118 (Character: v)

Code Point: 97 (Character: a)

Code Point: 32 (Character: )

Code Point: 105 (Character: i)

Code Point: 115 (Character: s)

Code Point: 32 (Character: )

Code Point: 102 (Character: f)

Code Point: 117 (Character: u)

Code Point: 110 (Character: n)

Code Point: 33 (Character: !)

Code Point: 128522 (Character: 😊)

-----------------------------------------------------------------------------------------------------------------

**2.Garbage collection (ZGC)**

Yes, with **ZGC (Z Garbage Collector)** in JDK 21, you can reclaim unused memory and make it available for future use, but there are specific details to keep in mind:

1. **Reclaiming Unused Memory**: ZGC performs **concurrent garbage collection**, meaning it can reclaim memory from objects that are no longer reachable (i.e., those that are garbage). This happens incrementally while your application is still running, without causing long pause times.
2. **Memory Return to the OS**: While ZGC is good at cleaning up unused objects in the heap, it does not directly return memory to the operating system after reclaiming it. Instead, it **shrinks the heap internally** by compacting and reusing space within the JVM’s heap. However, ZGC **won't automatically release memory** back to the OS unless explicitly configured through specific JVM flags for heap resizing.

**JVM flags are command-line options that control the behavior of the Java Virtual Machine (JVM) during runtime, such as memory management, garbage collection, and performance tuning.**

1. **Heap Shrinking**: In JDK 21, ZGC can resize the heap dynamically based on the memory usage of your application. For example, the **-XX:+ZUncommit** flag can be used to allow the garbage collector to release unused memory back to the OS. This enables more effective memory management in long-running applications.

Reclaiming unused memory in Java, particularly with **ZGC (Z Garbage Collector)**, involves a series of steps that allow the JVM to efficiently manage and free memory from objects that are no longer in use. Below is an explanation of the process:

### **1. Marking Phase (Identifying Reachable Objects):**

### The Marking Phase is the first step in garbage collection. During this step, the garbage collector looks for all objects in memory that your program can still use. It starts from the "roots," like:

### Variables in use,

### Active threads,

### Static fields in classes.

### If an object can still be accessed (directly or indirectly) by your program, it’s marked as alive and won't be deleted.

### Objects that can’t be accessed anymore are marked as unreachable. These unreachable objects are no longer needed and can be safely removed to free up memory.

### **2. Concurrent Marking:**

* ZGC uses **concurrent marking** to find all reachable objects while the application continues running. This avoids long stop-the-world pauses that would interrupt the application's execution.
* As ZGC works incrementally in the background, it marks objects without stopping the application threads. This makes ZGC suitable for applications that require low-latency operations.

### **3. Relocation Phase (Compacting the Heap):**

* After marking unreachable objects, ZGC begins the **relocation phase**, where it moves objects around in the heap to compact memory.
* **Live objects** (reachable ones) are moved to a new area within the heap, leaving gaps in the old space.
* ZGC uses **region-based allocation** where the heap is divided into smaller regions. It can move objects between these regions, efficiently utilizing available space.

### **4. Freeing Memory:**

* The **unreachable objects** that were identified during the marking phase are now ready to be **removed** from the heap. These objects are cleaned up, and the memory they occupied is marked as free.
* The free space is then reclaimed and made available for new objects.
* ZGC does not perform a "stop-the-world" phase during cleanup. It continues to work concurrently with application threads, ensuring minimal pause times.

### **5. Heap Shrinking:**

* **Heap shrinking** is an optional process, where the heap's size is adjusted to reclaim memory that is no longer needed. ZGC can resize the heap dynamically based on the application's memory usage, which allows for efficient memory utilization over time.
* If memory usage decreases (e.g., the application no longer needs as much memory), ZGC can shrink the heap to release unused space. The JVM can also release memory back to the operating system when configured to do so.

### **6. Releasing Memory to the OS:**

* ZGC does not immediately return unused memory back to the **operating system**. Instead, it **shrinks the heap internally**. However, with certain flags (such as -XX:+ZUncommit), the garbage collector can release unused memory back to the OS.
* This dynamic resizing helps avoid excessive memory usage and ensures that the JVM adapts to changes in application memory requirements.

