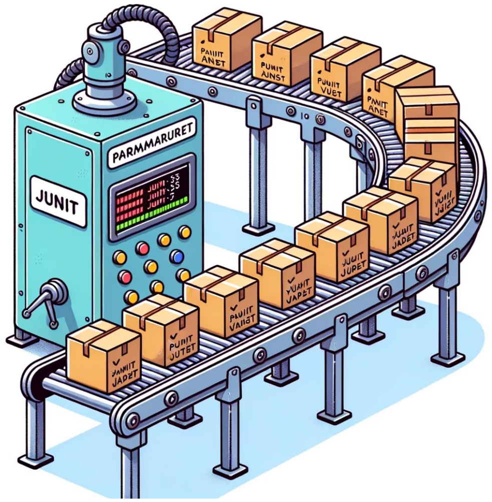
Parameterized Testing in JUnit 5



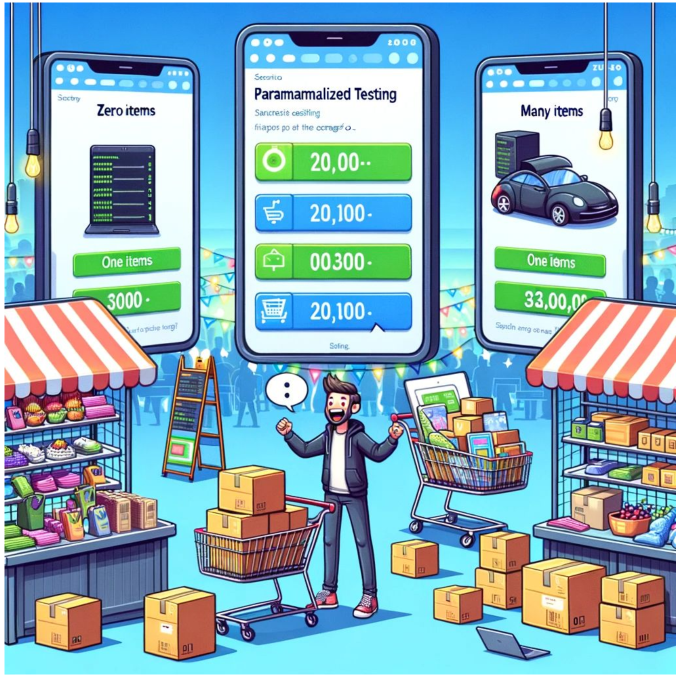
Understanding Parameterized Testing

Lesson Objectives

- Unveil the concept of Parameterized Testing,
- Learn how to create Parameterized Tests using JUnit 5,
- And explore the benefits and scenarios where Parameterized Testing shines.

Advantages of Parameterized Testing

- Code Reusability
- Ease of Maintenance
- Increased Coverage
- Efficient Identification of Edge Cases
- Readability and Clarity
- Reduced Code Duplication
- Time Efficiency
- Easier Debugging
- Flexible and Scalable
- Enhanced Reporting



Creating Parameterized Tests in JUnit 5

Dependencies

- JUnit Jupiter API
- JUnit Jupiter Params

Basic Parameterized Test using @ValueSource

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3, 4, 5})
void testIsEven(int number) {
    assertEquals(0, number % 2);
}
```

Parameterized Test using @EnumSource

```
@ParameterizedTest
@EnumSource(DayOfWeek.class)
void testIsWeekend(DayOfWeek day) {
    assertTrue(day == DayOfWeek.SATURDAY || day == DayOfWeek.SUNDAY);
}
```

Parameterized Test using @ArgumentsSource

```
@ParameterizedTest
@ArgumentsSource(MyArgumentsProvider.class)
void testWithArgumentsSource(int argument) {
    assertTrue(argument > 0 && argument < 10);
}

static class MyArgumentsProvider implements ArgumentsProvider {
    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext
context) {
        return Stream.of(1, 2, 3, 4, 5).map(Arguments::of);
    }
}
```

Parameterized Test using @MethodSource

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
```

