**Brief All Java Versions**

java 8 new features and enhancements

**Java 8** introduced a variety of new features and enhancements that significantly transformed how developers write Java code, along with performance improvements in the JVM and APIs. Here is a summary of the **new features** and **enhancements** in Java 8:

### ****New Features in Java 8****

1. **Lambda Expressions**:
   * Allows you to express instances of single-method interfaces (functional interfaces) more concisely.
   * **Example**:

java

Copy code

(int x, int y) -> x + y;

1. **Functional Interfaces**:
   * Introduced @FunctionalInterface annotation for interfaces with only one abstract method.
   * Common functional interfaces include Predicate<T>, Function<T, R>, Consumer<T>, and Supplier<T>.
2. **Stream API**:
   * A new API for processing sequences of elements (collections, arrays) in a functional programming style.
   * Allows operations such as filter, map, reduce, and more on collections.
   * **Example**:

List<String> names = Arrays.asList("John", "Jane", "Jack");

names.stream().filter(name -> name.startsWith("J")).forEach(System.out::println);

1. **Default Methods in Interfaces**:
   * Interfaces can now have methods with default implementations, allowing backward compatibility without requiring all implementing classes to update.
   * **Example**:

interface MyInterface {

void method1();

default void method2() {

System.out.println("Default method");

}

}

### 5 Static -****Utility or Helper Methods****

Static methods in interfaces are often used to provide utility or helper methods that are closely related to the interface. This allows you to bundle static methods directly with the interface, making them more convenient to access and keeping relevant code in one place.

**Example**:

java

Copy code

interface Calculator {

// Static method for adding two numbers

static int add(int a, int b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

// Using the static method from the interface

int sum = Calculator.add(5, 3);

System.out.println("Sum: " + sum); // Output: Sum: 8

}

}

**Use Case**: You can define utility functions related to the interface’s functionality, such as mathematical operations or validation methods, without the need to instantiate a class implementing the interface.

1. **Optional Class**:
   * Provides a container object which may or may not contain a non-null value, helping to avoid NullPointerException.
   * **Example**:

Optional<String> opt = Optional.ofNullable("Hello");

opt.ifPresent(System.out::println);

**Key Methods in the Optional Class**

1. **of(T value)**:
   * **Description**: Creates an Optional with the specified non-null value.
   * **Usage**: You should use this when you are sure that the value is not null, as it throws a NullPointerException if the value is null.

java

Copy code

Optional<String> optional = Optional.of("Hello");

1. **ofNullable(T value)**:
   * **Description**: Creates an Optional that may hold a null value. If the value is null, an empty Optional is returned.
   * **Usage**: Use this when the value may be null.

java

Copy code

Optional<String> optional = Optional.ofNullable(null); // returns Optional.empty()

1. **empty()**:
   * **Description**: Returns an empty Optional.
   * **Usage**: Use this to create an empty Optional object explicitly.

java

Copy code

Optional<String> optional = Optional.empty();

1. **isPresent()**:
   * **Description**: Returns true if a value is present; otherwise, false.
   * **Usage**: Use this to check if the Optional contains a non-null value.

java

Copy code

if (optional.isPresent()) {

System.out.println("Value is present");

}

1. **ifPresent(Consumer<? super T> action)**:
   * **Description**: If a value is present, it performs the given action (a lambda expression or method reference) on the value.
   * **Usage**: Use this to perform an action only when the value is present.

java

Copy code

optional.ifPresent(value -> System.out.println("Value: " + value));

1. **get()**:
   * **Description**: Returns the value if present; otherwise, throws NoSuchElementException.
   * **Usage**: Use with caution, as calling this on an empty Optional will throw an exception.

java

Copy code

String value = optional.get(); // risky if the value is not present

1. **orElse(T other)**:
   * **Description**: Returns the value if present; otherwise, returns the provided default value.
   * **Usage**: Use this to provide a fallback value when the Optional is empty.

java

Copy code

String value = optional.orElse("Default Value");

1. **orElseGet(Supplier<? extends T> other)**:
   * **Description**: Returns the value if present; otherwise, invokes the provided Supplier and returns its result.
   * **Usage**: Use this when the fallback value might be expensive to compute, allowing it to be computed lazily.

java

Copy code

String value = optional.orElseGet(() -> "Lazy Default Value");

1. **orElseThrow(Supplier<? extends X> exceptionSupplier)**:
   * **Description**: Returns the value if present; otherwise, throws an exception produced by the provided Supplier.
   * **Usage**: Use this when you want to throw a custom exception if the value is not present.

java

Copy code

String value = optional.orElseThrow(() -> new IllegalArgumentException("No value present"));

1. **map(Function<? super T, ? extends U> mapper)**:

* **Description**: If a value is present, it applies the provided mapping function to the value and returns an Optional of the result.
* **Usage**: Use this to transform the value if it is present.

java

Copy code

Optional<Integer> length = optional.map(String::length);

1. **flatMap(Function<? super T, Optional<U>> mapper)**:

* **Description**: Similar to map(), but the mapping function returns an Optional. If the mapping function returns Optional.empty(), the result is also Optional.empty().
* **Usage**: Use this when the mapping function itself returns an Optional.

java

Copy code

Optional<String> flatMapped = optional.flatMap(value -> Optional.of("Mapped Value"));

1. **filter(Predicate<? super T> predicate)**:

* **Description**: If a value is present and matches the given predicate, it returns that value. Otherwise, it returns an empty Optional.
* **Usage**: Use this to filter the value based on a condition.

java

Copy code

Optional<String> filtered = optional.filter(value -> value.length() > 3);

**Example Use Cases of Optional**

1. **Avoiding Null Checks**: Instead of manually checking for null values, you can use Optional to manage potential nulls.

java

Copy code

// Traditional null check

if (user != null && user.getName() != null) {

System.out.println(user.getName());

}

// Using Optional

Optional.ofNullable(user)

.map(User::getName)

.ifPresent(System.out::println);

1. **Providing Default Values**: You can use orElse() or orElseGet() to provide default values if the Optional is empty.

java

Copy code

String name = Optional.ofNullable(user.getName())

.orElse("Default Name");

1. **Chaining Operations**: The map() and flatMap() methods allow chaining of operations, especially when working with nested objects.

java

Copy code

Optional.ofNullable(user)

.flatMap(User::getAddress)

.map(Address::getCity)

.ifPresent(System.out::println);

**When to Use Optional**

* **Return Type for Methods**: Use Optional as a return type for methods that may return a null value.
* **Avoiding NullPointerException**: Use Optional when you want to clearly indicate that a value may be absent and avoid null pointer exceptions.

**Best Practices**

* **Do Not Overuse**: Optional is not intended for fields or method parameters. It is primarily meant as a return type for methods.
* **Do Not Use Optional.get() Unconditionally**: Avoid calling Optional.get() without checking if a value is present. Use methods like orElse() or orElseThrow() instead.

Optional is a powerful feature in Java 8 that helps avoid null checks and improves the readability and maintainability of code by explicitly handling nullable values.

1. **New Date and Time API (java.time)**:
   * A completely new date and time API that replaces java.util.Date and java.util.Calendar.
   * **Key Classes**: LocalDate, LocalTime, LocalDateTime, ZonedDateTime, Duration, and Period.
   * **Example**:

java

Copy code

LocalDate today = LocalDate.now();

LocalDate birthday = LocalDate.of(1990, Month.JANUARY, 1);

Java 8 introduced a new java.time package to address many shortcomings of the previous java.util.Date and java.util.Calendar APIs. It provides a comprehensive set of classes for working with dates, times, and time zones in a more user-friendly and flexible way.

**Parsing a String to LocalDate**

To parse a string into a LocalDate, you can use the DateTimeFormatter class to define the format of the date and then use the LocalDate.parse() method to convert the string into a LocalDate object.

**Example: Parse String to LocalDate**

java

Copy code

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

public class DateExample {

public static void main(String[] args) {

// Define the date format

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");

// Parse the string into LocalDate

String dateString = "18/10/2024";

LocalDate date = LocalDate.parse(dateString, formatter);

// Output the LocalDate

System.out.println("Parsed Date: " + date);

}

}

* **Explanation**:
  + DateTimeFormatter.ofPattern("dd/MM/yyyy"): Defines the pattern for parsing the date. In this case, it expects the date in day/month/year format.
  + LocalDate.parse(dateString, formatter): Parses the string into a LocalDate object.

**Parsing a String to LocalTime (e.g., "3:30 PM")**

You can also parse a time string like "3:30 PM" into a LocalTime object using DateTimeFormatter.

**Example: Parse String to LocalTime**

java

Copy code

import java.time.LocalTime;

import java.time.format.DateTimeFormatter;

public class TimeExample {

public static void main(String[] args) {

// Define the time format (h:mm a for 12-hour format with AM/PM)

DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern("h:mm a");

// Parse the string into LocalTime

String timeString = "3:30 PM";

LocalTime time = LocalTime.parse(timeString, timeFormatter);

// Output the LocalTime

System.out.println("Parsed Time: " + time);

}

}

* **Explanation**:
  + DateTimeFormatter.ofPattern("h:mm a"): Defines the pattern for parsing time. Here, h is for 12-hour time (1-12), mm is for minutes, and a represents AM/PM.
  + LocalTime.parse(timeString, timeFormatter): Parses the string into a LocalTime object.

**Parsing a String to LocalDateTime**

You can also combine a date and time into a LocalDateTime object using LocalDateTime.parse() and an appropriate DateTimeFormatter.

**Example: Parse String to LocalDateTime**

java

Copy code

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

public class DateTimeExample {

public static void main(String[] args) {

// Define the date and time format

DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("dd/MM/yyyy h:mm a");

// Parse the string into LocalDateTime

String dateTimeString = "18/10/2024 3:30 PM";

LocalDateTime dateTime = LocalDateTime.parse(dateTimeString, dateTimeFormatter);

// Output the LocalDateTime

System.out.println("Parsed DateTime: " + dateTime);

}

}

* **Explanation**:
  + DateTimeFormatter.ofPattern("dd/MM/yyyy h:mm a"): Defines the pattern for parsing a date and time.
  + LocalDateTime.parse(dateTimeString, dateTimeFormatter): Parses the string into a LocalDateTime object.

### Example: Convert LocalDate to String with a Custom Format

Here’s a code example demonstrating how to convert a LocalDate to a string in any custom format using DateTimeFormatter.

java

Copy code

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

public class LocalDateToStringExample {

public static void main(String[] args) {

// Get the current date or create a specific date

LocalDate localDate = LocalDate.now(); // or LocalDate.of(2024, 10, 18);

// Define a custom format

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");

// Format the LocalDate to a String

String formattedDate = localDate.format(formatter);

// Output the formatted date string

System.out.println("Formatted Date: " + formattedDate);

}

}

**Explanation:**

1. **LocalDate.now()**: This method gets the current date. Alternatively, you can use LocalDate.of(year, month, day) to create a specific date.
2. **DateTimeFormatter.ofPattern("dd/MM/yyyy")**: This creates a DateTimeFormatter with the pattern dd/MM/yyyy (day/month/year). You can change this pattern to any format you want, such as yyyy-MM-dd, MMM dd, yyyy, etc.
3. **localDate.format(formatter)**: This formats the LocalDate into the string representation based on the provided formatter.

**Different Patterns You Can Use**

* **yyyy-MM-dd**: 2024-10-18
* **dd/MM/yyyy**: 18/10/2024
* **MMM dd, yyyy**: Oct 18, 2024
* **EEEE, MMMM dd, yyyy**: Friday, October 18, 2024

**Example with a Different Format**

java

Copy code

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

public class CustomDateFormatExample {

public static void main(String[] args) {

// Define a specific date

LocalDate date = LocalDate.of(2024, 10, 18);

// Define a custom date format

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("EEEE, MMM dd, yyyy");

// Convert LocalDate to String in the desired format

String formattedDate = date.format(formatter);

// Output the result

System.out.println("Formatted Date: " + formattedDate);

}

}

Output:

mathematica

Copy code

Formatted Date: Friday, Oct 18, 2024

This approach gives you the flexibility to represent dates in any format using Java 8’s DateTimeFormatter and LocalDate.

1. **Method References**:
   * Provides a way to refer to methods or constructors directly using the :: symbol, making lambda expressions more concise.
   * **Example**:

java

Copy code

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

1. **Collectors (in Stream API)**:
   * A utility class that provides various methods to collect stream results into collections or other results like List, Set, Map.
   * **Example**:

java

Copy code

List<String> list = stream.collect(Collectors.toList());

1. **CompletableFuture**:
   * A new class for asynchronous programming, replacing the use of Future and making it easier to chain tasks and handle exceptions.
   * **Example**:

java

Copy code

CompletableFuture.supplyAsync(() -> "Hello").thenAccept(System.out::println);

Here’s a detailed example of how to use various CompletableFuture methods in Java 8, including supplyAsync(), thenApply(), thenCompose(), thenCombine(), and thenApplyAsync() while making an API call using RestTemplate. We’ll simulate fetching a list of objects from an API and then perform different operations on them using CompletableFuture.