### **Summary of the Process:**

1. **Marking**: Identifying all reachable objects.
2. **Concurrent Marking**: Marking happens without stopping application threads.
3. **Relocation**: Moving live objects to compact memory.
4. **Freeing Memory**: Cleaning up unreachable objects.
5. **Heap Shrinking**: Dynamically resizing the heap based on memory usage.
6. **Releasing Memory to OS**: (Optional) Freeing unused memory back to the operating system with certain flags.

-----------------------------------------------------------------------------------------------------------------

**3.Why is java secured?**

Java is secure due to the following reasons:

* Java programs run inside a virtual machine which is known as a sandbox.
* Java does not support explicit pointers.
* Byte-code verifier checks the code fragments for illegal code that can violate access rights to objects.
* It provides a java.security package that implements explicit security.
* It provides library level safety.
* A run-time security check takes place when we load new code.

[Java](https://www.javatpoint.com/java-tutorial) provides some other features that make Java more secure.

* JVM
* Security API's
* Security Manager
* Auto Memory Management
* No Concept of Pointers
* Compile-time Checking
* Cryptographic Security
* Java Sandbox
* Exception Handling
* ClassLoader

JVM

[JVM](https://www.javatpoint.com/jvm-java-virtual-machine) plays a vital role to provide security. It verifies the byte-code. The JVM provides guarantees that there is no unsafe operation going to execute. It also helps to diminish the possibilities of the programmers who suffer from memory safety flaws.

Security API's

Java class libraries provide several API that leads to security. These APIs contain cryptographic algorithms and authentication protocols that lead to secure communication.

* No explicit pointer.
* Java Programs run inside a virtual machine sandbox.
* **Classloader:** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
* **Bytecode Verifier:** checks the code fragments for illegal code that can violate access rights to objects.
* **Security Manager:** determines what resources a class can access such as reading and writing to the local disk.

Memory management

Java automatically manages memory which is known as garbage collection. The JVM manages memory itself. The programmers are free from memory management. Hence, there is no chance of fault in memory management.

Compile-time checking

Compile-time checking also makes Java secure. Consider a scenario in which an unauthorized method is trying to access the private variable, in this case, the JVM gives the compile-time error. It prevents the system from crashing.

Cryptographic Security

Java provides a class named java.secrurity.SourceCode that also provides security. If we get code from other sources, we should check from where the code is coming from. The class maintains the source information and provides guarantees to keep a digital signature and cryptographic security.

Java Sandbox

Java Sandbox is a major component of security consideration. It is a restricted area where applets are run. Java does not provide system resources without checking if an applet is to be run.

Exception Handling

The exception handling feature adds more security in Java. The feature reports the error to the programmer during the runtime. The code will not run until the programmer will not rectify it

-----------------------------------------------------------------------------------------------------------------

**4.Java 8**

<https://www.digitalocean.com/community/tutorials/java-8-features-with-examples>

Java 8 introduced several powerful features that enhance performance, readability, and maintainability of code. Below, we'll go through these features in detail:

### **1. forEach() Method in the Iterable Interface**

* Java 8 added the forEach() method to the Iterable interface (which is implemented by most collection classes). This method provides an easy way to iterate over elements in a collection.
* The forEach() method takes a **Consumer** functional interface as a parameter and applies it to each element of the collection.

**Example**:

List<String> list = Arrays.asList("apple", "banana", "cherry");

list.forEach(item -> System.out.println(item));

* In the above code, forEach() iterates through the list and prints each element. This makes the code more concise and readable than using a traditional for loop.
* **Benefits**:  
  + It allows processing elements in a more declarative style.
  + It can be combined with lambda expressions, making it easy to work with collections in a functional manner.

### **2. Default and Static Methods in Interfaces**

* **Default Methods**: In Java 8, interfaces can now have method implementations using the default keyword. This feature allows adding new methods to interfaces without breaking the existing implementations.

**Example**:  
 interface MyInterface {

default void greet() {

System.out.println("Hello from default method");

}

}

class MyClass implements MyInterface {

// No need to implement greet, it can be inherited.

