Tech with Jatin

10 Must Know Features Of Java

Tech with Jatin

λ- > {}

Lambda Expressions

Description:

Lambda expressions allow writing concise and expressive code by representing functions as objects.

Where to Use It:

For SDETs:

Great for writing clean and concise test logic, especially when dealing with collections in test data setup or validation.

For Developers:

Ideal for replacing anonymous inner classes, simplifying event handling, and working with APIs.

Code Snippet:

```
List<String> names = Arrays.asList("Katrina", "Alia", "Kiara");
names.forEach(name -> System.out.println(name));
```

Tech with Jatin

@functionalinterfaces

Functional Interfaces

Description:

An interface with a single abstract method, like Predicate, Function, and Consumer. Helps enable lambda expressions.

Where to Use It:

For SDETs:

Use Predicate for filtering data in automated tests or to define reusable conditions in validation logic.

For Developers:

Functional interfaces can simplify callback mechanisms and are commonly used with streams and lambda expressions.

Code Snippet:

```
@FunctionalInterface
interface Greeting {
    void greet(String name);
}
Greeting greeting = (name) -> System.out.println("Hello, " + name);
greeting.greet("Katrina");
```

Tech with Jatin

Stream API

Description:

A new abstraction for processing collections in a functional style, enabling efficient operations like filtering, mapping, and reducing.

Where to Use It:

For SDETs:

Perfect for data filtering, transformation, and validation in large datasets, such as processing test results or logs.

For Developers:

Efficiently handle collections, arrays, or lists in complex business logic, especially for data processing tasks.

Code Snippet:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squaredEvens = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());
System.out.println(squaredEvens); // Output: [4, 16]
```

Tech with Jatin

Default Methods

Description:

Interfaces can now have method implementations, allowing flexibility when adding new functionality without breaking existing code.

Where to Use It:

For SDETs:

Useful when building custom frameworks or libraries, allowing you to extend interfaces without breaking backward compatibility.

For Developers:

Helps in API evolution and framework design, enabling easier maintenance and feature addition without refactoring.

Code Snippet:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squaredEvens = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());
System.out.println(squaredEvens); // Output: [4, 16]
```

Optional Class

Description:

Avoid NullPointerException by using Optional to represent the presence or absence of a value.

Where to Use It:

For SDETs:

Great for handling optional test data, assertions, or configuration values, reducing the chance of null-related test failures.

For Developers:

Ideal for API design, where you want to explicitly indicate the possibility of a missing or null value without using exceptions.

Code Snippet:

```
Optional<String> optional = Optional.of("Hello");
optional.ifPresent(System.out::println); // Output: Hello

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

Tech with Jatin

New Date and Time API

Description:

Provides a comprehensive and immutable API for handling date and time, replacing Date and Calendar.

Where to Use It:

For SDETs:

Use it to work with time-based test data, scheduling test runs, or validating timestamps in reports.

For Developers:

A more reliable and thread-safe way to handle date and time, especially in enterprise applications with complex date calculations.

Code Snippet:

```
LocalDateTime now = LocalDateTime.now();
LocalDate date = LocalDate.of(2024, 9, 15);
LocalDate nextWeek = date.plusWeeks(1);
System.out.println(nextWeek); // Output: 2024-09-22
```

Tech with Jatin

Method References

Description:

Simplifies lambda expressions even further by referencing existing methods directly.

Where to Use It:

For SDETs:

Useful for clean test code when passing behavior to functions, particularly in data-driven testing.

For Developers:

Method references can reduce boilerplate code when using streams, listeners, or event handling.

Code Snippet:

```
List<String> names = Arrays.asList("Katrina", "Alia", "Kiara");
names.forEach(System.out::println);
```

Nashorn JavaScript Engine

Description:

Integrates JavaScript execution with Java using Nashorn, allowing JavaScript code to run within Java applications.

Where to Use It:

For SDETs:

Use Nashorn for executing JavaScript-based automation scripts within Java frameworks like Selenium.

For Developers:

Run JavaScript logic in hybrid applications, or work with JavaScript libraries directly in Java codebases.

Code Snippet:

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");
String script = "var welcome = 'Hello, Nashorn'; welcome;";
String result = (String) engine.eval(script);
System.out.println(result); // Output: Hello, Nashorn
```

Tech with Jatin

Parallel Streams

Description:

Parallel processing for faster execution of stream operations by utilizing multi-core processors.

Where to Use It:

For SDETs:

Use parallel streams to speed up test data processing or log analysis, especially when handling large datasets.

For Developers:

Improve performance in data-heavy applications, making use of multi-core processors for parallel execution..

Code Snippet:

```
List<Integer> parallelSquaredNumbers = numbers.parallelStream()
    .map(n -> n * n)
    .collect(Collectors.toList());
System.out.println(parallelSquaredNumbers);
```

Tech with Jatin

Type Annotations

Description:

Annotations can now be used to mark types, enhancing type-checking and reducing errors.

Where to Use It:

For SDETs:
Use type annotations in test frameworks to improve type safety, making automated tests more reliable.

For Developers:
Helps ensure strict type-checking and improves the use of generics, particularly in APIs and libraries.

Code Snippet:

```
@Target(ElementType.TYPE_USE)
@interface NonNull {}
@NonNull String myString = "Hello, Type Annotations!";
System.out.println(myString);
```