**Scenario:**

* We will use RestTemplate.exchange() to make an API call and retrieve a list of User objects.
* The methods of CompletableFuture will be demonstrated as we perform various asynchronous operations on this list.

**Example Code:**

java

Copy code

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

import java.util.function.Supplier;

import org.springframework.core.ParameterizedTypeReference;

import org.springframework.http.HttpMethod;

import org.springframework.http.ResponseEntity;

import org.springframework.web.client.RestTemplate;

class User {

private int id;

private String name;

// Constructors, Getters, and Setters

public User() {}

public User(int id, String name) {

this.id = id;

this.name = name;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public String toString() {

return "User{id=" + id + ", name='" + name + "'}";

}

}

public class CompletableFutureExample {

// Simulating an API call to get a list of users

public static CompletableFuture<List<User>> getUsers() {

return CompletableFuture.supplyAsync(() -> {

RestTemplate restTemplate = new RestTemplate();

String url = "https://jsonplaceholder.typicode.com/users"; // Replace with your own API URL

ResponseEntity<List<User>> response = restTemplate.exchange(

url,

HttpMethod.GET,

null,

new ParameterizedTypeReference<List<User>>() {}

);

return response.getBody();

});

}

public static void main(String[] args) throws Exception {

// Thread pool to simulate async tasks

ExecutorService executor = Executors.newFixedThreadPool(3);

// CompletableFuture to get list of users

CompletableFuture<List<User>> usersFuture = getUsers();

// Example of thenApply - Transform the list of users by getting their names

CompletableFuture<List<String>> userNamesFuture = usersFuture.thenApply(users -> {

System.out.println("Transforming user list to their names.");

return users.stream().map(User::getName).toList();

});

// Example of thenCompose - Flat map a future to make another call using user ID

CompletableFuture<String> userInfoFuture = usersFuture.thenCompose(users -> {

User user = users.get(0); // Fetching data of the first user

return CompletableFuture.supplyAsync(() -> "User Info for: " + user.getName());

});

// Example of thenCombine - Combine two futures (user list and user names)

CompletableFuture<String> combinedFuture = usersFuture.thenCombine(userNamesFuture, (users, userNames) -> {

return "First User: " + users.get(0).getName() + ", Name List: " + userNames;

});

// Example of thenApplyAsync - Running transformation in a separate thread

CompletableFuture<List<String>> asyncTransformedNamesFuture = usersFuture.thenApplyAsync(users -> {

System.out.println("Async Transforming user list to their names.");

return users.stream().map(User::getName).toList();

}, executor);

// Completing all CompletableFuture operations

usersFuture.thenRun(() -> System.out.println("All tasks completed!"));

// Waiting for the results

System.out.println("User Names: " + userNamesFuture.get()); // Result of thenApply

System.out.println("User Info: " + userInfoFuture.get()); // Result of thenCompose

System.out.println("Combined Result: " + combinedFuture.get()); // Result of thenCombine

System.out.println("Async Transformed Names: " + asyncTransformedNamesFuture.get()); // Result of thenApplyAsync

// Shutdown the executor

executor.shutdown();

executor.awaitTermination(10, TimeUnit.SECONDS);

}

}

**Explanation of CompletableFuture Methods:**

1. **supplyAsync(Supplier<U>)**:
   * Starts an asynchronous task that supplies a result. Here, it’s used to fetch the list of users from an API.

java

Copy code

CompletableFuture<List<User>> usersFuture = getUsers();

1. **thenApply(Function<T, U>)**:
   * Transforms the result of a CompletableFuture once it’s completed. We use this to transform the list of User objects into their names.

java

Copy code

CompletableFuture<List<String>> userNamesFuture = usersFuture.thenApply(users -> {

return users.stream().map(User::getName).toList();

});

1. **thenCompose(Function<T, CompletionStage<U>>):**
   * Allows chaining another asynchronous task that depends on the result of the first task. Here, we make another call using the first user's info.

java

Copy code

CompletableFuture<String> userInfoFuture = usersFuture.thenCompose(users -> {

User user = users.get(0);

return CompletableFuture.supplyAsync(() -> "User Info for: " + user.getName());

});

1. **thenCombine(CompletionStage<U>, BiFunction<T, U, V>):**
   * Combines the results of two CompletableFutures into a new future. In this case, we combine the original list of users and their names.

java

Copy code

CompletableFuture<String> combinedFuture = usersFuture.thenCombine(userNamesFuture, (users, userNames) -> {

return "First User: " + users.get(0).getName() + ", Name List: " + userNames;

});

1. **thenApplyAsync(Function<T, U>, Executor)**:
   * Executes the transformation on a different thread. We use this to asynchronously transform the user list into a list of names.

java

Copy code

CompletableFuture<List<String>> asyncTransformedNamesFuture = usersFuture.thenApplyAsync(users -> {

return users.stream().map(User::getName).toList();

}, executor);

1. **thenRun(Runnable)**:
   * Allows you to perform some action (Runnable) after the CompletableFuture has completed.

java

Copy code

usersFuture.thenRun(() -> System.out.println("All tasks completed!"));

**Output (Sample):**

plaintext

Copy code

Transforming user list to their names.

User Names: [Leanne Graham, Ervin Howell, Clementine Bauch, Patricia Lebsack, Chelsey Dietrich, Mrs. Dennis Schulist, Kurtis Weissnat, Nicholas Runolfsdottir V, Glenna Reichert, Clementina DuBuque]

User Info: User Info for: Leanne Graham

Combined Result: First User: Leanne Graham, Name List: [Leanne Graham, Ervin Howell, Clementine Bauch, Patricia Lebsack, Chelsey Dietrich, Mrs. Dennis Schulist, Kurtis Weissnat, Nicholas Runolfsdottir V, Glenna Reichert, Clementina DuBuque]

Async Transforming user list to their names.

Async Transformed Names: [Leanne Graham, Ervin Howell, Clementine Bauch, Patricia Lebsack, Chelsey Dietrich, Mrs. Dennis Schulist, Kurtis Weissnat, Nicholas Runolfsdottir V, Glenna Reichert, Clementina DuBuque]

All tasks completed!

**Conclusion:**

This example demonstrates the use of CompletableFuture methods such as supplyAsync(), thenApply(), thenCompose(), thenCombine(), and thenApplyAsync() in a real-world scenario where we fetch data from an API and perform various asynchronous operations on it.

**1. supplyAsync(Supplier<U>)**

**Purpose**:

* Used to start an asynchronous task that returns a result. The task is run in a different thread (by default, in the common ForkJoinPool).

**Use Case**:

* When you want to run a task asynchronously that generates a result without blocking the main thread.

**Example**:

java

Copy code

CompletableFuture<List<User>> usersFuture = CompletableFuture.supplyAsync(() -> {

// Simulate API call or I/O operation

return fetchUsersFromAPI();

});

**Difference**:

* supplyAsync() is suitable when you have a long-running task that produces a value and you don’t want to block the main thread while waiting for the result.

**2. thenApply(Function<T, U>)**

**Purpose**:

* Transforms the result of a CompletableFuture once it completes. It takes the result of the previous stage and applies a function to transform it synchronously.

**Use Case**:

* Use when you want to perform a transformation on the result of a CompletableFuture. For example, transforming a list of users into a list of usernames.

**Example**:

java

Copy code

CompletableFuture<List<String>> userNamesFuture = usersFuture.thenApply(users -> {

return users.stream().map(User::getName).toList();

});

**Difference**:

* thenApply() is a synchronous method, meaning the transformation happens in the same thread that completed the previous stage.

**3. thenApplyAsync(Function<T, U>)**

**Purpose**:

* Similar to thenApply(), but the transformation happens asynchronously, potentially in a different thread from the common pool (or a custom thread pool if provided).

**Use Case**:

* When the transformation itself is time-consuming or you want to perform it on another thread to avoid blocking the thread that completed the previous stage.

**Example**:

java

Copy code

CompletableFuture<List<String>> asyncUserNamesFuture = usersFuture.thenApplyAsync(users -> {

return users.stream().map(User::getName).toList();

});

**Difference**:

* thenApplyAsync() ensures that the transformation happens in a different thread (either the common pool or a custom executor), making it suitable for parallelism.

**4. thenCompose(Function<T, CompletionStage<U>>)**

**Purpose**:

* Allows you to chain multiple asynchronous tasks where the next task depends on the result of the previous one. It "flattens" a CompletableFuture of CompletableFuture<U> into a single CompletableFuture<U>.

**Use Case**:

* When you need to start another asynchronous task that depends on the result of the first one. For example, after fetching user data, you might want to fetch the user’s detailed profile.

**Example**:

java

Copy code

CompletableFuture<UserDetails> userDetailsFuture = usersFuture.thenCompose(users -> {

User user = users.get(0); // Get the first user

return fetchUserDetails(user.getId()); // Fetch user details asynchronously

});

**Difference**:

* thenCompose() is like "flatMap" in functional programming. It’s used for chaining dependent asynchronous tasks, while thenApply() is used to transform the result of the previous task synchronously.

**5. thenCombine(CompletionStage<U>, BiFunction<T, U, V>)**

**Purpose**:

* Combines the results of two independent CompletableFutures and produces a new result when both are complete. It is useful when you have two asynchronous tasks and need to perform an operation once both are finished.

**Use Case**:

* When you have two independent tasks and want to combine their results. For example, fetching user data and fetching user transaction history, and then combining both results.

**Example**:

java

Copy code

CompletableFuture<String> combinedFuture = usersFuture.thenCombine(userNamesFuture, (users, userNames) -> {

return "First User: " + users.get(0).getName() + ", Names List: " + userNames;

});

**Difference**:

* thenCombine() combines the results of two independent asynchronous operations, unlike thenCompose(), which chains dependent operations. It’s useful when tasks are unrelated but need to be combined.

**6. thenAccept(Consumer<T>)**

**Purpose**:

* Consumes the result of the CompletableFuture when it’s completed without returning any new CompletableFuture. It’s a terminal operation.

**Use Case**:

* When you only care about processing the result and not returning a value. For example, logging the result to the console.

**Example**:

java

Copy code

usersFuture.thenAccept(users -> {

users.forEach(user -> System.out.println(user.getName()));

});

**Difference**:

* thenAccept() doesn’t return any result. It’s useful when you just want to process the result, not transform it or chain another future.

**7. thenRun(Runnable)**

**Purpose**:

* Runs a Runnable once the CompletableFuture completes. The difference from thenAccept() is that thenRun() does not take the result of the previous CompletableFuture.

**Use Case**:

* When you want to execute a task after a CompletableFuture finishes, but you don’t need the result of the future.

**Example**:

java

Copy code

usersFuture.thenRun(() -> {

System.out.println("Task is complete!");

});

**Difference**:

* thenRun() is used when you don’t care about the result of the CompletableFuture, but just need to run a task afterward.

**8. exceptionally(Function<Throwable, U>)**

**Purpose**:

* Used for handling exceptions that occur in the CompletableFuture chain. It provides a fallback value in case of an error.

**Use Case**:

* When you want to handle errors and provide a default value or perform some action if an exception occurs during the async task.

**Example**:

java

Copy code

CompletableFuture<List<User>> safeUsersFuture = usersFuture.exceptionally(ex -> {

System.err.println("Error occurred: " + ex.getMessage());

return List.of(); // Return an empty list on error

});

**Difference**:

* exceptionally() is used specifically to handle exceptions and provide a fallback value in case of failure.

**Summary of Use Cases:**

* **supplyAsync()**: Run a task asynchronously that produces a result.
* **thenApply()**: Transform the result synchronously.
* **thenApplyAsync()**: Transform the result asynchronously on a different thread.
* **thenCompose()**: Chain dependent asynchronous tasks.
* **thenCombine()**: Combine results of two independent futures.
* **thenAccept()**: Process the result without returning a new value.
* **thenRun()**: Run a task after the future completes without using its result.
* **exceptionally()**: Handle exceptions and provide fallback values.

**Example of Combining These Concepts:**

java

Copy code

CompletableFuture<UserDetails> userDetailsFuture = CompletableFuture.supplyAsync(() -> {

return fetchUserFromAPI(); // First async task

}).thenCompose(user -> {

return fetchUserDetails(user.getId()); // Second dependent async task

}).thenApply(userDetails -> {

userDetails.enhanceWithAdditionalData(); // Transform the result

return userDetails;

}).thenCombine(fetchUserTransactions(), (userDetails, transactions) -> {

userDetails.setTransactions(transactions); // Combine two futures

return userDetails;

}).exceptionally(ex -> {

System.err.println("Error: " + ex.getMessage());

return new UserDetails(); // Fallback in case of error

});

This code demonstrates how you can chain multiple async operations, handle errors, and combine results of independent tasks.

1. **Nashorn JavaScript Engine**:
   * A new JavaScript engine for executing JavaScript code on the JVM, replacing the older Rhino engine.
   * **Example**:

ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");

engine.eval("print('Hello, World!')");

1. **Base64 Encoding and Decoding**:
   * Java 8 introduced the java.util.Base64 class for encoding and decoding Base64.

