JUnit

Understanding JUnit

Learning Objectives

By the end of this lesson, you will be able to:

- Learn about the basics of JUnit
- Explain the advantages of JUnit
- Learn how to set up JUnit
- Explain the JUnit classes with examples
- Discuss the refactoring process the JUnit test

Learning Objectives

By the end of this lesson, you will be able to:

- List and explain JUnit methods
- List and explain JUnit annotations
- Learn how to compare arrays in JUnit tests
- Execute JUnit test cases

A Day in the Life of a Full Stack Developer

You are working in an organization and have been assigned a project. Your role is to perform testing on the application and write test cases for the same. The idea is to test each module that gets developed.

Since the application is in Maven, you decide to use JUnit, as it can be a part of the unit testing framework and is crucial for test-driven development.

To do so, you will explore the basics of JUnit, set it up, understand the annotations, and execute its test cases.

JUnit: Overview

Discussion

Preventing Errors

Imagine a customer ordering a birthday cake for a friend using a food delivery app. He adds the cake to his cart and gets to the payment page to complete the order. However, he is unable to make the payment and the transaction fails.

- What will you do to make sure all transactions are flawless?
- What could be done to prevent this from happening again?

JUnit

Unit testing is a software testing methodology where individual units or components of a software system are tested independently to ensure that each unit works as expected.

JUnit plays an important role in Unit Testing as it provides a framework for writing and executing automated tests, ensuring that code changes do not break the existing functionality.

JUnit

JUnit is a unit testing framework that is:

- Crucial for test-driven development
- A part of unit testing frameworks

JUnit

It works on the principle of:

"First testing, then coding"

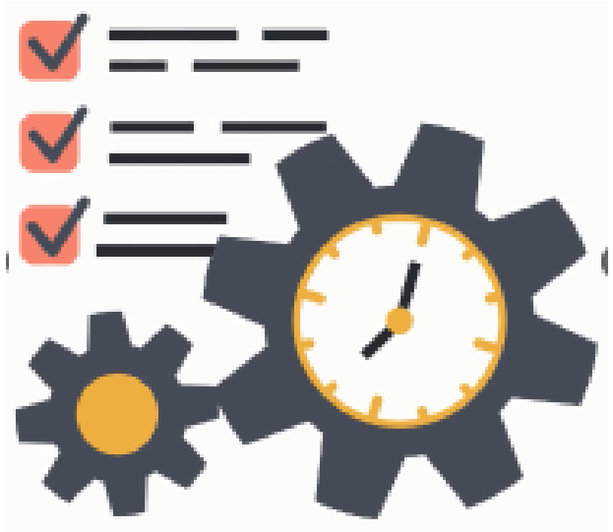
Testing

Coding

"Test a bit, code a bit, test a bit, code a bit."

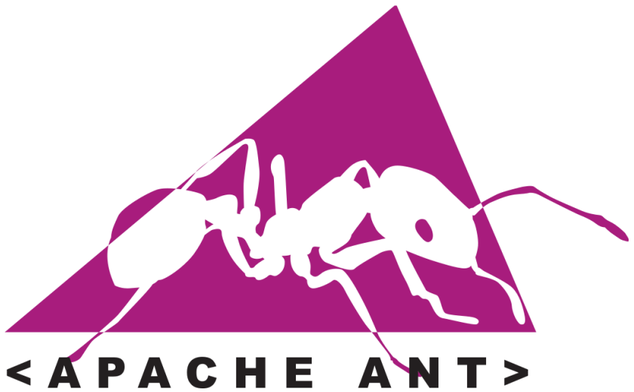
JUnit

It gives power to a developer and strength to program code.

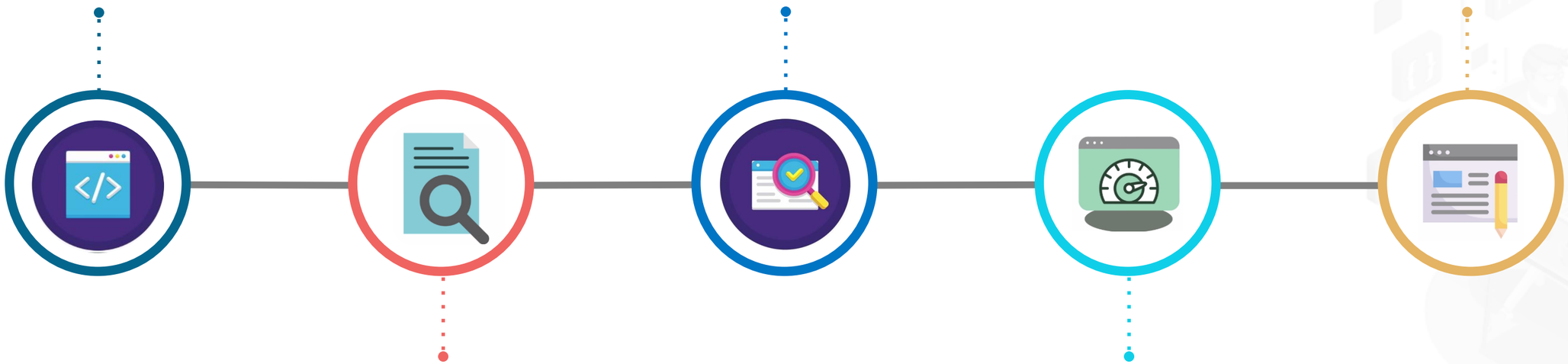


JUnit

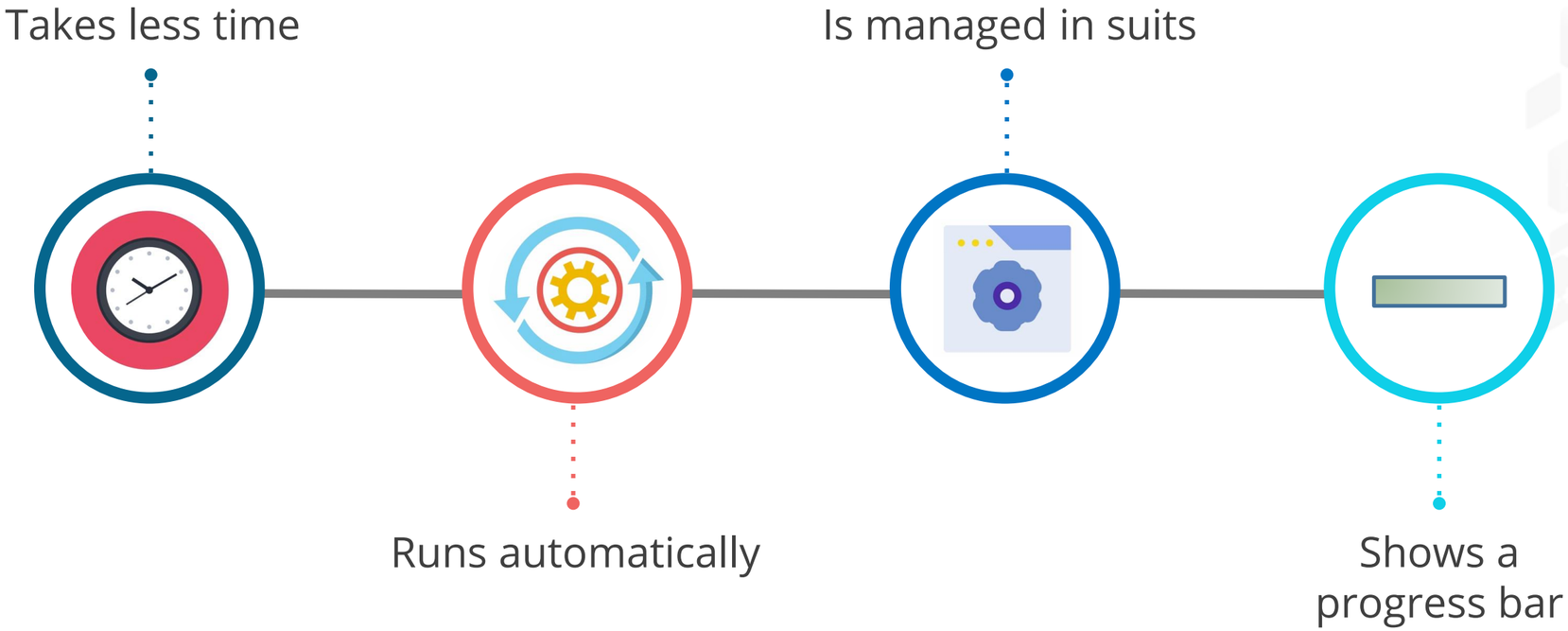
JUnit framework can easily be used with:



Features of JUnit



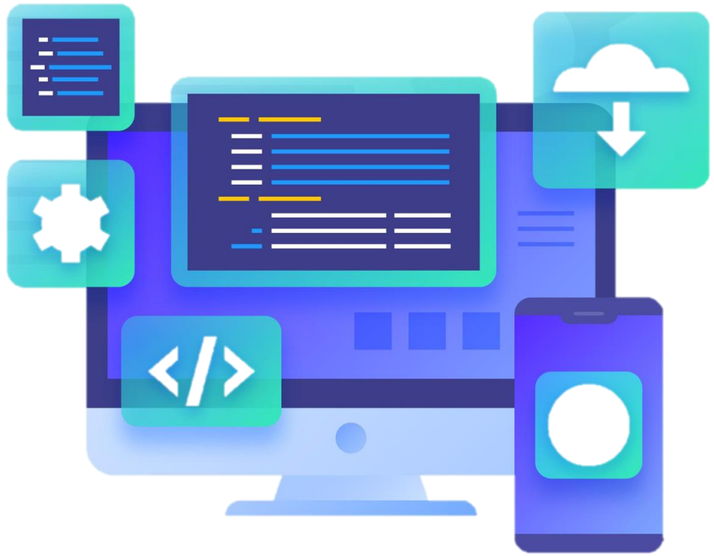
Features of JUnit



Why Is JUnit Widely Used?

Open-Source

It is an open-source framework.



Why Is JUnit Widely Used?

Early Bug-Finder

It finds out the bug quickly in the code.

Why Is JUnit Widely Used?

Best for TDD Environment

Developers first perform the tests and then resolve the issues.

JUnit uses assertions in the test and is the most efficient when they fail.

Using JUnit in Maven Project

Problem Statement:

You have been asked to use Junit in a Maven project and test the project by configuring it with the new compiler plugin.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Creating a new Maven project
2. Configuring the project with a new compiler plugin
3. Creating a test method

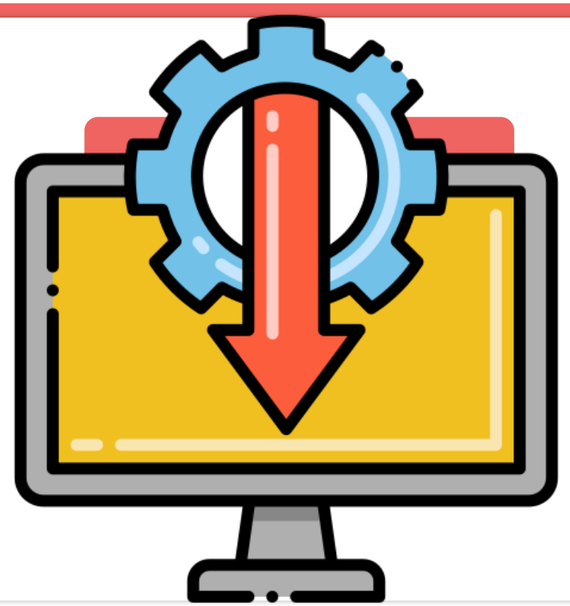
Setting Up JUnit

Setting Up JUnit

1. Install JDK

For Windows

Open a command console and run "java -version"



For Mac

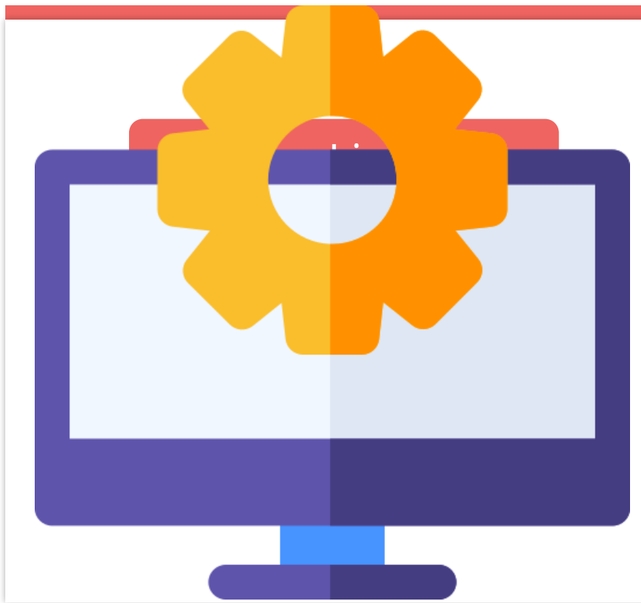
Open the terminal and run "java -version"

Setting Up JUnit

2. Set up Java environment

For Windows

Set the environment variable JAVA_HOME to **C:\Program Files\Java\jdkx.x.x_xx**



For Mac

Export **JAVA_HOME=/Library/JAVA/HOME**

Setting Up JUnit

3. Download the Unit archive: To download the latest jar file of JUnit from http://www.junit.org

Setting Up JUnit

4. Set the JUNIT_HOME environment variable

For Windows

Set the environment variable **JUNIT_HOME** to **C:\JUNIT**

For Linux

Export **JUNIT_HOME = /user/local/JUNIT**

For Mac

Export **JUNIT_HOME = /Library/JUNIT**

Setting Up JUnit

5. Set the CLASSPATH variable

For Windows

Set the environment variable CLASSPATH to **%CLASSPATH%;%JUNIT_HOME%\junit.x.xx.jar;**