}

**Why this matters**: This helps in evolving APIs without requiring changes to all classes that implement the interface.

* **Static Methods**: Interfaces can now have static methods, which were not allowed before Java 8.

**Example**:  
 interface MyInterface {

static void staticMethod() {

System.out.println("Static method in interface");

}

}

MyInterface.staticMethod(); // Calling static method

### **3. Functional Interfaces and Lambda Expressions**

* **Functional Interfaces**: A functional interface is an interface with just **one abstract method**, but it can have multiple default or static methods. These interfaces are the target types for lambda expressions or method references.  
  + Examples: Runnable, Callable, Comparator, Predicate, Function.
* **Lambda Expressions**: Lambda expressions allow you to pass behavior (functions) as arguments to methods, enabling functional programming features in Java.

**Example**:  
 List<String> list = Arrays.asList("apple", "banana", "cherry");

list.forEach(item -> System.out.println(item)); // Lambda expression

* + **Benefits**:
    - Concise and readable syntax.
    - Enables functional-style programming by passing functions as arguments.

### **4. Java Stream API for Bulk Data Operations on Collections**

* The **Stream API** in Java 8 is a new abstraction for handling sequences of elements (like collections or arrays) in a functional programming style.
* Streams allow you to perform bulk operations like filtering, mapping, and reducing without modifying the underlying data.
* Key Operations:
  + **Intermediate operations** (e.g., filter(), map(), distinct()) return a new stream.
  + **Terminal operations** (e.g., forEach(), collect(), reduce()) trigger the processing of the stream.

**Example**:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()

.filter(n -> n % 2 == 0)

.mapToInt(Integer::intValue)

.sum();

System.out.println(sum); // Output: 6 (2 + 4)

* **Benefits**:
  + Allows declarative data processing.
  + Operations can be done in parallel, making use of multi-core processors for performance improvements.
  + Code is more concise and readable.

### **5. Java Time API (New Date and Time API)**

* Prior to Java 8, Date and Calendar were problematic in terms of thread safety and mutability. The new **java.time** package provides a modern, immutable, and more flexible API for working with date and time.
* It includes classes like LocalDate, LocalTime, LocalDateTime, ZonedDateTime, Duration, and Period.

**Example**:

LocalDate today = LocalDate.now();

System.out.println(today); // Output: current date

* **Benefits**:
  + Immutable objects to avoid unintentional changes.
  + Thread-safe and easier to use compared to Date and Calendar.
  + Better handling of time zones and durations.

### **6. Collection API Improvements**

* Java 8 introduced several useful methods to the **Collection** interfaces, such as removeIf(), forEach(), spliterator(), and stream().

**Example**: Using removeIf() to remove elements conditionally:  
  
 List<String> list = new ArrayList<>(Arrays.asList("apple", "banana", "cherry"));

list.removeIf(s -> s.startsWith("b"));

System.out.println(list); // Output: [apple, cherry]

* **Benefits**:  
  + Simplifies code for common tasks (e.g., filtering elements from a collection).
  + Enhanced flexibility and functionality for working with collections.

### **7. Concurrency API Improvements**

* Java 8 introduced the **CompletableFuture** class for better handling of asynchronous tasks and parallel programming.
* It allows non-blocking, asynchronous programming and provides methods like thenApply(), thenAccept(), and join() to compose multiple tasks.

**Example**:

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 2)

.thenApplyAsync(result -> result \* 2);

future.thenAccept(result -> System.out.println(result)); // Output: 4

* **Benefits**:
  + Improves the ease of writing asynchronous and parallel code.
  + Handles more complex scenarios, such as chaining asynchronous tasks.

### **8. Java IO Improvements**

* Java 8 introduced improvements to the **java.nio.file** package and **java.nio** for easier handling of file systems and paths.
* **Files utility class** has several new methods such as readAllLines(), lines(), and copy(), which simplify file reading and writing.

**Example**:  
  
 Path path = Paths.get("test.txt");

try {

Files.write(path, "Hello, World!".getBytes());

List<String> lines = Files.readAllLines(path);

lines.forEach(System.out::println);

} catch (IOException e) {

e.printStackTrace();

}

* **Benefits**:  
  + Simplified handling of file reading/writing.
  + Enhanced support for file operations with stream-like behavior.