Here’s an example of how to encode and decode data using Base64 in Java:

**1. Encoding a String to Base64**

java

Copy code

import java.util.Base64;

public class Base64Example {

public static void main(String[] args) {

String originalString = "Hello, World!";

// Encode the string into Base64

String encodedString = Base64.getEncoder().encodeToString(originalString.getBytes());

System.out.println("Encoded String: " + encodedString);

}

}

**2. Decoding a Base64 Encoded String**

java

Copy code

import java.util.Base64;

public class Base64Example {

public static void main(String[] args) {

String encodedString = "SGVsbG8sIFdvcmxkIQ=="; // Example of Base64 encoded string

// Decode the Base64 string

byte[] decodedBytes = Base64.getDecoder().decode(encodedString);

String decodedString = new String(decodedBytes);

System.out.println("Decoded String: " + decodedString);

}

}

**3. Encoding and Decoding a Byte Array**

java

Copy code

import java.util.Base64;

public class Base64Example {

public static void main(String[] args) {

byte[] byteArray = {1, 2, 3, 4, 5};

// Encode the byte array to Base64

String encodedBytes = Base64.getEncoder().encodeToString(byteArray);

System.out.println("Encoded Bytes: " + encodedBytes);

// Decode the Base64 string back to byte array

byte[] decodedBytes = Base64.getDecoder().decode(encodedBytes);

System.out.println("Decoded Bytes: ");

for (byte b : decodedBytes) {

System.out.print(b + " ");

}

}

}

**Explanation:**

1. **Encoding**:
   * Base64.getEncoder().encodeToString(byte[]) is used to encode a byte[] or String to a Base64 encoded string.
2. **Decoding**:
   * Base64.getDecoder().decode(String) is used to decode a Base64 encoded string back to its original form.

This is useful for scenarios like encoding data for storage or transfer across systems where data integrity needs to be maintained.

### ****Enhancements in Java 8****

1. **JVM and Memory Management Enhancements**:
   * **PermGen Removal**:
     + The PermGen space was removed and replaced by **Metaspace**, which dynamically grows based on the application needs.
   * **Improved Garbage Collection**:
     + Java 8 improved the **G1 Garbage Collector** to reduce pause times and improve performance in multi-threaded applications.
2. **Parallel Sorting of Arrays**:
   * The Arrays.parallelSort() method was introduced to sort large arrays using multi-core processors.
   * **Example**:

int[] numbers = {3, 1, 4, 1, 5, 9};

Arrays.parallelSort(numbers);

1. **Security Enhancements**:
   * Support for **TLS 1.2** (Transport Layer Security) was added by default.
   * **Improved Cryptographic Algorithms** and random number generation with better defaults for secure applications.
2. **Type Annotations**:
   * Java 8 extended annotations to be applied to any use of a type, including generic types and casts, allowing more flexibility in applying annotations.
   * **Example**:

@NotNull List<String> strings = new ArrayList<>();

1. **Concurrency Enhancements**:
   * Introduced **LongAdder** and **DoubleAdder**, optimized classes for counting in concurrent environments, providing better performance than AtomicLong in highly contended situations.
2. **Method Handles and Lambda Metafactory**:
   * Enhancements to **Method Handles** and **LambdaMetafactory** improve the performance of lambda expressions, optimizing bytecode generation.
3. **JavaFX Enhancements**:
   * JavaFX was bundled with Java 8, becoming the primary platform for building rich desktop applications.
   * Introduced 3D graphics support and other UI improvements.
4. **Stream API Performance Enhancements**:
   * Streams can be run in parallel by invoking parallelStream(), taking advantage of multiple cores to improve performance.
   * **Example**:

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

numbers.parallelStream().forEach(System.out::println);

1. **Compact Profiles**:
   * Java 8 introduced **Compact Profiles**, which allow developers to deploy a reduced version of the Java Runtime Environment (JRE) for devices with limited resources.
2. **Reflection API Enhancements**:
   * Enhancements to the reflection API improved support for type annotations and parameterized types, making the API more flexible.

### Summary of New Features vs. Enhancements:

* **New Features** focus on adding new functionality, such as **lambda expressions**, the **Stream API**, **Optional**, and the **new Date and Time API**.
* **Enhancements** improve existing functionality, such as **better garbage collection**, **parallel sorting**, **security updates**, and **JVM memory management** with the removal of PermGen.

These changes made Java 8 a landmark release that modernized the language, improved performance, and supported more concise and functional programming styles.

**Functional Interface**

A functional interface has only one abstract method (called a **functional method**), but can have multiple default or static methods. The main benefit of functional interfaces is that they can be implemented by lambda expressions or method references.

The @FunctionalInterface annotation can be used to ensure an interface conforms to this pattern.

**Common Functional Interfaces in Java 8:**

1. **Function<T, R>**:
   * Represents a function that accepts one argument and produces a result.
   * **Abstract Method**: R apply(T t)
   * **Example**:

Function<Integer, String> intToString = Object::toString;

String result = intToString.apply(123); // "123"

1. **Predicate<T>**:
   * Represents a function that takes one argument and returns a boolean.
   * **Abstract Method**: boolean test(T t)
   * **Example**:

Predicate<String> isEmpty = String::isEmpty;

boolean result = isEmpty.test(""); // true

1. **Consumer<T>**:
   * Represents a function that accepts a single argument and returns no result (void).
   * **Abstract Method**: void accept(T t)
   * **Example**:

Consumer<String> print = System.out::println;

print.accept("Hello, world!"); // Outputs: "Hello, world!"

1. **Supplier<T>**:
   * Represents a supplier of results, takes no arguments, returns a result.
   * **Abstract Method**: T get()
   * **Example**:

Supplier<Double> randomSupplier = Math::random;

double random = randomSupplier.get(); // Returns a random double

1. **UnaryOperator<T>**:
   * A specialization of Function that accepts and returns the same type.
   * **Abstract Method**: T apply(T t)
   * **Example**:

UnaryOperator<Integer> square = x -> x \* x;

int result = square.apply(5); // 25

1. **BinaryOperator<T>**:
   * A specialization of BiFunction that accepts two arguments of the same type and returns a result of the same type.
   * **Abstract Method**: T apply(T t1, T t2)
   * **Example**:

BinaryOperator<Integer> sum = (a, b) -> a + b;

int result = sum.apply(2, 3); // 5

**BiFunctional Interfaces**

In addition to Function<T, R>, Java 8 introduced **bi-functional interfaces** that accept two input arguments.

1. **BiFunction<T, U, R>**:
   * Represents a function that accepts two arguments of types T and U and produces a result of type R.
   * **Abstract Method**: R apply(T t, U u)
   * **Example**:

BiFunction<Integer, Integer, String> sumToString = (a, b) -> String.valueOf(a + b);

String result = sumToString.apply(2, 3); // "5"

1. **BiPredicate<T, U>**:
   * Represents a predicate (boolean-valued function) that takes two arguments.
   * **Abstract Method**: boolean test(T t, U u)
   * **Example**:

BiPredicate<String, String> isEqual = String::equals;

boolean result = isEqual.test("Java", "Java"); // true

1. **BiConsumer<T, U>**:
   * Represents an operation that accepts two input arguments and returns no result (void).
   * **Abstract Method**: void accept(T t, U u)
   * **Example**:

BiConsumer<String, Integer> printWithAge = (name, age) -> System.out.println(name + " is " + age);

printWithAge.accept("John", 25); // Outputs: "John is 25"

1. **ToIntBiFunction<T, U>**:
   * A specialized version of BiFunction that returns an int instead of a generic type R.
   * **Abstract Method**: int applyAsInt(T t, U u)
   * **Example**:

ToIntBiFunction<String, String> compareLength = (a, b) -> a.length() - b.length();

int result = compareLength.applyAsInt("apple", "pear"); // -1

1. **ToDoubleBiFunction<T, U>**:
   * A specialized version of BiFunction that returns a double.
   * **Abstract Method**: double applyAsDouble(T t, U u)
   * **Example**:

ToDoubleBiFunction<Integer, Integer> divide = (a, b) -> (double) a / b;

double result = divide.applyAsDouble(5, 2); // 2.5

**Summary of Functional Interfaces vs. BiFunctional Interfaces:**

* **Functional Interfaces** take one argument (e.g., Function<T, R>, Predicate<T>, Consumer<T>) and return a result or perform an action.
* **BiFunctional Interfaces** take two arguments (e.g., BiFunction<T, U, R>, BiPredicate<T, U>, BiConsumer<T, U>) and return a result or perform an action

**1. Creation of Streams**

These methods create a stream from various sources such as collections, arrays, or specific values.

* **Stream.of(T... values)**:
  + Creates a stream from specified values.
  + **Example**:

Stream<String> stream = Stream.of("a", "b", "c");

* **Collection.stream()**:
  + Creates a sequential stream from a collection.
  + **Example**:

List<String> list = Arrays.asList("a", "b", "c");

Stream<String> stream = list.stream();

* **Arrays.stream(T[] array)**:
  + Creates a stream from an array.
  + **Example**:

int[] numbers = {1, 2, 3};

IntStream stream = Arrays.stream(numbers);

* **Stream.generate(Supplier<T> s)**:
  + Generates an infinite stream using a supplier.
  + **Example**:

Stream<Double> randoms = Stream.generate(Math::random);

* **Stream.iterate(T seed, UnaryOperator<T> f)**:
  + Generates an infinite stream from a seed and applies a function to each element to produce the next.
  + **Example**:

java

Copy code

Stream<Integer> stream = Stream.iterate(0, n -> n + 2); // 0, 2, 4, 6, ...

**2. Intermediate Operations**

These methods return a new stream. Intermediate operations are **lazy**; they are not executed until a terminal operation is invoked.

* **filter(Predicate<T> predicate)**:
  + Filters elements based on a given predicate.
  + **Example**:

java

Copy code

Stream<String> stream = Stream.of("apple", "banana", "cherry").filter(s -> s.startsWith("a"));

* **map(Function<T, R> mapper)**:
  + Transforms each element using a provided function.
  + **Example**:

java

Copy code

Stream<Integer> lengths = Stream.of("apple", "banana", "cherry").map(String::length);

* **flatMap(Function<T, Stream<R>> mapper)**:
  + Flattens nested structures into a single stream.
  + **Example**:

java

Copy code

Stream<List<Integer>> listOfLists = Stream.of(Arrays.asList(1, 2), Arrays.asList(3, 4));

Stream<Integer> flatStream = listOfLists.flatMap(List::stream); // 1, 2, 3, 4

* **distinct()**:
  + Removes duplicate elements.
  + **Example**:

java

Copy code

Stream<Integer> distinctStream = Stream.of(1, 2, 2, 3).distinct(); // 1, 2, 3

* **sorted()**:
  + Sorts the stream in natural order.
  + **Example**:

java

Copy code

Stream<String> sortedStream = Stream.of("apple", "cherry", "banana").sorted(); // apple, banana, cherry

* **sorted(Comparator<T> comparator)**:
  + Sorts the stream using a custom comparator.
  + **Example**:

java

Copy code

Stream<String> sortedStream = Stream.of("apple", "cherry", "banana").sorted(Comparator.reverseOrder());

* **peek(Consumer<T> action)**:
  + Performs an action for each element of the stream, mainly for debugging purposes.
  + **Example**:

java

Copy code

Stream<String> peekedStream = Stream.of("apple", "banana").peek(System.out::println);

* **limit(long maxSize)**:
  + Limits the stream to the specified number of elements.
  + **Example**:

java

Copy code

Stream<Integer> limitedStream = Stream.iterate(0, n -> n + 1).limit(5); // 0, 1, 2, 3, 4

* **skip(long n)**:
  + Skips the first n elements of the stream.
  + **Example**:

java

Copy code

Stream<Integer> skippedStream = Stream.of(1, 2, 3, 4, 5).skip(2); // 3, 4, 5

**3. Terminal Operations**

These operations produce a result or side-effect and are typically executed after all intermediate operations.