For Linux

Export **CLASSPATH=CLASSPATH:JUNIT_HOME/junitx.xx.jar:.**

For Mac

Export **CLASSPATH=CLASSPATH:JUNIT_HOME/junitx.xx.jar.**

Setting Up JUnit

6. To test JUnit Setup: Create a Java class with the name ExampleJunit

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class ExampleJunit {
 @Test
 public void addTest() {
  String response = “Example Saved”;
  assertEquals(“Example Saved”, str);
 }
}
```

Setting Up JUnit

6. Test JUnit Setup

Create a class with the name TestRun to run the test cases.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TestRun {
 public static void main(String[] args) {
  Result result =
JUnitCore.runClasses(TestJunit.class);
  for (Failure failure: result.getFailures()) {
   System.out.println(failure.toString());
  }
  System.out.println(result.wasSuccessful());
 }
}
```

Setting Up JUnit

7. Compile the classes

```
C:\YourTestDirectory>javac ExampleJunit.java TestRun.java
```

Finally, execute the Test Run to see the result.

```
C:\YourTestDirectory>java TestRun

Output: true
```

Features of JUnit

Discussion

Checkpoints

Imagine you are a software tester working on a banking application. You have to ensure the application is accurate and reliable by creating checkpoints for transferring and depositing money. In addition, the application needs to be secure and trustworthy for customers to update the account balance correctly after each transaction.

- Is it possible to track whether these operations are carried out correctly?
- What steps will you take to ensure perfect execution of every transaction?

Checkpoints

The setup method is executed before each test case is run and is used to initialize any objects or resources required by the test. This can include creating test data, initializing variables, or connecting to a database.

The teardown method is executed after each test case is run and is used to clean up any resources used by the test. This can include closing connections, deleting test data, or releasing resources back to the system.

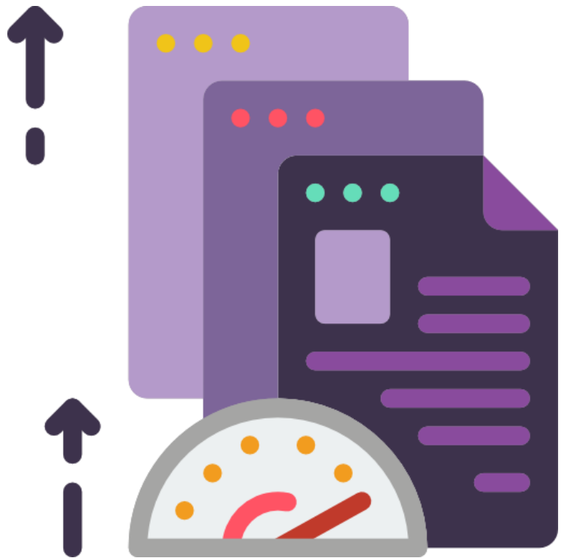
Features of JUnit

JUnit provides some important features that include:

- Fixtures
- Test Suites
- Test Runners

Fixtures

They are fixed states of objects employed as a baseline for running tests.



- Ensure a well-known and fixed environment for test execution
- Ensure the results are repeatable

Fixtures

They include:

setUp()

Executes before each test invocation

tearDown()

Executes after each test method

Fixtures

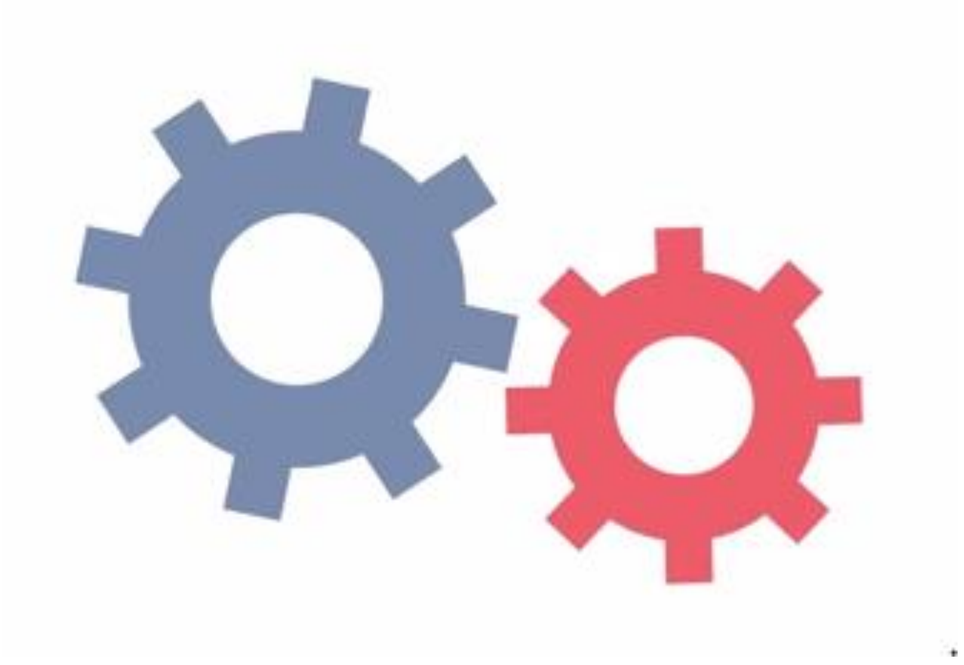
Example:

```
import junit.framework.*;
public class MyTestCase extends TestCase {
  protected int walletBalance, rewardPoints;

  // assigning the values
  protected void setUp(){
   walletBalance = 500 ;
   rewardPoints = 200;
  }
// test method for adding two values
  public void addTest(){
   double walletBalanceAfterRedeem = walletBalance +
rewardPoints/10;
   assertTrue(walletBalanceAfterRedeem == 520);
  }
}
```

Test Suites

It combines a few unit test cases and executes them together.

To run a suite, both @RunWith and @Suite annotations are used.



Test Suites

An example of a test suite that uses EmailTest1 and EmailTest2 classes:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

//JUnit Suite Test
@RunWith(Suite. class)
@Suite.SuiteClasses({
  EmailTest1.class ,EmailTest2.class
})
public class MyTestSuite {

}
```

Test Suites

EmailTest1 class:

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class EmailTest1 {

  String email = "john@example.com";
  Email emailUtil = new Email(email);

  @Test
  public void testPrintEmail() {
   System.out.println("Inside testPrintMessage()");
   assertEquals(email, emailUtil.printUserEmail());
  }
}
```

Test Suites

EmailTest2 class:

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class EmailTest2{

  String email = "john@example.com";
  Email emailUtil = new Email(email);

  @Test
  public void test() {
   System.out.println("Inside testSalutation()");
   String message = "Email Sent!" + email;
   assertEquals(message,emailUtil.sendEmail());
  }
}
```

Test Suites

Email.java Class manages the Email object using EmailTest1 and EmailTest2 classes.

```
public class Email {
  private String userEmail;

   // Constructor to create Email object
  public Email(String userEmail){
      this.userEmail = userEmail;
   }

   // prints the Email
   public String printUserEmail(){
      System.out.println(userEmail);
      return userEmail;
   }
```

Test Suites

Email.java Class manages the Email object using EmailTest1 and EmailTest2 classes.

