ItsRunTym

JAVA 8 Features

Java 8 Features

1. **Lambda Expressions**
2. **Functional Interfaces**
3. **Streams API**
4. **Default and Static Methods in Interfaces**
5. **Optional Class**
6. **Date and Time API (java.time package)**
7. **Nashorn JavaScript Engine**
8. **Method References**
9. **Collectors Class**
10. **Concurrency Enhancements**
11. **Comparable and Comparator**
12. **Annotations on Java Types**
13. **Repeating Annotations**
14. **New JavaFX Features**
15. **Enhanced Security Features**
16. **Parallel Array Sorting**
17. **Base64 Encoding and Decoding**
18. **PermGen Space Removal**
19. **Java Mission Control (JMC)**
20. **Compact Profiles**
21. **Unsigned Arithmetic Operations**

1. Lambda Expressions

- Lambda expressions enable functional programming in Java by allowing you to write inline implementation of functional interfaces, reducing boilerplate code.

- They facilitate the use of concise syntax for expressing instances of single-method interfaces.
- Lambda expressions are particularly useful in stream processing, event handling, and parallel programming.
- They promote cleaner and more readable code by focusing on the "what" (behavior) rather than the "how" (implementation details).

Java Lambda Expression Syntax

(argument-list) -> {body}

Example:

```java
java

// Before Java 8
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello from a runnable!");
    }
};

// Java 8 Lambda Expression
Runnable r2 = () -> System.out.println("Hello from a
lambda runnable!");

r1.run();
r2.run();
```

2. Functional Interfaces

- Functional interfaces have exactly one abstract method, making them suitable for use with lambda expressions.
- They can include default methods, which provide default implementations that can be overridden by implementing classes.
- Java 8 introduced static methods in interfaces to provide utility methods that are tightly related to the interface's purpose.
- Functional interfaces are foundational to Java's support for functional programming, enabling the use of lambda expressions and method references.
- The `@FunctionalInterface` annotation can be used to mark an interface as a functional interface.

Example:

```java
@FunctionalInterface
interface MyFunctionalInterface {
    void doSomething();
}

MyFunctionalInterface myFunc = () ->
System.out.println("Doing something!");
myFunc.doSomething();
```

3. Streams API

- Streams provide a declarative way to process sequences of elements.
- They support functional-style operations such as map, filter, reduce, and collect, which can be pipelined to operate on large data sets.
- Streams can be sequential or parallel, leveraging multicore architectures for improved performance.
- Introduced in Java 8, streams encourage concise and readable code for data manipulation and transformation tasks.

Example:

```java
import java.util.Arrays;
import java.util.List;

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

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

4. Default and Static Methods in Interfaces

 Default methods allow interface creators to add new methods to interfaces without breaking existing implementations.

 Static methods in interfaces provide utility methods that can be called directly on the interface itself.

 These features were introduced in Java 8 to enhance interface flexibility and backward compatibility.

 Default and static methods enable the development of more robust and reusable code in Java APIs.

Example:

```
java

interface MyInterface {
    default void defaultMethod() {
        System.out.println("This is a default
method");
    }

    static void staticMethod() {
        System.out.println("This is a static
method");
    }
}

class MyClass implements MyInterface {
    // No need to override defaultMethod
}

MyClass obj = new MyClass();
obj.defaultMethod();
MyInterface.staticMethod();
```

5. Optional Class

 The Optional class in Java is used to represent a value that may or may not be present.

 It helps prevent null pointer exceptions by encouraging explicit handling of null values.

- Optional provides methods like isPresent(), orElse(), and orElseThrow() to safely access and manipulate potentially null values.

- Introduced in Java 8, Optional promotes better coding practices by making the presence or absence of values explicit in the code.

Example:

```java
import java.util.Optional;

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

optional.ifPresent(System.out::println);  // Prints
"Hello"
System.out.println(optional.orElse("Default"));  //
Prints "Hello"

Optional<String> emptyOptional = Optional.empty();
System.out.println(emptyOptional.orElse("Default"));
// Prints "Default"
```

6. Date and Time API

- Java's java.time package introduced in Java 8 provides comprehensive date and time handling capabilities.

- It includes classes like LocalDate, LocalTime, LocalDateTime, Instant, Duration, and Period for date manipulation and formatting.

- java.time classes are immutable and thread-safe, promoting safer concurrent programming practices.

- The API addresses shortcomings of the earlier java.util.Date and java.util.Calendar classes, offering clearer and more intuitive date and time operations.

Example:

```java
import java.time.LocalDate;
import java.time.LocalDateTime;
```

```
import java.time.format.DateTimeFormatter;

LocalDate date = LocalDate.now();
LocalDateTime dateTime = LocalDateTime.now();
DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