* **forEach(Consumer<T> action)**:
  + Performs an action for each element.
  + **Example**:

java

Copy code

Stream.of("apple", "banana").forEach(System.out::println);

* **toArray(IntFunction<A[]> generator)**:
  + Converts the stream to an array.
  + **Example**:

java

Copy code

String[] array = Stream.of("a", "b", "c").toArray(String[]::new);

* **reduce(T identity, BinaryOperator<T> accumulator)**:
  + Reduces the elements of the stream to a single value using an initial value (identity) and a combining function.
  + **Example**:

java

Copy code

int sum = Stream.of(1, 2, 3).reduce(0, Integer::sum); // 6

* **collect(Collector<T, A, R> collector)**:
  + Collects the elements into a collection or another result.
  + **Example**:

java

Copy code

List<String> list = Stream.of("a", "b", "c").collect(Collectors.toList());

* **count()**:
  + Returns the number of elements in the stream.
  + **Example**:

java

Copy code

long count = Stream.of(1, 2, 3).count(); // 3

* **findFirst()**:
  + Returns the first element in the stream as an Optional.
  + **Example**:

java

Copy code

Optional<String> first = Stream.of("apple", "banana").findFirst();

* **findAny()**:
  + Returns any element in the stream as an Optional.
  + **Example**:

java

Copy code

Optional<String> any = Stream.of("apple", "banana").findAny();

* **allMatch(Predicate<T> predicate)**:
  + Returns true if all elements match the provided predicate.
  + **Example**:

java

Copy code

boolean allMatch = Stream.of(1, 2, 3).allMatch(n -> n > 0); // true

* **anyMatch(Predicate<T> predicate)**:
  + Returns true if any element matches the provided predicate.
  + **Example**:

java

Copy code

boolean anyMatch = Stream.of(1, 2, 3).anyMatch(n -> n > 2); // true

* **noneMatch(Predicate<T> predicate)**:
  + Returns true if no elements match the provided predicate.
  + **Example**:

java

Copy code

boolean noneMatch = Stream.of(1, 2, 3).noneMatch(n -> n > 3); // true

* **min(Comparator<T> comparator)**:
  + Returns the minimum element according to the provided comparator.
  + **Example**:

java

Copy code

Optional<Integer> min = Stream.of(3, 1, 2).min(Integer::compareTo); // 1

* **max(Comparator<T> comparator)**:
  + Returns the maximum element according to the provided comparator.
  + **Example**:

java

Copy code

Optional<Integer> max = Stream.of(3, 1, 2).max(Integer::compareTo); // 3

**4. Specialized Stream Types**

Java 8 also introduced specialized streams for primitive types:

* **IntStream**, **LongStream**, and **DoubleStream** for processing primitive int, long, and double values, respectively.

These streams offer additional methods like sum(), average(), and range() that are specific to numeric streams.

**Example: Combining Stream Operations**

Here’s how you might combine several of these operations:

java

Copy code

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

List<String> result = words.stream()

.filter(word -> word.length() > 5)

.map(String::toUpperCase)

.sorted()

.collect(Collectors.toList());

System.out.println(result); // Output: [BANANA, CHERRY]

This code filters words longer than 5 characters, converts them to uppercase, sorts them, and collects the result into a list.

**Java 9 new Features and version enhancements**

Java 9 introduced several new features and enhancements, bringing improvements to the Java programming language, its libraries, and the JVM. The most notable feature is the **Java Platform Module System (JPMS)**, also known as **Project Jigsaw**, but there are many other enhancements as well.

Here is an overview of the major new features and enhancements in Java 9:

**1. Module System (Project Jigsaw)**

The **Java Platform Module System (JPMS)**, or **Project Jigsaw**, is the flagship feature of Java 9. It provides a way to organize code into reusable modules and encapsulates internal code. This resolves issues related to classpath hell and allows for more scalable, maintainable, and secure Java applications.

**Key Features:**

* **Modular JDK**: The JDK itself is now modular, allowing applications to include only the necessary parts of the JDK, resulting in smaller and faster runtime images.
* **module-info.java**: Introduced to define module dependencies and exports.

**Example**:

module com.example.myapp {

requires java.base;

exports com.example.myapp.service;

}

* **Encapsulation**: Only explicitly exported packages are accessible to other modules.

**Benefits:**

* Better encapsulation.
* Faster startup times and smaller memory footprints.
* More secure and maintainable Java applications.

**2. JShell (The REPL for Java)**

Java 9 introduced **JShell**, a Read-Eval-Print Loop (REPL) for Java. It allows developers to quickly test and evaluate Java code snippets without having to create a complete class or method.

**Key Features:**

* **Interactive Command Line Tool**: You can write Java code snippets and get immediate feedback.
* **Immediate Execution**: No need to compile the code and run a complete application.

**Example:**

jshell> int x = 10;

jshell> x + 5;

**Benefits:**

* Great for learning, testing, and experimenting with Java code.
* Useful for prototyping without the need for full program scaffolding.

**3. Factory Methods for Collections**

Java 9 introduces **convenience factory methods** for creating immutable lists, sets, and maps using the **List.of()**, **Set.of()**, and **Map.of()** methods.

**Key Features:**

* Immutable: The created collections are immutable.
* Concise: You no longer need to use Arrays.asList() or manually add elements.

**Example:**

List<String> list = List.of("A", "B", "C");

Set<Integer> set = Set.of(1, 2, 3);

Map<Integer, String> map = Map.of(1, "One", 2, "Two");

**Benefits:**

* Simplifies the creation of small, fixed collections.
* Enhances readability and reduces boilerplate code.

**4. Private Interface Methods**

Java 9 allows **private methods** in interfaces. These methods can be used to share code between default and static methods, improving code reuse within interfaces.

**Key Features:**

* **Private Methods in Interfaces**: These methods cannot be overridden by implementing classes and are only available to other methods in the interface.

**Example:**

interface MyInterface {

default void defaultMethod() {

helperMethod();

}

private void helperMethod() {

System.out.println("Helper Method");

}

}

**Benefits:**

* Code reuse within interfaces.
* Cleaner interface code by reducing code duplication.

**5. Stream API Enhancements**

The **Stream API** introduced in Java 8 has been further improved in Java 9. Several new methods have been added to make working with streams more powerful and concise.

**New Methods in Stream API:**

* **takeWhile(Predicate)**: Takes elements from the stream as long as the predicate is true.
* **dropWhile(Predicate)**: Drops elements while the predicate is true and processes the rest.
* **ofNullable(T element)**: Returns a stream of a single element if present, or an empty stream if null.
* **iterate(T seed, Predicate, UnaryOperator)**: An overload of the iterate method that supports a condition to stop.

**Example:**

Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7);

List<Integer> result = stream.takeWhile(n -> n < 5).collect(Collectors.toList()); // [1, 2, 3, 4]

Here are some examples of how to use the **Stream** methods dropWhile(), ofNullable(), and iterate() introduced in Java 8 and enhanced in later versions:

**1. dropWhile(Predicate) (Java 9):**

This method returns a stream that discards elements as long as the given predicate returns true. Once the predicate returns false, the remaining elements are included in the stream.

**Example:**

java

Copy code

import java.util.List;

import java.util.stream.Stream;

public class DropWhileExample {

public static void main(String[] args) {

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

// dropWhile: Drops numbers less than 5

numbers.stream()

.dropWhile(n -> n < 5)

.forEach(System.out::println); // Output: 5, 6, 7, 8, 9

}

}

In this example, dropWhile skips all elements that are less than 5 and includes the rest.

**2. Stream.ofNullable() (Java 9):**

This method returns a sequential Stream that contains a single element if the specified element is non-null. Otherwise, it returns an empty stream.

**Example:**

java

Copy code

import java.util.stream.Stream;

public class OfNullableExample {

public static void main(String[] args) {

String value = null;

// ofNullable: If the value is null, it returns an empty stream

Stream.ofNullable(value)

.forEach(System.out::println); // Output: (no output because it's null)

value = "Hello, World!";

Stream.ofNullable(value)

.forEach(System.out::println); // Output: Hello, World!

}

}

In this case, when value is null, Stream.ofNullable() results in an empty stream, avoiding a NullPointerException.

**3. Stream.iterate() (Enhanced in Java 9):**

In Java 8, Stream.iterate() was used to generate an infinite stream. Java 9 introduced a new overloaded version that takes a predicate to limit the stream generation.

**Example (Java 9 style with a predicate):**

import java.util.stream.Stream;

public class IterateExample {

public static void main(String[] args) {

// Generate numbers from 1 to 10 using iterate

Stream.iterate(1, n -> n <= 10, n -> n + 1)

.forEach(System.out::println); // Output: 1, 2, 3, ... 10

}

}

In this example, Stream.iterate() generates numbers starting from 1 and continues until the number exceeds 10.

**Example (Java 8 style infinite stream):**

import java.util.stream.Stream;

public class InfiniteIterateExample {

public static void main(String[] args) {

// Infinite stream of even numbers, limit it to 10 numbers

Stream.iterate(0, n -> n + 2)

.limit(10)

.forEach(System.out::println); // Output: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18

}

}

Here, Stream.iterate() generates an infinite stream of even numbers, but we use limit(10) to restrict the output to the first 10 elements.

**Summary of Methods:**

1. **dropWhile(Predicate)**: Skips elements as long as the predicate is true and then returns the rest of the elements.
2. **Stream.ofNullable(T)**: Creates a stream with a single element if the object is non-null, otherwise returns an empty stream.
3. **Stream.iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)**: Generates a stream by iterating a function until the predicate returns false.

**Benefits:**

* More control over stream processing.
* Enhanced flexibility in working with data streams.

**the internal representation of the String class:**

In **Java 9**, the internal representation of the String class was changed to store the underlying data in a **byte[]** array instead of a **char[]** array. This enhancement was part of the **Compact Strings** optimization introduced to improve memory efficiency, especially for strings that contain only Latin-1/ISO-8859-1 characters.

**Why the Change?**

Prior to Java 9, the String class stored characters in a char[], where each character was represented using 2 bytes (16 bits) because Java's char type uses UTF-16 encoding. This meant that even strings with single-byte characters (e.g., those using only ASCII or Latin-1) consumed 2 bytes per character.

In Java 9, the Compact Strings feature was introduced:

* If a string contains only Latin-1 characters (which can be represented in 1 byte), it uses a byte[] array with 1 byte per character.
* If a string contains characters that require 2 bytes (UTF-16 encoding), the string uses a byte[] array with 2 bytes per character.

This change resulted in significant memory savings for applications that handle a lot of Latin-1 strings.

**Key Changes in Java 9:**

1. **Internal Representation**: The string data is now stored in a byte[] instead of a char[].
2. **coder Field**: Along with the byte[], a coder field was introduced to determine whether the string is using a single-byte (Latin-1) or two-byte (UTF-16) representation.

**Example (Illustrating the Concept):**

java

Copy code

public class StringInternalRepresentation {

public static void main(String[] args) {

String asciiString = "Hello"; // Contains only Latin-1 characters

String unicodeString = "こんにちは"; // Contains UTF-16 characters

// Internally, 'asciiString' is stored using a byte array with 1 byte per character

// 'unicodeString' is stored using a byte array with 2 bytes per character

System.out.println("ASCII String: " + asciiString);

System.out.println("Unicode String: " + unicodeString);

}

}

**Benefits of Compact Strings:**

* **Memory Efficiency**: For strings containing only Latin-1 characters, the memory footprint is reduced by half.
* **Performance Improvement**: Operations involving strings that fit into the Latin-1 encoding can be more efficient due to smaller memory usage and reduced garbage collection overhead.

**Summary:**

* In **Java 9**, the String class was optimized to store its internal data in a **byte[]** instead of a **char[]** as part of the **Compact Strings** feature.
* This optimization improves memory usage for applications that deal primarily with Latin-1 encoded strings.

**6. CompletableFuture API Enhancements**

**CompletableFuture** was introduced in Java 8 to handle asynchronous programming. Java 9 introduces several enhancements and new methods to this API, making it more powerful for building asynchronous pipelines.

**New Methods:**

* **new CompletableFuture.completedFuture(T value)**: Returns a completed CompletableFuture.
* **orTimeout()**: Completes the future with a TimeoutException if it does not complete within a specified time.
* **completeOnTimeout()**: Completes the future with a specified value if it does not complete within the specified time.
* **copy()**: Returns a new CompletableFuture that has the same completion state.

**Benefits:**

* Simplifies handling of timeouts in asynchronous code.
* Easier to work with asynchronous programming patterns.

**7. Multi-Resolution Image API**

Java 9 introduces the **Multi-Resolution Image API**, which allows developers to work with images containing multiple resolutions. This is useful when you need to scale images based on the display resolution (like for different devices).

**Key Features:**

* MultiResolutionImage interface.
* Ability to get images of different resolutions from a single source.

**Example:**

Image multiResolutionImage = ...;

Image lowRes = multiResolutionImage.getResolutionVariant(16, 16);

**Benefits:**

* Better support for high-resolution displays.
* Efficient image rendering for multiple devices.

**8. Process API Enhancements**

The **Process API** has been enhanced in Java 9 to provide more control over and information about system processes.

**Key Features:**

* **ProcessHandle API**: Allows interaction with native processes, such as querying process IDs, states, start time, and CPU usage.
* **ProcessHandle.Info**: Provides detailed information about the process.

**Example:**

ProcessHandle currentProcess = ProcessHandle.current();

System.out.println("Process ID: " + currentProcess.pid());

**Benefits:**

* Easier to monitor and manage system processes from within Java code.

**9. HTTP/2 Client (Incubator)**

Java 9 introduced a new **HTTP/2 Client API** (in incubator module), replacing the old HttpURLConnection class. This new API supports HTTP/2 and WebSockets, allowing for more modern web communication.

**Key Features:**

* **HTTP/2 Support**: More efficient than HTTP/1.1, with features like multiplexing, server push, and header compression.
* **Asynchronous Operations**: Supports non-blocking, asynchronous requests.
* **WebSocket Support**: Full WebSocket support.