```
// add "Email Sent!" to the userEmail
   public String sendEmail(){
      String message = "Email Sent! " + userEmail;
      System.out.println(message);
      return message;
   }
}
```

Test Runners

It is employed to run the test cases.

Example:

```
// Run EmailTest1
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class EmailTestRunner {
 public static void main(String[] args) {
  Result result = JUnitCore.runClasses(EmailTest1.class);
  for (Failure failure: result.getFailures()) {
   System.err.println("Test Case Failed:
"+failure.toString());
  }
  System.out.println("Test Case Success:
"+result.wasSuccessful());
 }
}
```

Test Runners

```
// Run EmailTest2
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class EmailTestRunner {
 public static void main(String[] args) {
  Result result = JUnitCore.runClasses(EmailTest2.class);
  for (Failure failure: result.getFailures()) {
   System.err.println("Test Case Failed:
"+failure.toString());
  }
  System.out.println("Test Case Success:
"+result.wasSuccessful());
 }
}
```

JUnit Classes

JUnit Classes

In JUnit, three important classes are used for writing and testing units:

Assert

It contains a set of assert methods.

TestCase

It contains a test case that indicates that the fixture should execute multiple tests.

TestResult

It contains methods to compile the results after a test case is run.

Assert Class

The Assert class is used to validate the steps during execution.

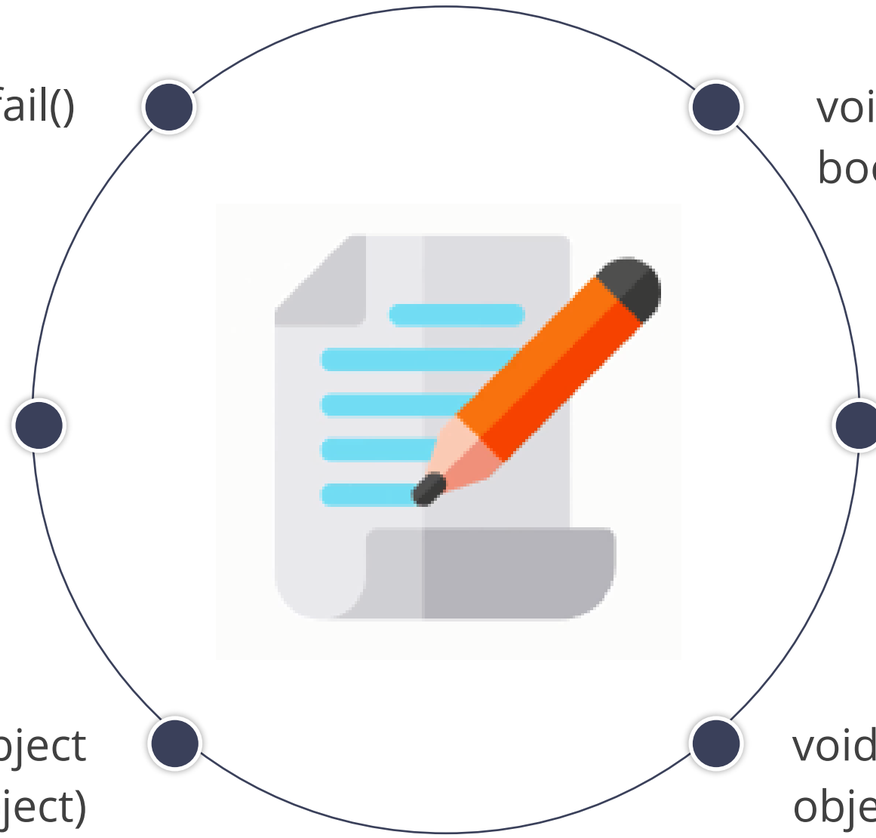
It provides several assertion methods that are useful for writing tests.

Syntax:

```
public class Assert extends java.lang.Object
```

Assert Class

The important methods available in the Assert class, which includes:



- void fail()
- void assertEquals(boolean expected, boolean actual)
- void assertTrue(boolean condition)
- void assertFalse(boolean condition)
- void assertNull(Object object)
- void assertNotNull(Object object)

TestCase Class

It defines the capability to run multiple tests.

Syntax:

```
public abstract class TestCase extends Assert implements Test
```

TestCase Class

The important methods available in TestCase class are:

- void tearDown()
- TestResult run()
- void setUp()
- void run(TestResult result)

TestResult Class

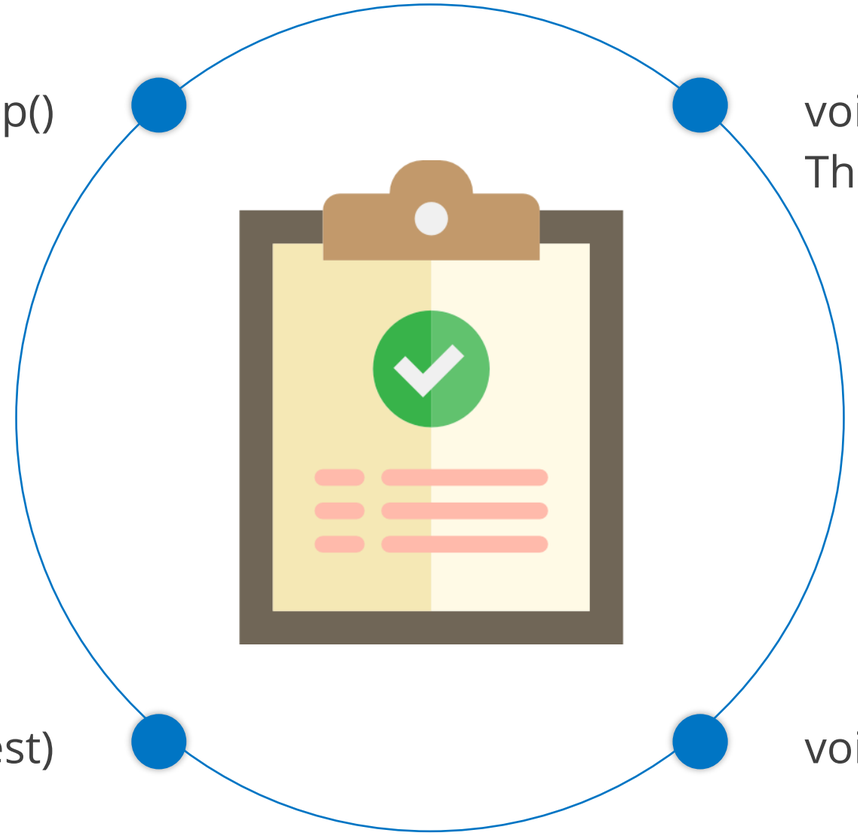
It collects the results of executing a test case.

Syntax:

```
public class TestResult extends Object
```

TestResult Class

The important methods of TestResult class include:



void stop()

void addError(Test test, Throwable t)

void startTest(Test test)

void run(TestCase test)

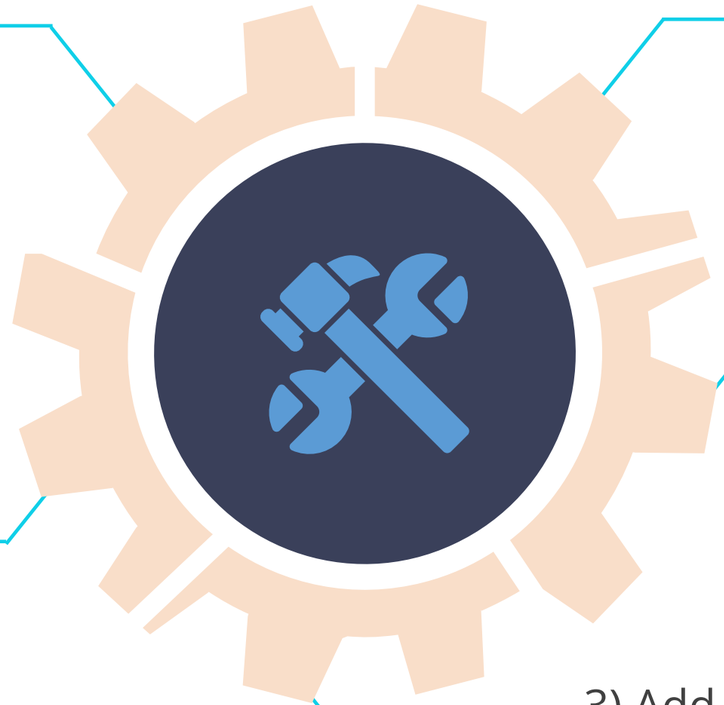
First Successful JUnit

First Successful JUnit

Create the Java class that needs to be tested, like Email.java, in C:\>YourTestDirectory

```
public class Email {
  private String userEmail;
   // Constructor to create Email object
  public Email(String userEmail){
      this.userEmail = userEmail;
   }
   // prints the Email
   public String printUserEmail(){
      System.out.println(userEmail);
      return userEmail;
   }
   // add "Email Sent!" to the userEmail
   public String sendEmail(){
      String message = "Email Sent! " + userEmail;
      System.out.println(message);
      return message;
   }
}
```

First Successful JUnit



1) Create a test class, for instance, Test.java in C:\>YourTestDirectory

2) Add a test method testPrint()

3) Add an Annotation @Test to method testPrint()

4) Implement the test condition

5) Check the condition using assertEquals API

First Successful JUnit

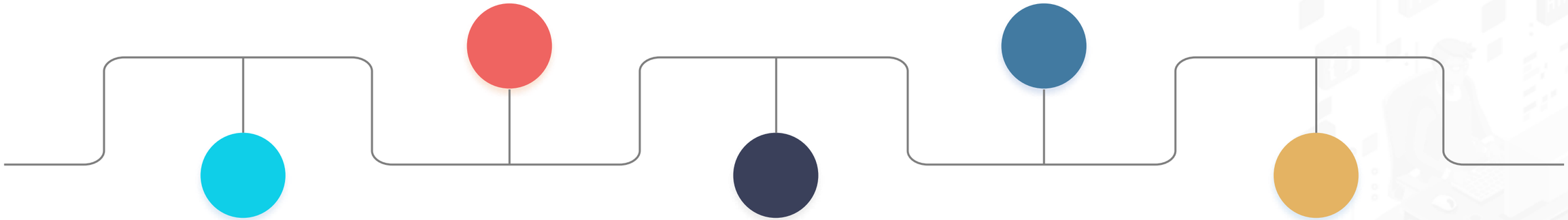
Example:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class Test1 {
  String email = “fionna@example.com";
  Email emailUtil = new Email(message);

  @Test
  public void testPrint() {
   assertEquals(email,emailUtil.printUserEmail());
  }
}
```

First Successful JUnit

Use the runClasses method to execute the test case

Use the getFailure method to obtain the failure(s)



Focus on creating a TestRun class of Java

Get the result of test cases executed in the Result object

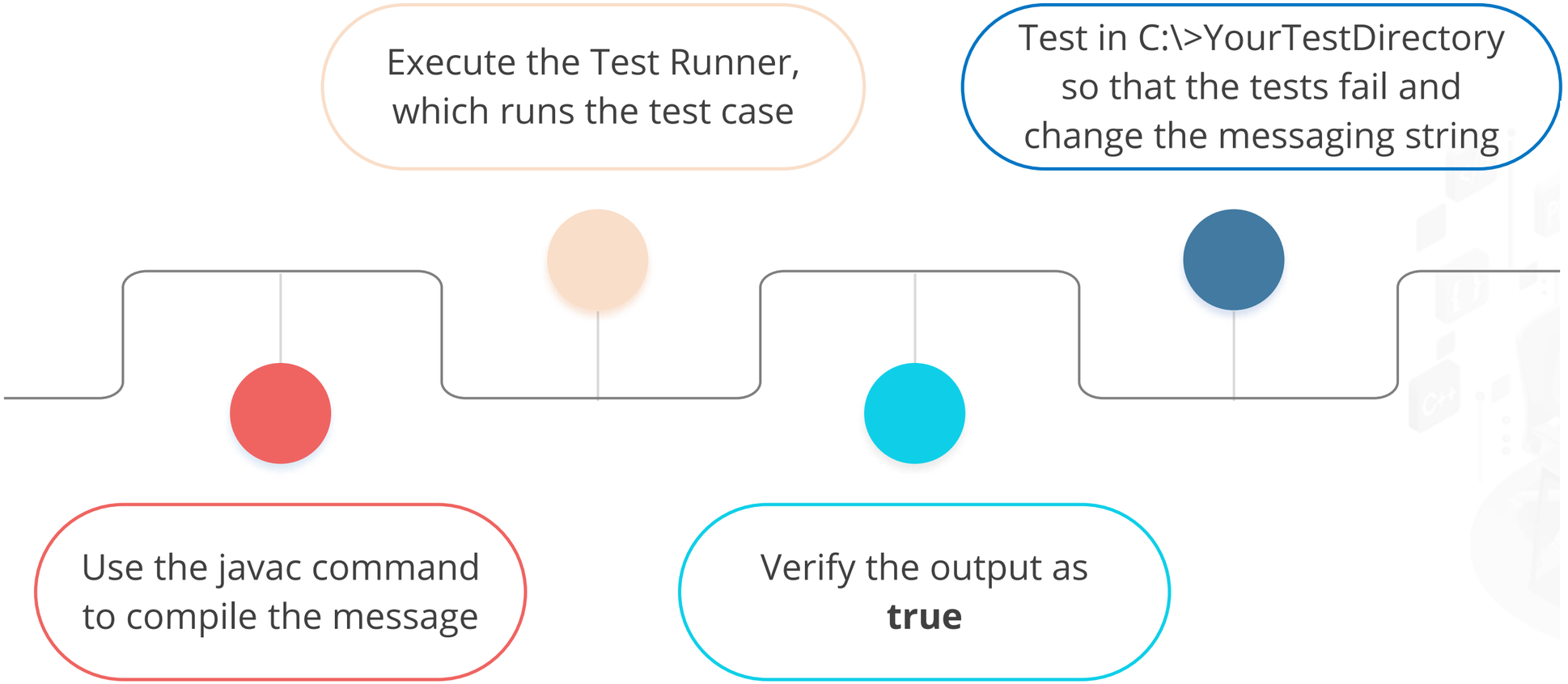
Get the success result with the help of the wasSuccessful() method