### **Conclusion:**

Java 8 introduced a host of new features that make Java more powerful, efficient, and readable. From **functional programming** support with **lambda expressions** and **functional interfaces** to the **Stream API** and **improved concurrency**, these changes modernized Java and set the stage for the future. Additionally, the **Java Time API** and **improvements in the Collection and IO APIs** brought better handling and manipulation of data and resources. These features have become essential tools for developers to write cleaner, more expressive, and performant Java code.

**5.CompletableFuture in Java 8:**

<https://www.geeksforgeeks.org/completablefuture-in-java/?ref=ml_lbp>

CompletableFuture provides a powerful and flexible way to write asynchronous, non-blocking code. It was introduced in Java 8 and has become popular due to its ease of use and ability to handle complex asynchronous workflows

What is a CompletableFuture?

CompletableFuture is a class in [java.util.concurrent](https://www.geeksforgeeks.org/java-util-concurrent-package/) package that implements the Future and CompletionStage Interface. It represents a future result of an asynchronous computation. It can be thought of as a container that holds the result of an asynchronous operation that is being executed in a different thread. It provides a number of methods to perform various operations on the result of the async computation.

------------------------------------------------------------------------------------------------------------------

**6.functional interface**

<https://www.geeksforgeeks.org/functional-interfaces-java/>

<https://www.javatpoint.com/java-8-functional-interfaces>

In Java, **functional interfaces** are used primarily to support **lambda expressions** and **method references**, which make the code more concise, readable, and flexible when working with functional programming concepts.

### **1. Why use Functional Interfaces in Java?**

A **functional interface** is an interface that has just one abstract method. It can have multiple default or static methods, but only one abstract method. Functional interfaces enable Java to support **functional programming** by allowing the use of **lambda expressions** and **method references**.

* **Improved readability**: You can write code more clearly and succinctly, especially when passing behavior as arguments to methods or creating small pieces of code (like predicates or actions).
* **Enabling functional programming**: They allow you to treat behavior as data, passing logic (in the form of lambdas) to methods and functions.
* **Cleaner code**: Reduces boilerplate code associated with creating anonymous inner classes.

### **2. Lambda Expressions**

Lambda expressions provide a clear and concise way to represent **one-method interfaces** (i.e., functional interfaces). They allow you to pass a block of code as an argument to a method or store it as a variable.

**Syntax:**

(parameters) -> expression

**Example:**

// Functional interface

@FunctionalInterface

interface MathOperation {

int operate(int a, int b);

}

// Using a lambda expression

MathOperation add = (a, b) -> a + b;

System.out.println(add.operate(5, 3)); // Output: 8

Here, the MathOperation interface is functional, and the lambda expression (a, b) -> a + b provides a concrete implementation of the operate method.

### **3. Method References**

Method references are a shorthand notation to call a method using a reference to it. They make code more readable and concise when you already have a method that matches the functional interface.

**Syntax:**

ClassName::methodName

**Example:**

// Functional interface

@FunctionalInterface

interface Greeting {

void greet(String name);

}

// Using method reference

Greeting greetMessage = System.out::println;

greetMessage.greet("Hello!"); // Output: Hello!

In this case, System.out::println is a method reference to the println method, which matches the greet method of the Greeting functional interface.

### **Summary:**

* **Functional interfaces** enable the use of **lambda expressions** and **method references**.
* **Lambda expressions** allow you to express instances of single-method interfaces concisely.
* **Method references** provide a shorter syntax to invoke methods directly.
* These features simplify code, make it more readable, and enable **functional programming** in Java.

------------------------------------------------------------------------------------------------------------------

**7.Volatile**

<https://www.javatpoint.com/volatile-keyword-in-java>

<https://www.geeksforgeeks.org/volatile-keyword-in-java/>

**Transient**

<https://www.javatpoint.com/transient-keyword>

<https://www.geeksforgeeks.org/transient-keyword-java/>

---------------------------------------------------------------------------------------------------------------------