Copilot

Java 8 Features Detailed Guide

1. Lambda Expressions

Definition: Lambda expressions allow you to write concise and flexible code for instances of functional interfaces (interfaces with a single abstract method). They eliminate the need for boilerplate code associated with anonymous inner classes.

Syntax:

```
(parameters) -> expression
(parameters) -> { statements; }
```

Key Points:

- **Functional Interfaces:** A functional interface is an interface with exactly one abstract method. Java 8 provides several built-in functional interfaces such as Function, Consumer, Predicate, and Supplier.
- **Target Types:** The type of a lambda expression is inferred by the compiler based on the context, such as the target type of a variable or method parameter.

Use Case:

- **Event Handling:** Simplifies handling events by allowing you to write more readable and maintainable code.

Example:

```java
import java.util.ArrayList;
import java.util.List;

public class LambdaDeepDive {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Lambda expression to print each name
        names.forEach(name -> System.out.println(name));

        // Lambda expression to filter names starting with 'A'
        names.stream().filter(name -> name.startsWith("A")).forEach(System.out::println);
    }
}
```

Details:

- **Scopes and Capturing:** Lambdas can capture and use variables from their enclosing scope, known as closing over variables. These variables must be final or effectively final.

```java
public class LambdaScopeExample {
    public static void main(String[] args) {
        int multiplier = 2;
        List<Integer> numbers = List.of(1, 2, 3);

        // Lambda expression capturing multiplier
        numbers.stream().map(n -> n * multiplier).forEach(System.out::println);
    }
}
```

2. Functional Interfaces

Definition: A functional interface has exactly one abstract method, and it can contain other methods (default or static). They serve as the target type for lambda expressions and method references.

Built-in Functional Interfaces:

- **Function<T, R>:** Takes an argument of type T and returns a result of type R.
- **Consumer:** Takes an argument of type T and returns nothing.
- **Supplier:** Supplies a result of type T and takes no arguments.
- **Predicate:** Tests a condition on an argument of type T and returns a boolean.

Example:

```java
import java.util.function.Function;
import java.util.function.Predicate;

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Function to convert String to its length
        Function<String, Integer> lengthFunction = String::length;
        System.out.println(lengthFunction.apply("Hello")); // 5

        // Predicate to check if a string is empty
        Predicate<String> isEmptyPredicate = String::isEmpty;
        System.out.println(isEmptyPredicate.test("")); // true
    }
}
```

Details:

- **Custom Functional Interfaces:** You can create your own functional interfaces using the @FunctionalInterface annotation, though it is optional.

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

3. Streams API

Definition: The Streams API provides a way to process sequences of elements (e.g., collections) in a functional style, supporting operations like filtering, mapping, and reducing.

Key Operations:

- **Intermediate Operations:** Return a new stream and are lazy (e.g., filter, map).
- **Terminal Operations:** Produce a result or side-effect and trigger processing (e.g., collect, forEach).

Example:

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

public class StreamsDeepDive {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        // Filter, map, and collect using Streams API
        List<String> processedNames = names.stream()
            .filter(name -> name.length() > 3)
            .map(String::toUpperCase)
            .sorted()
            .collect(Collectors.toList());

        System.out.println(processedNames); // [ALICE, CHARLIE, DAVID]
    }
}
```

Details:

- **Parallel Streams:** You can parallelize operations with parallelStream(), which allows the stream to be processed in parallel.

```
names.parallelStream().forEach(name -> {
    System.out.println(name + " - " + Thread.currentThread().getName());
});
```

4. Optional Class

Definition: The Optional class is a container object that may or may not contain a non-null value. It is used to represent the possibility of a value being absent, thus avoiding null checks.

Key Methods:

- **of(T value):** Creates an Optional with a non-null value.
- **empty():** Creates an empty Optional.
- **ofNullable(T value):** Creates an Optional that may contain a value or be empty.
- **ifPresent(Consumer<? super T> action):** Executes the provided action if a value is present.
- **orElse(T other):** Returns the value if present, otherwise returns the provided default value.

Example:

```
import java.util.Optional;

public class OptionalDeepDive {
    public static void main(String[] args) {
        Optional<String> nameOptional = Optional.of("John");
        Optional<String> emptyOptional = Optional.empty();

        // Using ifPresent
        nameOptional.ifPresent(name -> System.out.println("Name: " + name));

        // Using orElse
        String name = emptyOptional.orElse("Default Name");
        System.out.println(name); // Default Name

        // Using orElseGet with Supplier
        String otherName = emptyOptional.orElseGet(() -> "Another Default Name");
        System.out.println(otherName); // Another Default Name

        // Using orElseThrow with exception
        try {
            String value = emptyOptional.orElseThrow(() -> new RuntimeException("Value is missing"));
        } catch (RuntimeException e) {
            System.out.println(e.getMessage()); // Value is missing
        }
    }
}
```

Details:

- **Chaining Methods:** You can chain methods to perform operations based on the presence or absence of a value.

```
String result = Optional.of("hello")
                        .filter(s -> s.length() > 3)
                        .map(String::toUpperCase)
                        .orElse("DEFAULT");
System.out.println(result); // HELLO
```

5. Default and Static Methods in Interfaces

Definition: Java 8 allows interfaces to have default methods with an implementation and static methods. Default methods enable you to add new methods to interfaces without breaking existing implementations.

Default Methods:

- **Syntax:** `default returnType methodName() { // implementation }`

Static Methods:

- **Syntax:** `static returnType methodName() { // implementation }`

Example:

```
interface MyInterface {
    default void defaultMethod() {
        System.out.println("Default method");
    }

    static void staticMethod() {
        System.out.println("Static method");
    }
}

public class DefaultStaticMethodExample implements MyInterface {
    public static void main(String[] args) {
        MyInterface obj = new DefaultStaticMethodExample();
        obj.defaultMethod(); // Output: Default method

        MyInterface.staticMethod(); // Output: Static method
    }
}
```

Details:

- **Conflict Resolution:** If a class implements multiple interfaces with default methods having the same signature, it must override the method to resolve the conflict.

```
interface InterfaceA {
    default void show() {
        System.out.println("InterfaceA");
    }
}

interface InterfaceB {
    default void show() {
        System.out.println("InterfaceB");
    }
}

public class ConflictResolution implements InterfaceA, InterfaceB {
    @Override
    public void show() {
        System.out.println("ConflictResolution");
    }

    public static void main(String[] args) {
        new ConflictResolution().show(); // Output: ConflictResolution
    }
}
```

6. Method References

Definition: Method references provide a shorthand syntax for calling methods directly. They are used to pass methods as arguments to functional interfaces.

Types of Method References:

- **Static Method Reference:** `ClassName::staticMethodName`
- **Instance Method Reference on a Particular Object:** `instance::instanceMethodName`
- **Instance Method Reference on an Arbitrary Object:** `ClassName::instanceMethodName`
- **Constructor Reference:** `ClassName::new`

Example:

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

```java
public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Static method reference
        names.forEach(MethodReferenceExample::print);

        // Instance method reference on a particular object
        MethodReferenceExample instance = new MethodReferenceExample();
        names.forEach(instance::printUpperCase);

        // Constructor reference
        List<String> filteredNames = names.stream()
                                          .filter(name -> name.length() > 3)
                                          .map(String::new)
                                          .collect(Collectors.toList());
    }

    public static void print(String name) {
        System.out.println(name);
    }

    public void printUpperCase(String name) {
        System.out.println(name.toUpperCase());
    }
}
```

Details:

- **Advantages:** Method references can make code more readable and concise compared to lambda expressions, especially when passing existing methods