System.out.println(date);  // Prints current date
System.out.println(dateTime.format(formatter));  //
Prints current date and time
```

7. Nashorn JavaScript Engine

- Nashorn is a lightweight JavaScript engine introduced in Java 8, enabling seamless integration of JavaScript with Java applications.

- It allows scripting capabilities within Java applications, facilitating dynamic scripting and extension of Java applications with JavaScript logic.

- Nashorn supports ECMAScript 5.1 specification and provides interoperability between Java and JavaScript code.

- The engine can be embedded in Java applications to execute JavaScript code dynamically at runtime.

Example:

```
java

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

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

try {
    engine.eval("print('Hello from JavaScript')");
} catch (ScriptException e) {
    e.printStackTrace();
}
```

8. Method References

 Method references provide a shorthand notation for lambda expressions to invoke methods.

 They can refer to static methods, instance methods, constructors, and arbitrary object methods using different syntax (::).

 Method references improve code readability by reducing verbosity, especially for lambda expressions that simply call existing methods.

 Introduced in Java 8, method references enhance the expressiveness and flexibility of functional programming in Java.

Example:

```
java

import java.util.Arrays;

String[] names = {"John", "Jane", "Jack", "Doe"};
Arrays.sort(names, String::compareToIgnoreCase);
System.out.println(Arrays.toString(names));  //
Prints sorted names
```

9. Collectors Class

 The Collectors class in Java provides utility methods for transforming elements of a stream into various data structures (like lists, sets, and maps) or performing aggregations.

 It supports common reduction operations such as grouping, partitioning, and joining elements.

 Collectors facilitate efficient and concise data processing with streams by encapsulating complex reduction operations.

 This class is essential for stream operations involving grouping, summarization, and collecting results into collections or aggregating values.

Example:

```
java

import java.util.Arrays;
```

```java
import java.util.List;
import java.util.stream.Collectors;

List<String> names = Arrays.asList("John", "Jane",
"Jack", "Doe");
List<String> filteredNames = names.stream()
    .filter(name -> name.startsWith("J"))
    .collect(Collectors.toList());

System.out.println(filteredNames);  // Prints [John,
Jane, Jack]
```

10. Concurrency Enhancements

 Java provides several enhancements for concurrent programming, including the java.util.concurrent package introduced in Java 5.

 Features like ExecutorService, ConcurrentHashMap, CountDownLatch, and Semaphore support efficient and thread-safe concurrent operations.

 Java 8 introduced CompletableFuture for asynchronous programming, offering a higher-level abstraction for handling asynchronous tasks.

 These enhancements simplify concurrent programming in Java, improving scalability and responsiveness of multi-threaded applications.

Example:

```java
java

import java.util.concurrent.CompletableFuture;

CompletableFuture.supplyAsync(() -> "Hello")
    .thenApplyAsync(result -> result + " World")
    .thenAcceptAsync(result ->
System.out.println(result));
```

11. Comparable & Comparator

Comparable

The `Comparable` interface is used to define the natural ordering of objects. It has a single method, `compareTo`, which compares the current object with another object of the same type.

Example:

```
java

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Student implements Comparable<Student> {
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(Student other) {
        return Integer.compare(this.age, other.age);  //
Compare based on age
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }

    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("John", 25));
        students.add(new Student("Jane", 22));
        students.add(new Student("Jack", 28));
        students.add(new Student("Doe", 20));

        Collections.sort(students);
        System.out.println(students);  // Output: [Doe
(20), Jane (22), John (25), Jack (28)]
    }
```

```
}
```

Comparator

The `Comparator` interface is used to define multiple ways of sorting objects. Java 8 introduced default and static methods to make it easier to chain and create comparators.

Example:

```
java

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

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

// Using Comparator with method references and chaining
names.sort(Comparator.comparing(String::length).thenCompa
ring(String::compareToIgnoreCase));

System.out.println(names);  // Output: [Doe, Jack, Jane,
John]
```

Key Points

- **Comparable**: Defines natural ordering. Implemented within the class to compare instances.
- **Comparator**: Defines custom ordering. Can be used to create multiple sorting sequences and does not modify the class.

12. Annotations on Java Types

 Annotations provide metadata about Java code, which can be used by the compiler or at runtime.

 Annotations on types (classes, interfaces, enums) are used to provide additional information or behavior to the types.

 Examples include @Override (ensures that a method overrides a superclass method), @Deprecated (marks a method or class as obsolete), and custom annotations for framework-specific or application-specific behavior.

Annotations enhance code readability, maintainability, and provide a way to convey semantics or constraints to the Java compiler or runtime environment.

Example:

java

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.TYPE_USE)
@interface MyAnnotation {}

public class Example {
    @MyAnnotation String myString;
}
```

13. Repeating Annotations

Java 8 introduced repeating annotations, allowing an annotation to be applied multiple times to the same element.

They are useful when multiple instances of the same annotation need to be associated with a program element, such as specifying multiple roles or permissions for a method or class.

Repeating annotations are declared using a container annotation that specifies @Repeatable and holds the repeated annotation type.

This feature simplifies annotation usage and improves code clarity by reducing the need for container annotations or complex metadata structures.