**Example:**

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create("http://example.com"))

.build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());

**Benefits:**

* Better performance for modern web communication.
* Cleaner, more flexible HTTP client code.

**10. Optional API Enhancements**

Java 9 improves the **Optional** class, adding new methods to make working with optionals more convenient.

**New Methods:**

* **ifPresentOrElse()**: Executes an action if a value is present, or runs an alternative action if not.
* **or()**: Returns an alternative optional if the value is not present.
* **stream()**: Converts an Optional into a stream.

**Example:**

Optional<String> optional = Optional.of("Hello");

optional.ifPresentOrElse(

System.out::println,

() -> System.out.println("Value is not present")

);

**Benefits:**

* More expressive handling of Optional values.
* Simplifies the handling of empty and non-empty optionals.

**11. Miscellaneous Changes**

* **Diamond Operator for Anonymous Classes**: Java 9 allows the diamond operator (<>) to be used in anonymous classes.
* **Improved Deprecation**: The @Deprecated annotation now includes attributes like forRemoval and since to indicate the version and removal plan.
* **StackWalker API**: A new API for stack walking and inspecting the call stack.
* **Multi-Version JARs**: Support for packaging multiple versions of classes targeting different Java versions in a single JAR.

**Java 11 new features and version enhancements: -**

**Stream.toList()** was introduced in **Java 10**. This method provides a more concise way to collect elements from a stream into an immutable List. Unlike Collectors.toList(), which creates a mutable list, Stream.toList() returns an unmodifiable list, ensuring that the list cannot be modified after it is created.

**Usage**

Here's an example of how to use Stream.toList():

java

Copy code

import java.util.stream.Stream;

public class StreamToListExample {

public static void main(String[] args) {

// Using Stream.toList() to collect elements into an immutable List

var list = Stream.of("Java", "Python", "JavaScript")

.toList();

System.out.println(list); // Output: [Java, Python, JavaScript]

// Attempting to modify the list will throw an UnsupportedOperationException

// list.add("C++"); // Uncommenting this line will cause an error

}

}

**Key Differences**

* **Collectors.toList()**: Returns a mutable List, which can be modified after creation.
* **Stream.toList()**: Returns an immutable List, which cannot be changed after it is created.

Java 11, released in September 2018, introduced several new features and enhancements to the Java platform. It was a **Long-Term Support (LTS)** release, making it one of the more important versions of Java in recent years. Many new APIs, deprecations, and changes were introduced to make Java more streamlined, efficient, and modern. Below is a summary of the most notable new features and version enhancements in Java 11:

**1. var in Lambda Parameters**

Java 11 extends the use of the **var keyword** to lambda expressions. Previously introduced in Java 10, var could only be used in local variable declarations. In Java 11, it can now be used in lambda parameters to infer types.

**Example:**

(var x, var y) -> x + y

**Benefits:**

* More concise syntax and code readability in lambda expressions.
* Enables annotations to be added to lambda parameters.

**2. HTTP Client (Standard)**

The new **HTTP Client API** that was incubated in Java 9 and improved in Java 10 is now standardized in Java 11. It supports both **HTTP/1.1** and **HTTP/2**, making it more modern and efficient for web communication, and provides an alternative to HttpURLConnection.

**Key Features:**

* **Asynchronous Requests**: Supports non-blocking operations.
* **WebSockets Support**: Full support for WebSocket communication.
* **HTTP/2 Support**: Allows multiplexing and better performance than HTTP/1.1.

**Example:**

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create("https://example.com"))

.build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());

**Benefits:**

* Simplifies HTTP communication.
* Provides better performance with modern protocols.

**3. New String Methods**

Several new methods have been added to the String class in Java 11 to make string manipulation easier and more concise.

**New String Methods:**

* **isBlank()**: Checks if a string contains only whitespace or is empty.
* **strip()**: Removes leading and trailing whitespace (similar to trim() but more Unicode-aware).
* **stripLeading()**: Removes leading whitespace.
* **stripTrailing()**: Removes trailing whitespace.
* **lines()**: Returns a stream of lines extracted from the string.
* **repeat(int count)**: Repeats the string a specified number of times.

**Example:**

public class LinesExample {

public static void main(String[] args) {

String str = " Hello World ";

System.out.println(str.isBlank()); // false

System.out.println(str.strip()); // "Hello World"

System.out.println(str.repeat(3)); // " Hello World Hello World Hello World "

String multiLineString = "Hello, World!\nWelcome to Java.\nEnjoy coding!";

// Using lines() to process each line in the string

multiLineString.lines()

.forEach(System.out::println);

}

}

**Output**

css

Copy code

Hello, World!

Welcome to Java.

Enjoy coding!

**Benefits:**

* Simplifies common string operations.
* Improves performance for whitespace handling and string repetition.

**4. Files.readString() and Files.writeString()**

Java 11 introduced methods to simplify reading and writing strings from/to files. These methods help reduce boilerplate code and make working with files more intuitive.

**Example:**

Path path = Paths.get("example.txt");

// Write a string to the file

Files.writeString(path, "Hello, World!");

// Read the content of the file

String content = Files.readString(path);

System.out.println(content); // Outputs: Hello, World!

**Benefits:**

* Simplifies file I/O operations.
* Provides more concise and readable code for reading and writing text files.

**5. Optional Enhancements**

Java 11 introduces new methods to the **Optional** class, making it easier to work with optionals.

**New Optional Methods:**

* **isEmpty()**: Returns true if the optional is empty, false otherwise (complementing isPresent()).
* **stream()**: Converts an Optional into a stream.

**Example:**

Optional<String> optional = Optional.of("Hello");

optional.ifPresentOrElse(System.out::println, () -> System.out.println("Value not present"));

**Benefits:**

* Enhances the usability of Optional by making it easier to handle empty values.

**6. Local-Variable Syntax for Lambda Parameters**

As part of the var enhancement, Java 11 allows **local-variable syntax** (using var) in the parameters of lambda expressions, making lambda parameter declaration more consistent with local variable declarations.

**Example:**

java

Copy code

List<String> list = List.of("a", "b", "c");

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

**Benefits:**

* Consistency in the usage of the var keyword across different contexts.
* Enables adding annotations to lambda parameters.

**7. Deprecation and Removal of Features**

Several APIs and features have been deprecated or removed in Java 11 as part of the effort to clean up the JDK and modernize the platform.

**Key Removals:**

* **Java EE (JAX-WS, JAXB)**: Java 11 removed modules related to Java EE technologies like JAX-WS and JAXB, which are now available as separate libraries outside of the JDK.
* **Applet API**: The Applet API was deprecated in Java 9 and is removed in Java 11.
* **java.se.ee Module**: This module was deprecated and removed, which includes packages for things like CORBA, JAX-WS, and other EE technologies.

**Key Deprecations:**

* **Nashorn JavaScript Engine**: The Nashorn engine, introduced in Java 8, is deprecated and marked for removal in future releases.

**Benefits:**

* Reduces the size of the JDK by removing outdated and less-used features.
* Encourages the use of more modern and modular alternatives.

**8. var with Anonymous Inner Classes**

Java 11 allows the use of var in anonymous inner classes, making it more consistent with its use in lambda expressions and local variable declarations.

**Example:**

Consumer<String> consumer = new Consumer<>() {

@Override

public void accept(String t) {

System.out.println(t);

}

};

**9. Epsilon Garbage Collector (Experimental)**

Java 11 introduced the **Epsilon Garbage Collector**, a **no-op garbage collector** that handles memory allocation without reclaiming memory. This is useful for performance testing or applications where manual memory management is implemented.

**Key Features:**

* **No-op GC**: Allocates memory without automatic garbage collection.

**Benefits:**

* Useful for performance testing.
* Allows developers to control memory management explicitly in certain scenarios.

**10. Flight Recorder**

Java Flight Recorder (JFR), a low-overhead, high-performance monitoring and diagnostics tool, was made open-source and included as part of the JDK in Java 11. JFR collects data on application performance, latency, and memory usage.

**Key Features:**

* **Low overhead**: Minimal impact on application performance.
* **Detailed analysis**: Offers insights into JVM behavior for performance tuning.

**Benefits:**

* Improves the ability to monitor and debug production applications.

**11. ZGC: A Low-Latency Garbage Collector (Experimental)**

Java 11 introduced **Z Garbage Collector (ZGC)**, an experimental garbage collector designed for **low-latency** applications. It can handle large heaps (multi-terabyte) with very short pause times.

**Key Features:**

* **Low-latency**: Pause times do not exceed 10ms.
* **Concurrent**: Most of the garbage collection happens concurrently with the application.

**Benefits:**

* Ideal for systems that require minimal garbage collection pause times, such as real-time applications or large-scale enterprise systems.

**12. Unicode 10 Support**

Java 11 brings support for **Unicode 10**, which includes new characters and scripts.

**13. Nested try-with-resources Improvements**

In Java 11, you no longer need to declare variables explicitly inside the try-with-resources block. You can reuse variables that are effectively final.

**Example:**

BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

try (reader) {

System.out.println(reader.readLine());

}

**Benefits:**

* Simplifies code by allowing you to reuse existing resources in try-with-resources.

**Java 15 new features and version enhancements**

Java 15, released in September 2020, brought a range of new features, improvements, and updates that continued to modernize and evolve the Java language and platform. Below is a detailed overview of the most important features and enhancements introduced in Java 15:

**1. Sealed Classes (Preview)**

Java 15 introduced **sealed classes** as a preview feature, allowing developers to restrict which other classes or interfaces can extend or implement a given class or interface. This provides more control over inheritance and can help with maintaining class hierarchies in a more controlled manner.

**Key Features:**

* Sealed classes control which classes can extend them using the permits clause.
* Only the classes mentioned in the permits clause can extend the sealed class.
* A sealed class can only be extended by **final classes**, **non-sealed classes**, or **sealed subclasses**.

**Example:**

public abstract sealed class Shape permits Circle, Rectangle {

// Shape class code

}

final class Circle extends Shape {

// Circle class code

}

final class Rectangle extends Shape {

// Rectangle class code

}

**Benefits:**

* More control over class hierarchies.
* Improved encapsulation and maintainability of code.
* Helps avoid incorrect usage of inheritance.

**2. Records (Second Preview)**

Java 15 continued the preview of **Records**, which were initially introduced in Java 14. Records provide a concise way to declare classes that are mainly used to hold data. They are immutable by design and automatically generate boilerplate code like getters, equals(), hashCode(), and toString().

**Key Features:**

* **Compact syntax** for data classes.
* **Immutability**: Fields are final by default.
* **Generated methods**: Constructors, equals(), hashCode(), toString(), and accessors are auto-generated.

**Example:**

java

Copy code

public record Point(int x, int y) {}

**Benefits:**

* Reduces boilerplate code for data-carrying classes.
* Enforces immutability by default.
* Improves code readability and maintainability.

**3. Text Blocks (Standard)**

**Text blocks** were made a standard feature in Java 15 after being a preview feature in previous versions. They provide a way to write multi-line strings more cleanly and concisely, improving readability and reducing the need for escape sequences.

**Key Features:**

* **Multi-line string literals** enclosed in triple quotes (""").
* Preserves line breaks and indentation.
* Eliminates the need for escaping quotes or newlines within string literals.

**Example:**

java

Copy code

String json = """

{

"name": "John",

"age": 30

}

""";

**Benefits:**

* Cleaner syntax for multi-line strings.
* Easier to maintain and more readable, especially for embedding HTML, JSON, SQL, etc.

**4. Pattern Matching for instanceof (Second Preview)**

This feature simplifies the instanceof pattern by allowing the **pattern matching** directly within the instanceof operator, reducing the need for explicit type casting. This was introduced as a second preview in Java 15 after being initially previewed in Java 14.

**Key Features:**

* **Pattern Matching** in instanceof checks.
* Automatically casts the variable to the desired type upon a successful match.

**Example:**

if (obj instanceof String s) {

System.out.println(s.toUpperCase()); // No need for explicit cast

}

**Benefits:**

* Cleaner and more concise code.
* Reduces the likelihood of casting errors and improves code readability.

**5. Hidden Classes**

**Hidden classes** are classes that cannot be directly used by the bytecode of other classes but can be used dynamically by frameworks and libraries. These classes are intended for use by **dynamic language runtimes** and **frameworks** that need to define classes at runtime.

**Key Features:**

* Hidden classes are **non-discoverable** and **cannot be referenced** directly by other classes.
* Useful for generating proxy classes or dynamically loading classes at runtime (common in frameworks like Spring, Hibernate, etc.).

**Example Use Cases:**

* Dynamic proxy classes generated at runtime.
* Implementation of language features for dynamic JVM languages.

**Benefits:**

* Better support for dynamic languages and frameworks.
* Enhances security by making classes inaccessible for direct use.