First Successful JUnit

Example:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRun1 {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(Test.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

First Successful JUnit



First Successful JUnit

Example:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class Test {
   String email = "george@example.com";
   Email emailUtil = new Email(email);

   @Test
   public void testPrintMessage() {
      email = "kia@example.com";
      assertEquals(email,emailUtil.printUserEmail());
   }
}
```

First Successful JUnit

Execute the same Test Runner, keeping the rest of the classes the same:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {

   public static void main(String[] args) {
      Result result =
JUnitCore.runClasses(TestJunit.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

First Successful JUnit

Execute the Test Runner class and check the output:

result.wasSuccessful() = false

Refactoring JUnit Test

Refactoring JUnit Test

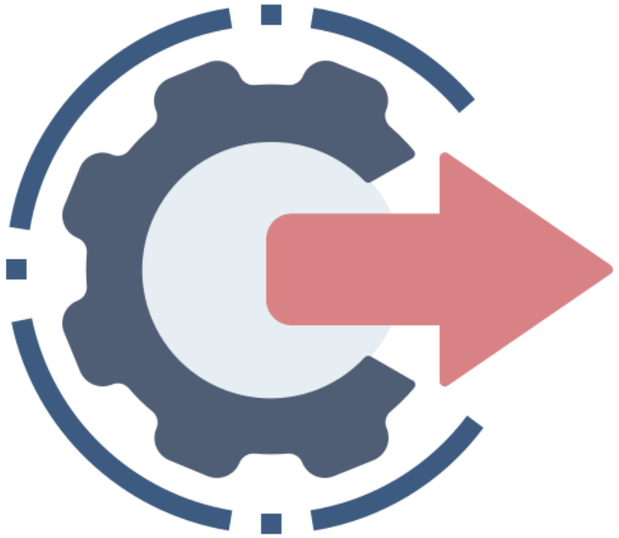
An assertion is used to check if a particular condition or logic returns true or false.

Note

If it is false, then an AssertionError is thrown.

Refactoring JUnit Test

An assertion helps to validate the expected output with the actual.



It is the best way of refactoring the JUnit test.

Refactoring JUnit Test

Advantages of using an assertion:

- It is the fastest, best, and most effective way to detect and fix bugs.
- It helps the user to gain confidence that assumptions about a particular condition's output are correct.
- It is a precondition for a method call or can be post-conditioned too.

assertTrue and assertFalse

assertTrue and assertFalse

Create a Java class named MyAssertionTest.java in C:\>YourTestDirectory:

```
import org.junit.Test;
import static org.junit.Assert.*;
public class MyAssertionTest {
   @Test
   public void testCase() {
     //test data
      String expectedEmail = new String ("john@example.com");
      String userEmail = new String ("john@example.com");
      String name1 = null;
      String name2 = "Fionna";
      String name3 = "Fionna";
      int walletBalance = 200;
      int cabFare = 500;
      String[] expectedSkills = {"html", "css", "javascript"};
      String[] knownSkills =  {"html", "css", "javascript"};
//Check that two objects are equal
```

assertTrue and assertFalse

Create a Java class named MyAssertionTest.java in C:\>YourTestDirectory:

```
assertEquals(expectedEmail, userEmail);
      //Check that a condition is true
      assertTrue (walletBalance < cabFare);
      //Check that a condition is false
      assertFalse(walletBalance > cabFare);
      //Check that an object isn't null
      assertNotNull(name2);
      //Check that an object is null
      assertNull(name1);
      //Check if two object references point to the same object
      assertSame(name2,name3);
      //Check if two object references not point to the same
object
      assertNotSame(name1,name2);
      //Check whether two arrays are equal to each other.
      assertArrayEquals(expectedSkills, knownSkills);
   }}
```

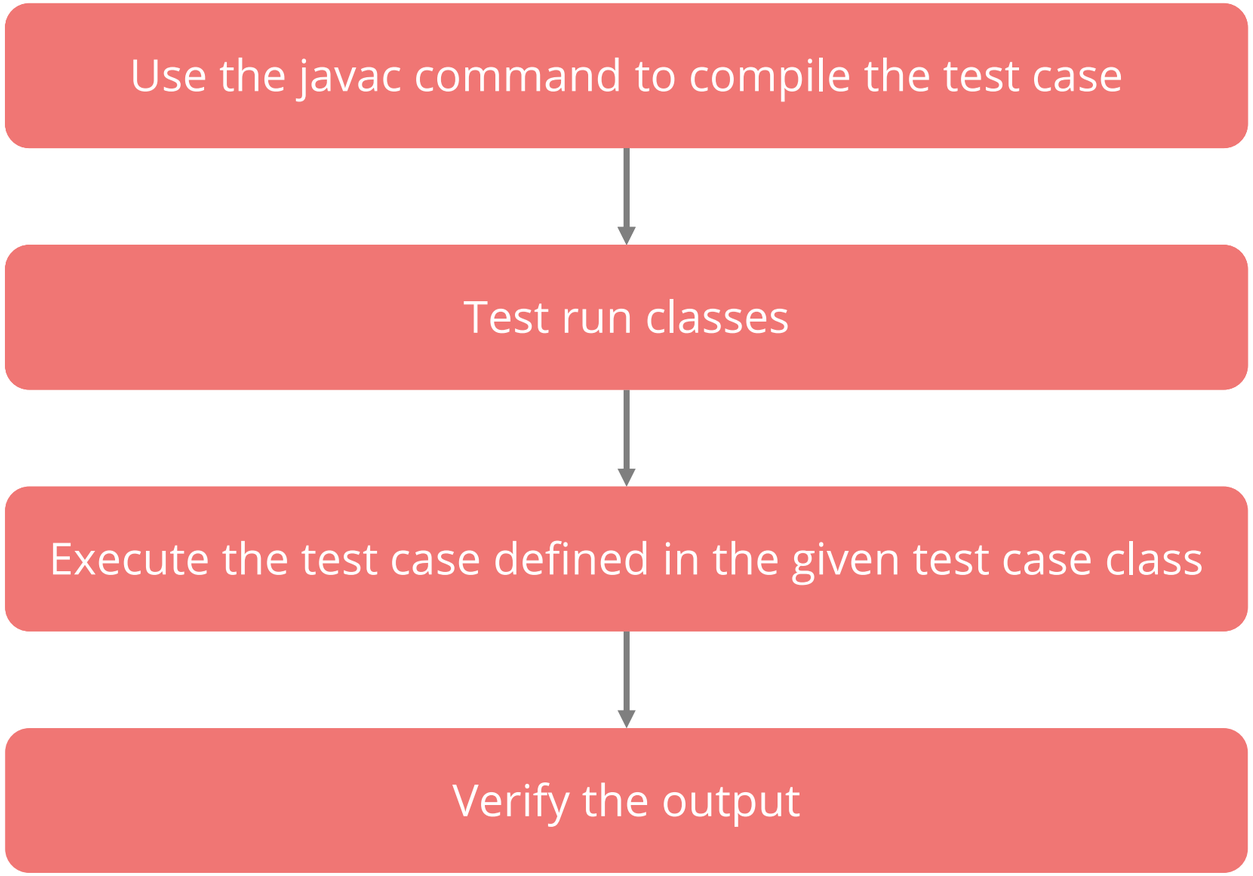
assertTrue and assertFalse

Create a Java class named TestRunner.java in C:\> YourTestDirectory to run the test cases:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
 public static void main(String[] args) {
  Result result = JUnitCore.runClasses(MyAssertionTest.class);
  for (Failure failure: result.getFailures()) {
   System.err.println("Test Case Failed: "+failure.toString());
  }
  System.out.println("Test Case Success:
"+result.wasSuccessful());
 }
}
```

assertTrue and assertFalse

The following are the steps to be performed to work with assertTrue and assertFalse:



Using Various Assertions in JUnit5

Problem Statement:

You have been asked to create various assertions in Junit 5.