Parameterized Test using @CsvSource

```java
@ParameterizedTest
@CsvSource({
    "apple, 1",
    "banana, 2"
})
void testWithCsvSource(String fruit, int rank) {
    assertNotNull(fruit);
    assertNotEquals(0, rank);
}
```

Exploring Advanced Features

Argument Conversion

```
@ParameterizedTest
@ValueSource(strings = {"2020-01-01", "2024-01-01"})
void isLeapYear(Date date) {
    assertTrue(date.isLeapYear());
}
```

Argument Aggregation

```
@ParameterizedTest
@CsvSource({"apple, 1", "banana, 2"})
void testFruits(Fruit fruit) {
    assertNotNull(fruit.getName());
    assertTrue(fruit.getQuantity() > 0);
}

static class Fruit {
    private String name;
    private int quantity;

    Fruit(String name, int quantity) {
        this.name = name;
        this.quantity = quantity;
    }
    // getters
}
```

Customizing Display Names

```java
@ParameterizedTest(name = "{index} => fruit={0}, rank={1}")
@CsvSource({"apple, 1", "banana, 2"})
void testFruits(String fruit, int rank) {
    // test logic here
}
```

Best Practices in Parameterized Testing

Organizing Code for Parameterized Tests

- A well-organized test is a well-communicated test. Ensure that your tests are grouped logically and are easy to find.
- It’s advisable to keep your Parameterized Tests in a separate class or grouped together within a test class.
- Also, ensure that the data sources for your Parameterized Tests are close to the tests themselves or easy to locate.

Naming Conventions and Commenting

- Meaningful names are your best friends! Name your tests in a way that describes what the test is doing.
- JUnit 5 provides a feature to customize the display name of your Parameterized Tests using @DisplayName and @DisplayNameGeneration. Take advantage of this to make your tests self-explanatory.
- Comments are crucial, especially when the logic of the data providers or the tests is complex.

Widely Used Naming Conventions for Parameterized Tests

- **Descriptive Method Names with Inputs:**
    - Format: [methodName]_With[InputType1]And[InputType2]_Should[ExpectedResult]
    - Example: calculateInterest_WithPrincipalAndRate_ShouldReturnCorrectInterest
- **Incorporating Parameter Values in Test Names:**
    - Format: [methodName]_When[ParameterCondition]_Then[ExpectedResult]
    - Example: addNumbers_WhenNegativeInput_ThenThrowException
- **Naming Based on Test Scenarios or Use Cases:**
    - Format: Should_[ExpectedResult]_When_[Scenario]
    - Example: Should_ThrowException_When_NegativeInputs
- **Including Expected Outcome in the Name:**
    - Format: [methodName]_[ExpectedOutcome]_Given[Condition]
    - Example: divide_NonZeroResult_GivenNonZeroDivisor
- **Using Data Type or Nature of Test Data in the Name:**
    - Format: [methodName]_With[DataTypeOrNature]
    - Example: processPayment_WithInvalidCardDetails, processPayment_WithExpiredCard
- **Describing the Test Purpose Clearly:**
    - Format: test[MethodName]_[Condition]_Expect[Outcome]
    - Example: testCalculateAge_InvalidBirthdate_ExpectException
- **Custom Annotations or Naming Extensions:**
    - Some frameworks provide ways to customize the display name of parameterized tests dynamically based on the input parameters.
    - Example: @DisplayName("Test add method with {0} and {1}").

Avoiding Common Pitfalls

- Avoid a large number of parameters in your tests. It can make your tests hard to read and maintain.
- Ensure your data sets are comprehensive but not excessive. Covering edge cases is good, overdoing it is not.
- Lastly, ensure that failing tests are easy to diagnose. A failing Parameterized Test should clearly indicate what input caused the failure.