**6. ZGC (Z Garbage Collector) Improvements**

Java 15 continued improving the **Z Garbage Collector (ZGC)**, a low-latency garbage collector designed for handling large heaps (multi-terabyte) with minimal impact on application throughput and responsiveness. In Java 15, **ZGC becomes a production-ready feature**.

**Key Improvements:**

* **Concurrent class unloading**: ZGC now supports unloading classes concurrently without stopping the application.
* **Improved performance**: Further optimizations to improve memory usage and overall performance.

**Benefits:**

* Ideal for applications requiring low-latency garbage collection.
* Scales well with large heap sizes, improving responsiveness.

**7. Shenandoah Garbage Collector (Production-Ready)**

The **Shenandoah GC**, an experimental low-pause-time garbage collector introduced earlier, is now made **production-ready** in Java 15. Shenandoah GC focuses on reducing GC pause times by performing most of its work concurrently with application threads.

**Key Features:**

* **Low-pause times** by performing more garbage collection work concurrently.
* Optimized for applications requiring high throughput and minimal interruption.

**Benefits:**

* Reduces garbage collection pauses, making it suitable for low-latency applications.
* Competes with ZGC for use cases involving large heaps and low pause times.

**8. Foreign-Memory Access API (Second Incubator)**

Java 15 continues the development of the **Foreign-Memory Access API**, which was first introduced in Java 14 as an incubator feature. This API allows Java programs to access memory outside of the Java heap in a safe and efficient manner.

**Key Features:**

* **Direct memory access** without needing to use Unsafe.
* Provides the ability to **allocate**, **access**, and **manipulate** foreign memory (native memory).

**Use Cases:**

* Working with off-heap memory for performance optimization.
* Interfacing with native code and memory-mapped files.

**Benefits:**

* Allows for more efficient use of memory outside the Java heap.
* Opens up better interoperation between Java and native applications.

**9. Edwards-Curve Digital Signature Algorithm (EdDSA)**

Java 15 introduced support for the **Edwards-Curve Digital Signature Algorithm (EdDSA)**, a modern public-key signature algorithm known for its performance and security features.

**Key Features:**

* Support for **Ed25519** and **Ed448** signature schemes.
* **Faster and more secure** than many existing algorithms.

**Benefits:**

* Provides modern cryptographic capabilities to Java applications.
* Ideal for applications requiring efficient and secure digital signatures.

**10. Deprecation and Removal of Features**

As with every major release, Java 15 deprecated or removed outdated features and APIs to modernize the language and platform.

**Key Changes:**

* **Nashorn JavaScript Engine**: The Nashorn JavaScript engine, which was deprecated in Java 11, is now removed in Java 15.
* **RMI Activation**: The **RMI Activation** framework, which was seldom used, is deprecated for removal in a future release.

**Benefits:**

* Keeps the JDK leaner and more focused on modern and widely used features.
* Encourages developers to adopt more up-to-date alternatives.

**11. Miscellaneous Enhancements**

**JEP 372: Remove the Nashorn JavaScript Engine**

* The **Nashorn JavaScript engine** was removed from the JDK in Java 15 as it was no longer actively maintained or widely used.

**JEP 371: Value-Based Classes**

* **Value-based classes** (like Optional, LocalDateTime, etc.) are now explicitly marked and documented as value-based, helping developers use them correctly and avoid anti-patterns.

**Conclusion**

Java 15 introduced several key features like **sealed classes**, **text blocks**, and **records**, along with improvements to garbage collection (ZGC and Shenandoah), **pattern matching**, and the **Foreign-Memory Access API**. It continued to refine Java's modern features while also deprecating and removing outdated components, making Java a more secure, high-performance, and developer-friendly platform.

**Java 16 new features and version enhancements**

Java 16, released in March 2021, brought new features, enhancements, and improvements to the Java platform, continuing its evolution with a focus on developer productivity, performance, and language improvements. Below is an overview of the most important features and enhancements introduced in Java 16:

**1. Records (Standard)**

Records, which were a preview feature in earlier releases (Java 14 and Java 15), became a standard feature in Java 16. They provide a concise way to define classes whose main purpose is to hold immutable data.

**Key Features:**

* **Immutable data holders**: Fields are implicitly final and cannot be modified after creation.
* **Generated methods**: The compiler automatically generates methods like equals(), hashCode(), and toString() for records.
* **Compact syntax**: A more compact syntax than traditional classes.

**Example:**

public record Point(int x, int y) {}

**Benefits:**

* Reduces boilerplate code for data classes.
* Enforces immutability by default, making records more reliable and easy to maintain.

**2. Pattern Matching for instanceof (Standard)**

Pattern matching for the instanceof operator became a standard feature in Java 16 after being previewed in Java 14 and Java 15. It simplifies the use of instanceof by combining the type check and cast into a single step.

**Key Features:**

* Automatically casts the variable to the desired type if the instanceof check passes.
* Reduces the need for explicit type casting.

**Example:**

if (obj instanceof String s) {

System.out.println(s.toUpperCase()); // No explicit cast needed

}

**Benefits:**

* More concise and readable code.
* Reduces potential for casting errors.

**3. Sealed Classes (Second Preview)**

Sealed classes, which were introduced as a preview feature in Java 15, received a second preview in Java 16. Sealed classes allow developers to control which classes or interfaces can extend or implement them, providing a more controlled inheritance hierarchy.

**Key Features:**

* Use the sealed modifier to define which classes can extend or implement a class or interface.
* A class can be **sealed**, **final**, or **non-sealed**.
* The permits clause specifies allowed subclasses.

**Example:**

java

Copy code

public abstract sealed class Shape permits Circle, Rectangle {}

final class Circle extends Shape {}

final class Rectangle extends Shape {}

**Benefits:**

* Helps enforce strict class hierarchies and prevents unintended inheritance.
* Provides better control over how APIs are extended or used.

**4. Vector API (Incubator)**

Java 16 introduced the **Vector API** as an incubator feature. This API allows developers to perform vector computations (operations that can be applied to arrays of data) more efficiently by utilizing hardware-level SIMD (Single Instruction, Multiple Data) instructions.

**Key Features:**

* **Vector operations**: Efficient processing of data arrays in parallel.
* **SIMD instructions**: Allows Java to take advantage of hardware-level SIMD instructions for better performance.

**Example:**

java

Copy code

Vector<Integer> vector = IntVector.fromArray(IntVector.SPECIES\_256, new int[] {1, 2, 3, 4}, 0);

Vector<Integer> result = vector.add(10); // Adds 10 to each element

**Benefits:**

* Significant performance improvements for tasks that can be parallelized.
* Enables developers to write high-performance code for scientific and data processing applications.

**5. Foreign Linker API (Incubator)**

The **Foreign Linker API** was introduced as an incubator feature in Java 16. This API simplifies the process of linking and interfacing with native code and libraries (such as C or C++ libraries) from Java.

**Key Features:**

* Provides a pure Java API for linking and invoking native functions.
* Helps simplify the process of working with native libraries, reducing the need for JNI (Java Native Interface).

**Benefits:**

* Allows for more efficient integration between Java and native code.
* Reduces the complexity of interacting with native libraries, making it easier for developers to work with native functions.

**6. Foreign-Memory Access API (Third Incubator)**

The **Foreign-Memory Access API**, introduced as an incubator in earlier releases, continued its evolution in Java 16. This API allows Java programs to safely and efficiently access memory outside the Java heap (foreign memory), such as native memory.

**Key Features:**

* Safe and efficient access to off-heap memory.
* Helps avoid using unsafe memory management practices.

**Benefits:**

* Provides better performance and flexibility for memory-intensive applications.
* Ideal for high-performance systems that require large amounts of memory (e.g., databases, game engines).

**7. Strong Encapsulation for JDK Internals by Default**

With Java 16, the JDK internals are strongly encapsulated by default, preventing access to internal APIs that are not meant for public use. This is part of an ongoing effort to enforce stronger modularity in the JDK, which started in Java 9 with Project Jigsaw.

**Key Features:**

* Strong encapsulation of internal APIs (sun.\*, com.sun.\* packages) by default.
* Programs that depend on these internals will need to migrate to supported APIs or use command-line flags to access them.

**Benefits:**

* Improves the security and maintainability of the JDK.
* Encourages developers to use stable, supported APIs rather than relying on internal, unstable ones.

**8. Packaging Tool**

Java 16 introduced a new **Packaging Tool** that allows developers to package Java applications as native installers (such as .msi, .deb, .pkg, etc.) for various platforms (Windows, Linux, macOS). The tool was initially introduced as an incubator in Java 14 and became a standard feature in Java 16.

**Key Features:**

* Generates native installers for Java applications.
* Supports packaging Java applications into platform-specific formats (e.g., .msi, .pkg, .deb).

**Example:**

bash

Copy code

jpackage --name MyApp --input target/ --main-jar MyApp.jar --type pkg

**Benefits:**

* Makes it easier to distribute Java applications with native installers.
* Streamlines deployment and installation on various operating systems.

**9. ZGC (Z Garbage Collector) Enhancements**

Java 16 continued to improve the **Z Garbage Collector (ZGC)**, which is designed for low-latency applications. Enhancements in this release include better memory utilization and performance improvements.

**Key Features:**

* **Concurrent thread-stack processing**: ZGC now processes thread stacks concurrently, further reducing pauses.
* **Improved NUMA awareness**: Better handling of memory on multi-processor systems with Non-Uniform Memory Access (NUMA).

**Benefits:**

* Lower garbage collection pause times.
* Improved performance and scalability for applications with large heaps.

**10. Other Miscellaneous Features and Enhancements**

**Unix-Domain Socket Channels (JEP 380)**

Java 16 introduced support for **Unix-domain socket channels** on Unix-like systems. Unix-domain sockets are used for inter-process communication (IPC) and are faster and more secure than network-based socket communication for processes running on the same machine.

**Example:**

Path socketPath = Paths.get("/tmp/mysocket");

UnixDomainSocketChannel channel = UnixDomainSocketChannel.open(socketPath);

**Benefits:**

* Provides faster and more secure communication between processes on Unix-like systems.
* Useful for applications that require fast local IPC.

**Alpine Linux Support**

Java 16 introduced **Alpine Linux support** for the JDK. Alpine Linux is commonly used in Docker containers due to its small footprint. Java 16’s support for Alpine Linux helps developers build more lightweight Java applications using Docker.

**Benefits:**

* Reduces the size of Docker images by leveraging the lightweight Alpine Linux distribution.
* Improves containerization support for Java applications.

**Vector API and Foreign-Memory API: Both the Vector API and the Foreign-Memory Access API continue to be incubated and refined in Java 16, making progress toward becoming stable features in future releases.**

**Java 17 new features and version enhancements with example**

Java 17, released in September 2021, is a Long-Term Support (LTS) version, meaning it will receive extended support from Oracle. It brought a number of significant improvements, new features, and enhancements that continue to refine the language and platform. Here’s a comprehensive overview of the key features and enhancements introduced in Java 17, along with examples:

**1. Sealed Classes (Standard)**

**Sealed classes**, which were in preview in previous releases, are now a standard feature in Java 17. Sealed classes allow developers to control which other classes or interfaces can extend or implement them.

**Key Features:**

* Use the sealed modifier to define classes or interfaces with restricted subclassing.
* Classes extending a sealed class must be marked as final, sealed, or non-sealed.
* The permits clause explicitly lists the allowed subclasses.

**Example:**

public abstract sealed class Shape permits Circle, Rectangle {}

public final class Circle extends Shape {

// Circle implementation

}

public final class Rectangle extends Shape {

// Rectangle implementation

}

**Benefits:**

* Improves maintainability by controlling inheritance hierarchies.
* Provides clearer APIs by explicitly stating subclassing rules.

**2. Pattern Matching for switch (Preview)**

Java 17 introduced **Pattern Matching for switch** as a preview feature, allowing more powerful and expressive switch statements. It enables pattern matching directly within switch expressions.

**Key Features:**

* Match patterns and types within switch cases.
* Handles both traditional values and types.

**Example:**

static String formatterPatternSwitch(Object obj) {

return switch (obj) {

case Integer i -> String.format("int %d", i);

case Long l -> String.format("long %d", l);

case Double d -> String.format("double %f", d);

case String s -> String.format("String %s", s);

default -> obj.toString();

};

}

**Benefits:**

* Simplifies code by allowing type-based pattern matching in switch.
* Makes the switch more expressive and powerful for handling various types.

**3. Strong Encapsulation of JDK Internals (JEP 403)**

In Java 17, **strong encapsulation of internal APIs** is enforced by default, meaning it is no longer possible to access internal JDK APIs using reflection without explicitly enabling it. This continues the trend started in Java 9 with Project Jigsaw.

**Key Features:**

* JDK internals are no longer accessible by default.
* Access to JDK internals can be enabled using command-line flags, but this is discouraged.

**Benefits:**

* Improves security by preventing unintended access to internal JDK APIs.
* Encourages developers to use standard, supported APIs.

**4. Context-Specific Deserialization Filters (JEP 415)**

Java 17 introduced **context-specific deserialization filters**, which enhance the security of deserialization operations by allowing developers to apply filters at runtime to control the classes that can be deserialized.