ASSISTED PRACTICE

Assisted Practice: Guidelines

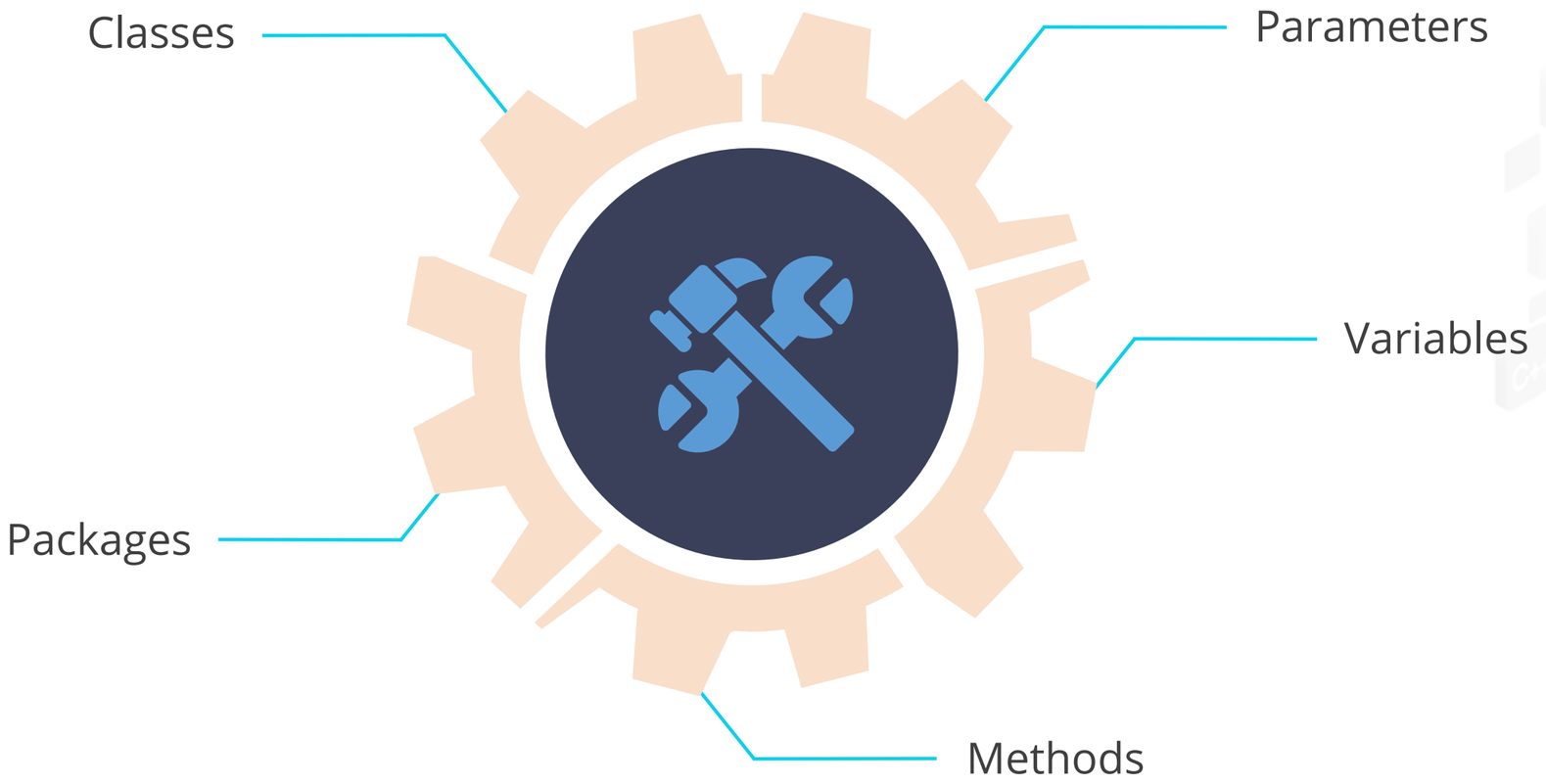
Steps to be followed are:

1. Using **assertEquals()**
2. Using **assertAll()**
3. Using **assertThrows()**
4. Using **assertTimeout()**

JUnit Annotations

JUnit Annotations

JUnit Annotations are a special type of syntactic meta-data that can be put in the Java code.



JUnit Annotations

Annotations allow users to learn and implement a Unit test with ease.

JUnit Annotation

It provides information regarding test methods and specifies:

- Which one will be executed after and before the test methods?
- Which one will be executed after all the test methods?
- Which class or method will be ignored during execution?

JUnit Annotations

@Test

@Before

@After

@Ignore

It provides information that the public void method it is attached to can be executed as a test case.

JUnit Annotations

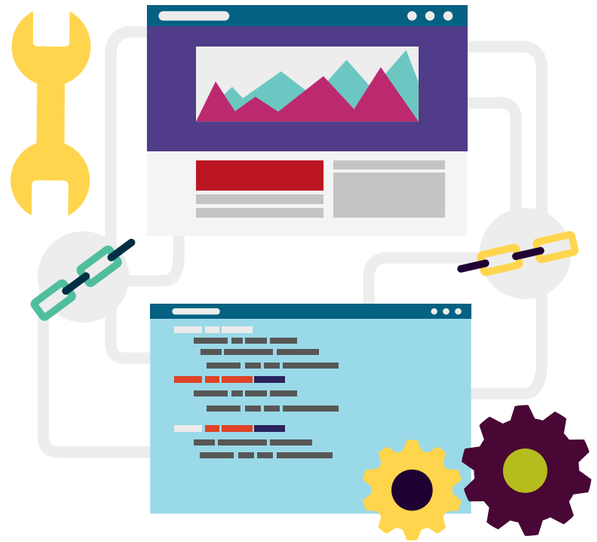
- @Test
- @Before
- @After
- @Ignore

It helps a method to be executed before every test method.

JUnit Annotations

- @Test
- @Before
- @After
- @Ignore

It helps a method to be executed after the test method.



JUnit Annotations

- @Test
- @Before
- @After
- @Ignore

It is used to ignore a test that will not be executed.