Example:

java

```
import java.lang.annotation.*;

@Repeatable(MyAnnotations.class)
@interface MyAnnotation {
    String value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotations {
    MyAnnotation[] value();
```

```
}

@MyAnnotation("First")
@MyAnnotation("Second")
public class Example {}

public class Main {
    public static void main(String[] args) {
        MyAnnotation[] annotations =
Example.class.getAnnotationsByType(MyAnnotation.class);
        for (MyAnnotation annotation : annotations) {
            System.out.println(annotation.value());
        }
    }
}
```

14. New JavaFX Features

 JavaFX is a platform for creating rich internet applications (RIAs) and GUI applications in Java.

 New features introduced in JavaFX include improved CSS styling capabilities, support for web components, enhanced 3D graphics rendering, and integration with modern UI controls.

 JavaFX provides a rich set of APIs for multimedia, animation, and user interface development, making it suitable for building modern desktop and mobile applications.

 JavaFX continues to evolve with updates and enhancements to support latest Java versions and improve user experience in Java-based applications.

15. Enhanced Security Features

 Java continuously enhances security features to protect applications from vulnerabilities and malicious attacks.

 Features include improved TLS support, stronger encryption algorithms, and security provider enhancements.

 Security enhancements also cover permissions and access control mechanisms for Java applications running in different environments.

 Java updates regularly address security vulnerabilities and provide guidelines for secure coding practices to safeguard applications and data.

16. Parallel Array Sorting

 Java 8 introduced parallel sorting for arrays, utilizing multiple threads for faster sorting of large arrays.

 The Arrays.parallelSort() method divides the array into smaller parts and sorts them concurrently using the Fork/Join framework.

 Parallel array sorting improves sorting performance on multi-core processors by leveraging parallelism.

 It is suitable for sorting operations where the data set is large and sorting can be done independently on different segments of the array.

Example:

```
java

import java.util.Arrays;

int[] array = {5, 3, 8, 1, 2};
Arrays.parallelSort(array);
System.out.println(Arrays.toString(array));  // Output:
[1, 2, 3, 5, 8]
```

17. Base64 Encoding and Decoding

 Java provides built-in support for Base64 encoding and decoding through classes in the java.util package.

 java.util.Base64 class provides methods to encode binary data (bytes) into a Base64 encoded string and decode Base64 encoded strings back to binary data.

 Base64 encoding is used for encoding binary data into text format, which is useful for transmitting data over text-based protocols like HTTP.

 Java's Base64 support simplifies encoding and decoding operations, ensuring interoperability and data integrity in Java applications.

Example:

```
java

import java.util.Base64;

String original = "Hello, World!";
```

```
String encoded =
Base64.getEncoder().encodeToString(original.getBytes());
String decoded = new
String(Base64.getDecoder().decode(encoded));

System.out.println(encoded);  // Output:
SGVsbG8sIFdvcmxkIQ==
System.out.println(decoded);  // Output: Hello, World!
```

18. PermGen Space Removal

- PermGen (Permanent Generation) was a part of Java's memory model used to store class metadata and interned strings.

- Starting from Java 8, PermGen space was removed and replaced with the Metaspace to improve memory management and reduce issues related to PermGen space exhaustion.

- Metaspace dynamically adjusts its size based on the application's class metadata requirements, reducing the risk of OutOfMemoryError caused by PermGen space limitations.

- This change enhances Java application performance and stability by providing more flexible and efficient memory management for class metadata.

19. Java Mission Control (JMC)

- Java Mission Control is a tool for monitoring, managing, and profiling Java applications.

- It provides real-time and historical data about JVM performance, memory usage, threads, and garbage collection.

- JMC enables developers and administrators to analyze application behavior, diagnose performance issues, and optimize JVM settings.

- It includes features like Flight Recorder for capturing detailed runtime information and Mission Control Client for visualizing and analyzing JVM data.

20. Compact Profiles

- Compact Profiles in Java provide subsets of the Java SE platform tailored for different deployment scenarios.

 They include selected APIs necessary for specific application types or devices, reducing the footprint and improving startup time of Java applications.

 Compact Profiles help optimize resource usage in embedded systems, IoT devices, and environments with limited memory and processing power.

 Java SE 8 introduced three compact profiles (Compact 1, 2, and 3) to provide flexibility in deploying Java applications across diverse platforms and devices.

21. Unsigned Arithmetic Operations

 Java traditionally does not support unsigned data types (like unsigned int) found in some other programming languages.

 Unsigned arithmetic operations can be simulated using larger signed data types (e.g., using long for unsigned int).

 Java 8 introduced methods in Integer, Long, Short, and Byte classes to support conversion and manipulation of unsigned values.

 This enables Java programmers to perform bitwise and arithmetic operations on unsigned integers with improved compatibility and performance.

Example:

```
java

int a = -1;
System.out.println(Integer.toUnsignedString(a));  //
Output: 4294967295
```