**Key Features:**

* Allows setting filters on deserialization to specify which classes are allowed or denied.
* Filters can be applied at the **context** level, enabling more fine-grained control over deserialization.

**Example:**

ObjectInputStream ois = new ObjectInputStream(inputStream);

ois.setObjectInputFilter(filter);

Object obj = ois.readObject();

**Benefits:**

* Improves security by preventing deserialization of untrusted or malicious classes.
* Helps mitigate vulnerabilities associated with unsafe deserialization.

**5. Foreign Function & Memory API (Incubator)**

Java 17 continues to incubate the **Foreign Function & Memory API**, which allows Java programs to interact with native code and memory safely and efficiently. This API provides a safe, pure-Java mechanism for calling native libraries (similar to JNI) and accessing off-heap memory.

**Key Features:**

* Access foreign memory (native memory) safely without using Unsafe.
* Interoperate with native code (C, C++) without writing JNI code.

**Example:**

try (var session = MemorySession.openConfined()) {

var nativeMemory = session.allocate(4);

nativeMemory.set(ValueLayout.JAVA\_INT, 0, 42);

int value = nativeMemory.get(ValueLayout.JAVA\_INT, 0);

System.out.println(value); // Prints 42

}

**Benefits:**

* Simplifies working with native memory and functions.
* Provides safer and more maintainable access to foreign memory compared to Unsafe or JNI.

**6. MacOS/AArch64 Port (JEP 391)**

Java 17 adds support for running the JDK on **Apple Silicon (AArch64)**, allowing Java applications to run natively on Mac devices with the new ARM-based architecture (M1 chips).

**Key Features:**

* Adds native support for **AArch64** on **macOS** platforms.
* Provides performance benefits by running natively on Apple Silicon rather than through emulation.

**Benefits:**

* Improves performance and efficiency on Apple Silicon devices.
* Ensures the JDK remains compatible with future Apple hardware.

**7. Deprecating the Applet API (JEP 398)**

Java 17 officially deprecates the **Applet API** for removal in a future release. Applets, which were once widely used for web-based applications, have become obsolete with the discontinuation of browser support for Java plug-ins.

**Key Features:**

* The **Applet API** is deprecated and marked for future removal.
* Developers are encouraged to migrate any remaining applets to modern web technologies.

**Benefits:**

* Removes outdated technology that is no longer in use.
* Encourages developers to adopt modern alternatives like JavaFX or web applications.

**8. Vector API (Second Incubator)**

The **Vector API**, which allows developers to express vector computations in Java, is further incubated in Java 17. This API leverages SIMD (Single Instruction, Multiple Data) hardware instructions to improve performance in vectorized operations.

**Key Features:**

* Allows efficient computation on vectors of data using hardware-level SIMD instructions.
* Improves performance for data-heavy applications (e.g., scientific computing, image processing).

**Example:**

IntVector a = IntVector.fromArray(IntVector.SPECIES\_256, new int[] {1, 2, 3, 4, 5, 6, 7, 8}, 0);

IntVector b = IntVector.fromArray(IntVector.SPECIES\_256, new int[] {1, 1, 1, 1, 1, 1, 1, 1}, 0);

IntVector c = a.add(b); // Vectorized addition

**Benefits:**

* Significant performance improvements for workloads that can leverage vector operations.
* Brings high-performance computing closer to the JVM without relying on native libraries.

**9. Other Miscellaneous Features**

**Deprecating the Security Manager for Removal (JEP 411)**

* The **Security Manager**, which has been part of Java since the 1990s, is deprecated for removal in a future version of Java. It was designed to provide a security sandbox for running untrusted code but has become less relevant in modern Java applications.

**Removal of the Experimental AOT and JIT Compiler (JEP 410)**

* Java 17 removes the **Ahead-of-Time (AOT)** and **Just-in-Time (JIT) Compiler** that were introduced as experimental features in earlier versions. These compilers were not widely adopted and are now removed to simplify the JDK.

**Other Small Enhancements**

* **NullPointerException** messages improved, making it easier to identify what caused the exception.
* Improved **macOS rendering pipeline** for better performance and compatibility.

**Conclusion**

Java 17 brings a wide range of new features and enhancements, from **sealed classes** and **pattern matching for switch** to further developments in the **Foreign Function & Memory API** and **Vector API**. As a Long-Term Support (LTS) release, Java 17 is designed to provide stability and longevity for production environments, making it a critical upgrade for developers and organizations looking for long-term support and feature improvements.

With new language features like **sealed classes** and the continued refinement of performance-enhancing APIs like the **Vector API** and **Foreign Memory API**, Java 17 offers developers both modern capabilities and improved security and performance.

**Java 21 new features and version enhancements with example**

Java 21, released in September 2023, is another Long-Term Support (LTS) version of the Java platform. It introduces a variety of new features, enhancements, and improvements aimed at making Java more powerful, flexible, and easier to use. Below is an overview of the key features and enhancements introduced in Java 21, along with examples where applicable.

**1. New Language Features**

**1.1. Pattern Matching for Switch (Standard)**

Pattern matching for switch expressions, previously in preview, is now a standard feature in Java 21. This enhancement allows for more expressive and concise switch statements that can match complex patterns.

**Example:**

public static String formatShape(Shape shape) {

return switch (shape) {

case Circle c -> "Circle with radius: " + c.radius();

case Rectangle r -> "Rectangle with width: " + r.width() + " and height: " + r.height();

default -> "Unknown shape";

};

}

**1.2. Record Patterns (Preview)**

Record patterns provide a way to destructure record types directly within switch expressions or instanceof checks, enhancing the readability and expressiveness of the code.

**Example:**

java

Copy code

public static String describeShape(Object obj) {

return switch (obj) {

case Point(int x, int y) -> "Point at (" + x + ", " + y + ")";

case Circle(int radius) -> "Circle with radius " + radius;

default -> "Unknown shape";

};

}

**1.3. Sequence Patterns (Preview)**

Sequence patterns allow matching and destructuring of sequences (e.g., lists and arrays) directly in pattern matching. This makes it easier to work with collections.

**Example:**

public static String analyzeList(List<Object> list) {

return switch (list) {

case List<Integer> integers && integers.size() > 2 -> "List of integers with more than 2 elements.";

case List<String> strings -> "List of strings.";

default -> "Unknown type.";

};

}

**2. Virtual Threads (Preview)**

Java 21 introduces **Virtual Threads** as a preview feature, allowing developers to create lightweight, concurrent threads that can scale efficiently. Virtual threads are designed to simplify the development of high-throughput applications while reducing the complexity of managing traditional threads.

**Key Features:**

* Virtual threads can be created easily using the new Thread.ofVirtual() method.
* They can handle a massive number of concurrent tasks with minimal overhead.

**Example:**

Runnable task = () -> {

System.out.println("Running in a virtual thread: " + Thread.currentThread().getName());

};

Thread.startVirtualThread(task);

**Benefits:**

* Reduces the complexity of asynchronous programming.
* Enables high concurrency with low memory overhead.

**3. Foreign Function & Memory API (Standard)**

The **Foreign Function & Memory API** is now a standard feature, enabling Java programs to interact with native code and memory safely and effectively. This API provides a mechanism for invoking native functions and accessing memory outside the Java heap.

**Key Features:**

* Allows safe access to foreign memory using a new memory access API.
* Provides a cleaner and more efficient way to call native functions.

**Example:**

try (var session = MemorySession.openConfined()) {

var buffer = session.allocate(4);

buffer.set(ValueLayout.JAVA\_INT, 0, 42);

int value = buffer.get(ValueLayout.JAVA\_INT, 0);

System.out.println(value); // Prints 42

}

**Benefits:**

* Simplifies integration with native libraries.
* Enhances performance by allowing efficient access to off-heap memory.

**4. Standard Library Enhancements**

**4.1. Enhanced String Methods**

Java 21 introduces several new methods to the String class, making string manipulation more straightforward.

**New Methods:**

* String::repeat(int count): Repeats the string a specified number of times.
* String::stripIndent(): Removes the common leading whitespace from each line.
* String::translateEscapes(): Translates Unicode escape sequences in strings.

**Example:**

java

Copy code

String repeated = "Java".repeat(3); // "JavaJavaJava"

String indented = """

Hello

World

""".stripIndent(); // "Hello\n World"

String escaped = "Line1\nLine2\\u003A".translateEscapes(); // "Line1\nLine2:"

**4.2. HttpClient Enhancements**

The HttpClient API has received improvements, including support for HTTP/3 and additional methods for handling responses and requests more effectively.

**Example:**

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create("https://example.com"))

.header("Accept", "application/json")

.build();

client.sendAsync(request, HttpResponse.BodyHandlers.ofString())

.thenAccept(response -> System.out.println(response.body()));

**5. Deprecations and Removals**

**5.1. Removal of the Security Manager**

Java 21 officially removes the Security Manager, which has been deprecated for removal in previous versions. This change reflects a shift towards more modern security practices.

**5.2. Deprecation of the Applet API**

The Applet API remains deprecated, emphasizing the need for modern alternatives for web applications.

**6. Miscellaneous Enhancements**

**6.1. New instanceof Simplifications**

The instanceof operator can now use type patterns, allowing developers to check the type and cast in a single step.

**Example:**

if (obj instanceof String s) {

System.out.println(s.toUpperCase()); // Implicitly casts to String

}

**6.2. Performance Improvements**

Java 21 includes various performance improvements across the JVM and libraries, enhancing startup time, garbage collection, and overall runtime performance.

**Conclusion**

Java 21 introduces numerous enhancements aimed at improving developer productivity, code readability, and performance. Features like **Virtual Threads** simplify concurrent programming, while enhancements to pattern matching and the Foreign Function & Memory API make working with external systems more straightforward and efficient.

**Java 23 version enhancements**

JDK 17 binaries are free to use in production and free to redistribute, at no cost, under the [Oracle No-Fee Terms and Conditions](https://www.java.com/freeuselicense) (NFTC).

Java is the world's most prolific programming language and development platform. Oracle Java SE Universal Subscription offers the most complete support and advanced management features available today. Protect your enterprise investment with security updates and world-class support from the stewards of Java. Now the subscription includes entitlement to Oracle GraalVM, Java SE Subscription Enterprise Performance Pack, and Java Management Service, an Oracle Cloud service that monitors and manages all your Java deployments.

**Java 23 is now available**

The next Java release improves the performance, stability, and security of Java application development.

JDK 23 binaries are free to use in production and free to redistribute, at no cost, under the [Oracle No-Fee Terms and Conditions](https://www.java.com/freeuselicense) (NFTC).

JDK 23 will receive updates under these terms, until March 2025, when it will be superseded by JDK 24.

**what is the java jdk latest release**

The latest release of Java is **JDK 23**, launched in September 2024. This version brings several improvements in terms of language features, performance, and security.

Notable features of JDK 23 include:

1. **Primitive Types in Patterns**: This enhancement simplifies pattern matching for primitive types, making it easier for developers to work with them across instanceof and switch contexts.
2. **Graal Just-In-Time (JIT) Compiler**: Integrated directly into the Oracle JDK, providing faster startup times and improving performance for applications that generate large amounts of garbage early on.
3. **Structured Concurrency**: This helps developers manage multithreaded code more efficiently by grouping related tasks into single units of work.
4. **Markdown Support in Javadoc**: JDK 23 allows developers to write documentation comments using Markdown, making it easier to maintain readable API documentation.
5. **Z Garbage Collector (ZGC)**: Enabled by default in generational mode, it improves efficiency by reducing the resource cost associated with garbage collection.

These improvements make JDK 23 a powerful tool for modern Java applications, with particular focus on performance, usability, and maintaining backward compatibility.

1. **Primitive Types in Patterns**

is a feature introduced as a **preview** in Java 23 under **Project Amber**. This feature enhances pattern matching, instanceof, and switch by extending their capabilities to work seamlessly with **primitive types**.

**Key Highlights:**

1. **Pattern Matching for Primitive Types**: Previously, pattern matching in Java worked well for reference types (like String, Object, etc.) but had limitations when dealing with primitive types (like int, double, etc.). With this enhancement, you can now use pattern matching with primitive types in all pattern contexts.

Example:

if (x instanceof int i) {

System.out.println(i \* 2); // x is now matched and cast as an int

}

1. **Switch Expression Support for Primitives**: Java’s switch expressions can now handle primitive patterns, improving the usability of switch for conditions involving primitive values.

most exciting updates in **JDK 23** is the enhancement of **primitive type patterns** in ***instanceof***and ***switch***statements. This improvement allows developers to perform **pattern matching** with **primitive types**, making the code more expressive and efficient. By allowing **primitive types** to be used directly in these statements, developers can write cleaner and more intuitive code.

**Code Example:**

Object obj = 42;

**switch** (obj) {

**case** Integer i -> System.out.println("Integer: " + i);

**case** Long l -> System.out.println("Long: " + l);

**default** -> System.out.println("Other: " + obj);

}

Example:

switch (x) {

case int i -> System.out.println("Integer: " + i);

case double d -> System.out.println("Double: " + d);

default -> System.out.println("Other type");

}

**3. Simplified Module Imports**

The**new module import declarations** in **JDK 23**simplify module management by allowing modules to declare dependencies more flexibly. This feature is particularly useful for large projects, enabling finer control over module dependencies and improving the[**modularization of Java applications**](https://www.geeksforgeeks.org/modular-approach-in-programming/). By allowing the imports statement in module declarations, developers can manage dependencies in a more organized way.

**import** **module** java.base;

**public** **class** **Example** {

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

Map<Character, List<String>> grouped = Stream.of("apple", "banana", "cherry")

.collect(Collectors.groupingBy(s -> Character.toUpperCase(s.charAt(0))));

System.out.println(grouped);

}

}

**Explanation**: The**import module** feature in JDK 23 simplifies the process of including modules in your project. In the example***, import module java.base***; allows all public top-level classes and interfaces in the java.base module to be imported, streamlining dependency management and making the **codebase** more modular and maintainable. This reduces boilerplate code and improves readability.

1. **Improved Uniformity**: By allowing primitive types in patterns, instanceof, and switch, Java becomes more consistent and expressive. It reduces the friction between working with primitive and reference types in complex code.
2. The **Graal Just-In-Time (JIT) Compiler**

**this** is an advanced feature integrated into the Oracle JDK, including the latest **Java 23** release. It's designed to improve the performance of Java applications, particularly those that are resource-intensive or involve complex calculations.

**Key Aspects of Graal JIT:**

1. **Performance Boost**: The Graal JIT compiler enhances the speed of applications by optimizing bytecode at runtime. This is particularly useful for long-running applications like microservices or backend systems, which benefit from improved performance over time.
2. **Improved Startup Times**: Java traditionally uses the HotSpot JIT compiler, but Graal offers faster startup times for applications that generate a large amount of garbage early on. This can lead to quicker execution and better efficiency when deploying Java-based cloud services, which makes it particularly relevant for cloud-native solutions.
3. **Better Code Optimization**: Graal JIT aggressively optimizes bytecode by leveraging techniques like **inlining** (embedding frequently called methods directly into the code) and **escape analysis** (identifying objects that do not escape a method or thread). These optimizations reduce memory overhead and improve throughput, making Java programs run faster, especially in high-performance environments like web services and financial applications.
4. **Multi-language Support**: Graal JIT is also part of the **GraalVM** ecosystem, which supports multiple languages (like JavaScript, Python, and Ruby) within the same runtime. This makes it appealing to developers who are working in polyglot environments.
5. **Relevance for Cloud and Containerized Environments:**

If you're working with **containerized Java applications** in platforms like **AWS Lambda**, the Graal JIT compiler is an excellent tool. It helps optimize your Java workloads in resource-constrained environments like microservices, where startup time and runtime efficiency are critical.

With the Graal JIT, you can take advantage of optimizations without the need to modify your code significantly, and your applications will automatically benefit from better memory and execution performance.

1. **Structured Concurrency**

This is a feature introduced in **Java 23** (still in **preview**) to simplify working with concurrent code, specifically focusing on making it more manageable, reliable, and easier to maintain.

### Key Concepts:

1. **Simplified Thread Management**: Structured concurrency provides an API that allows developers to organize multiple related tasks into a single **unit of work**. This eliminates the complexity and risks that often come with manually managing threads, such as potential **memory leaks**, **thread cancellations**, and **shutdown issues**.
2. **Task Grouping**: Instead of managing individual threads or ExecutorService, structured concurrency uses a **scoped** model to group tasks that are related. If one task in the group fails or needs to be canceled, the entire group can be managed together, reducing potential issues with thread management.
3. **Error Handling and Cancellation**: One of the common challenges in multi-threaded programming is handling errors when tasks are canceled or encounter an exception. Structured concurrency improves upon this by providing mechanisms to automatically handle these situations without leaving orphaned threads, ensuring more robust and predictable behavior.
4. **Improved Observability**: Structured concurrency provides better observability into running tasks, which helps with debugging and monitoring. Since all related tasks are structured under a common unit, it’s easier to keep track of task progress, completion, or failure across multiple threads.

import java.util.concurrent.\*;

import java.util.List;

public class StructuredConcurrencyExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

// Using StructuredTaskScope for structured concurrency

try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

// Start multiple tasks concurrently

Future<String> task1 = scope.fork(() -> fetchDataFromServiceA());

Future<String> task2 = scope.fork(() -> fetchDataFromServiceB());

// Wait for all tasks to complete

scope.join(); // Wait for both tasks to finish

scope.throwIfFailed(); // Propagate exceptions, if any

// Process results once all tasks have completed

String resultA = task1.resultNow();

String resultB = task2.resultNow();

System.out.println("Results: " + resultA + ", " + resultB);

}

}

Response handle() **throws** ExecutionException, InterruptedException {

**try** (**var** scope = **new** StructuredTaskScope.ShutdownOnFailure()) {

Supplier<String> user = scope.fork(() -> findUser());

Supplier<Integer> order = scope.fork(() -> fetchOrder());

scope.join() *// Join both subtasks*

.throwIfFailed(); *// Propagate errors if any*

*// Both subtasks have succeeded, compose their results*

**return** **new** Response(user.get(), order.get());

}

}

// Simulated services for data fetching

static String fetchDataFromServiceA() throws InterruptedException {

Thread.sleep(1000); // Simulate delay

return "Data from Service A";

}

static String fetchDataFromServiceB() throws InterruptedException {

Thread.sleep(1200); // Simulate delay

return "Data from Service B";

}

}

**Explanation:**

1. **StructuredTaskScope**: The StructuredTaskScope is used to manage related tasks within a scope. When the scope is closed (using try-with-resources), it ensures that all tasks are handled appropriately (either completed or failed).
2. **Forking Tasks**: We use scope.fork() to run tasks concurrently. Each task runs in a separate thread, but they are managed as part of the same group.
3. **Join and ThrowIfFailed**:
   * scope.join() waits for all tasks to complete.
   * scope.throwIfFailed() propagates any exception that occurred during task execution.
4. **Result Retrieval**: Once all tasks have completed, results are obtained via task.resultNow(), ensuring the tasks have finished executing successfully.

**Updated Class-File API**

The changes to the **Class-File API in JDK 23**provide a new API for reading and writing Java class files; this is very important in frameworks and tools needing to process a great deal of[**Java bytecode**](https://www.geeksforgeeks.org/byte-code-in-java/). This change will improve the programmability of manipulation and analysis of[**Java class files**](https://www.geeksforgeeks.org/java-class-file/), and also ease building **advanced tools** that interact with**bytecode.**

**Code Example:**

ClassFile classFile

= ClassFile.read(Paths.get("MyClass.class"));

List<Method> methods = classFile.methods();

**for** (Method method : methods) {

System.out.println("Method name: " + method.name());

}

**Explanation**: Example of reading a class file and listing its methods: The read method of ClassFile reads the class file, and the methods method retrieves the list of methods defined in the class. This JDK 23 enhancement enables programmatic manipulation and analysis of Java class files for easier development of advanced tools interacting with bytecode.

**Status:** This feature is in its second preview stage.

**6. Stream Gatherers**

**Stream gatherers** are introduced to improve the efficiency of data processing operations, particularly in applications that heavily use**Java streams**. This new mechanism allows for more efficient collection and processing of data streams, making stream-based operations faster and more resource-efficient.

Java

List<String> result = Stream.of("a", "b", "c", "d")

.gather(s -> s.toUpperCase())

.toList();

System.out.println(result);

**Explanation**: This example illustrates the use of the gather method, which allows custom intermediate operations on a stream. Th**e lambda expression s -> s.toUpperCase()**transforms each element of the stream to uppercase. The toList method collects the transformed elements into a list. This feature in**JDK 23** enhances the flexibility and efficiency of stream pipelines.

**Status:** This feature is in its second preview.

**7. Vector API Enhancements**

The[**Vector API**](https://www.geeksforgeeks.org/java-program-to-implement-vector-api/)**in JDK 23**continues its development with new enhancements, providing a mechanism to express vector computations that compile to optimal vector hardware instructions on supported**CPU architectures**. This API enables high-performance vector computations, beneficial for numerical and data-intensive applications.

**Code Example:**

Java

FloatVector.Species SPECIES = FloatVector.SPECIES\_256;

float[] array = { 1.0f, 2.0f, 3.0f, 4.0f };

FloatVector vector

= FloatVector.fromArray(SPECIES, array, 0);

FloatVector result = vector.mul(2);

result.intoArray(array, 0);

System.out.println(Arrays.toString(array));

**Status:** This feature is currently in its eighth incubator stage.

**Explanation**: This example showcases the creation of a***FloatVecto****r* using the Vector API. The species defines the size and shape of the vector, and the**fromArray**method initializes the vector with an array of floats. The ***mul***method multiplies each element by 2, and the intoArray method stores the result back into an array. These enhancements in JDK 23 enable high-performance vector computations, beneficial for numerical and data-intensive applications.

1. **Markdown support in Javadoc**

Java 23 introduces **Markdown support in Javadoc** as part of its new features, allowing developers to write more readable and expressive documentation. It enables the use of **Markdown syntax** alongside traditional HTML in Javadoc comments, making it easier to write structured and aesthetically pleasing documentation.

/\*\*

\* # Markdown in Javadoc Example

\*

\* This is an example of using \*\*Markdown\*\* in Javadoc comments.

\*

\* ## Features

\* - Use of \*\*bold\*\* text

\* - Use of \*italic\* text

\* - Code blocks:

\*

\* ```java

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

\* ```

\*

\* ## Advantages

\* 1. Easier to write

\* 2. More readable

\* 3. Seamless integration with existing HTML tags

\*

\* Here's a link to [Java documentation](https://docs.oracle.com/javase/tutorial/).

\*/

public class MarkdownJavadocExample {

/\*\*

\* Main method demonstrating Javadoc Markdown support.

\*

\* @param args Command line arguments

\*/

public static void main(String[] args) {

System.out.println("Check out the Markdown support in Javadoc!");

}

}

**Key Markdown Features Supported:**

* **Headings** (#, ##, etc.)
* **Bold** (\*\*text\*\*)
* **Italic** (\*text\*)
* **Lists** (both ordered and unordered)
* **Code blocks** (using triple backticks ```)

1. **Z Garbage Collector (ZGC)**:

Enabled by default in generational mode, it improves efficiency by reducing the resource cost associated with garbage collection.

The **Z Garbage Collector (ZGC)** is a highly scalable, low-latency garbage collector introduced in **JDK 11** and improved in later versions like **Java 23**. It is designed to handle very large heaps of memory with minimal impact on application throughput and latency.

**Key Features of ZGC:**

1. **Low Latency**: ZGC is designed to keep **pause times under 10ms**, regardless of the heap size. This makes it especially suited for applications where responsiveness and predictable latency are critical, such as real-time systems, gaming, or financial services.
2. **Scalability**: ZGC can handle heap sizes ranging from a few megabytes to **multi-terabyte** heaps. It scales well without increasing pause times, which makes it ideal for applications that require a large memory footprint.
3. **Concurrent Operations**: Most of ZGC’s operations are performed **concurrently** with the execution of the application, meaning that it does not require the application to pause for long garbage collection cycles. Key operations like marking, relocating, and compacting memory are done in the background, reducing the impact on application performance.
4. **Generational Mode**: Starting in **Java 21**, ZGC supports **generational garbage collection**, which divides the heap into **young** and **old** generations. This results in more efficient memory management, as objects that are frequently created and destroyed (young generation) are collected more frequently, while long-lived objects (old generation) are managed separately.
5. **No Full GC**: ZGC avoids traditional **"stop-the-world"** full garbage collections. It continuously manages memory in small, incremental steps, ensuring that the application runs smoothly even during garbage collection.
6. **Coloring Pointers**: ZGC uses **colored pointers** to track object locations and states without pausing the application. This technique allows ZGC to move objects around in memory without disrupting ongoing application threads.

**Example of Enabling ZGC:**

To use ZGC, you can enable it with the following JVM options:

java -XX:+UseZGC -Xms10g -Xmx10g -jar your-application.jar

Here, -XX:+UseZGC activates ZGC, and -Xms and -Xmx set the initial and maximum heap sizes to 10GB.

**Benefits of ZGC:**

1. **Improved Throughput**: ZGC's concurrent approach ensures minimal impact on application throughput, especially for applications with large heaps.
2. **Predictable Performance**: ZGC minimizes unpredictable long pauses during garbage collection, leading to more consistent application performance.
3. **Ideal for Modern Applications**: It is well-suited for modern applications running in environments like **cloud services** or **microservices**, where low-latency and scalability are critical.

**Use Cases:**

* **Large-scale web applications** that handle high traffic and require minimal response time.
* **Real-time systems** that can’t afford long pauses due to garbage collection.
* **Big data applications** or **AI systems** using large heaps of memory.

#-------------------------------------------