JUnit Annotations

Create a Java class named MyAnnotationTest.java in C:\>YourTestDirectory:

```
import org.junit.After;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
public class MyAnnotationTest {
   //execute before test
   @Before
   public void before() {
      System.out.println("before method executed");
   }
   //execute after test
   @After
   public void after() {
      System.out.println("after  method executed");
   }
```

JUnit Annotations

Create a Java class named MyAnnotationTest.java in C:\>YourTestDirectory:

```
 //test case
   @Test
   public void test() {
      System.out.println("test  method executed");
   }
   //test case ignore and will not execute
   @Ignore
   public void ignoreTest() {
      System.out.println("ignore test  method executed");
   }
}
```

JUnit Annotations

Create a Java class named TestRunner.java in C:\>YourTestDirectory for running the annotations:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
 public static void main(String[] args) {
  Result result =
JUnitCore.runClasses(MyAnnotationTest.class);
  for (Failure failure: result.getFailures()) {
   System.err.println("Test Case Failed:
"+failure.toString());
  }
  System.out.println("Test Case Success:
"+result.wasSuccessful());
 }
}
```

JUnit Annotations

01 Compile the test case and test runner class with the help of the javac command

02 Run the test runner, which will execute the test case

03 Verify the output

JUnit Annotations

JUnit offers two special annotations:

@BeforeClass

Helps run a method before executing any test methods

@AfterClass

Performs the method after all the tests have ended

JUnit Annotations

Create a Java class named MyAnnotationTest.java in C:\>YourTestDirectory:

```
import org.junit.AfterClass;
import org.junit.BeforeClass;
public class MyAnnotationTest {
   //execute before class
   @BeforeClass
   public static void beforeClass() {
      System.out.println("before class executed");
   }
   //execute after class
   @AfterClass
   public static void  afterClass() {
      System.out.println("after class executed");
   }
}
```

JUnit Annotations

Create a Java class named TestRunner.java in C:\>YourTestDirectory for running the annotations:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TestRunner {
 public static void main(String[] args) {
  Result result =
JUnitCore.runClasses(MyAnnotationTest.class);
  for (Failure failure: result.getFailures()) {
   System.err.println("Test Case Failed:
"+failure.toString());
  }
  System.out.println("Test Case Success:
"+result.wasSuccessful());
 }
}
```

JUnit Annotations

Compile the test case and test runner class with the help of javac command

Run the test runner, which will execute the defined test case

Verify the output

Using Various Annotations in JUnit5

Problem Statement:

You have been asked to use the various annotations available in Junit for methods and test cases.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

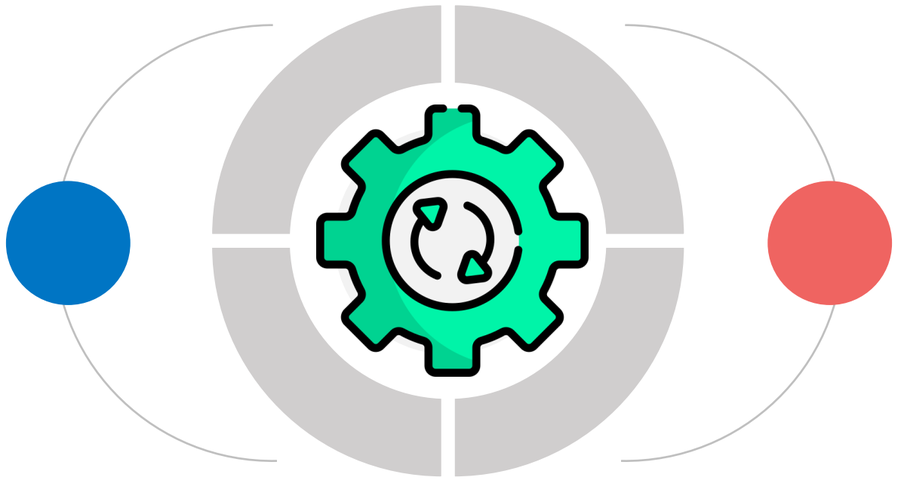
1. Executing **BeforeAll, AfterAll, BeforeEach,** and **AfterEach** annotations
2. Executing **Disabled** annotation
3. Executing **fail()** method

Comparing Arrays in JUnit Tests

Comparing Arrays in JUnit Tests

The Assert class provides assertion methods that help in:

Better structure



Readability of code

Comparing Arrays in JUnit Tests

The assertArrayEquals() method checks whether the two object arrays are equal or not.

If two object arrays are not equal, the method throws an assertion error.

If the actual and expected outputs are null, they are considered equal.

Comparing Arrays in JUnit Tests

assertArrayEquals() method checks if:

01 Arrays have the same number of elements or not

02 All the elements are the same or not

03 There is any mismatch in the order results that can result in failure

Comparing Arrays in JUnit Tests

Example:

```
import org.junit.Test;
import static org.junit.Assert.*;
 public class AssertArrayEqualsExample {
     @Test
    public void myTestMethod(){

        //assume that the below array represents expected
result
        String[] expectedOutput = {"html", "css",
"javascript"};
        //assume that the following array is returned from the
method
        //to be tested.
        String[] output = {"html", "css", "javascript"};
        assertArrayEquals(expectedOutput, output);
    }
}
```

Testing Exceptions in JUnit

Testing Exceptions in JUnit

JUnit:

- Provides an option to trace the exception handling of code
- Enables testing if the code throws a particular exception or not

Testing Exceptions in JUnit

The expected parameter is used with @Test annotation.

Syntax:

```
@Test(expected)
```

Testing Exceptions in JUnit

Create a class to be tested in Java, such as Error.java, in C:\>YourTestDirectory

Add a message for the error condition inside the printError() method:

```
public class Error {
 private String message;
//Constructor
  //@param message to be printed
  public Error(String message){
   this.message = message;
  }
  // prints the message and performs Arithmetic
Division by 0
  public void printError(){
   System.out.println(message);
   int a = 0;
   int b = message.length()/a;
  }
}
```

Testing Exceptions in JUnit

Add an expected ArithmeticException to the testPrint() test case:

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;
public class Test {
  String message = "Testing Division By 0";
  Error error = new Error(message);
  @Test(expected = ArithmeticException.class)
  public void testPrint() {
   System.out.println("Inside testPrint()");
   error.printError();
  }
}
```

Testing Exceptions in JUnit

Create a Runner class of Java named TestRun in C:\>YourTestDirectory to run the above test case:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TestRun {
   public static void main(String[] args) {
      Result result =
JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Testing Exceptions in JUnit

Compile the error, test case, and test runner classes

Execute the test cases defined in the given test case class

Verify the output

Parameterized Testing

Parameterized Testing

It allows the execution of the same test repeatedly using different values.

Parameterized Testing

Steps to create a parameterized test:

1. Annotate test class using @RunWith(Parameterized.class)
2. Create a public static method annotated using @Parameters
3. Create a public constructor that takes values identical to a single **row** of test data
4. Create an instance variable for every **column** of test data
5. Create the test case(s) using the instance variables as the test data sources

Parameterized Testing

Syntax:

```
@RunWith(Parameterized.class)
public class TestClass {

 @Parameterized.Parameters
   public void testMethod() {
      //…
    }

//….
}
```

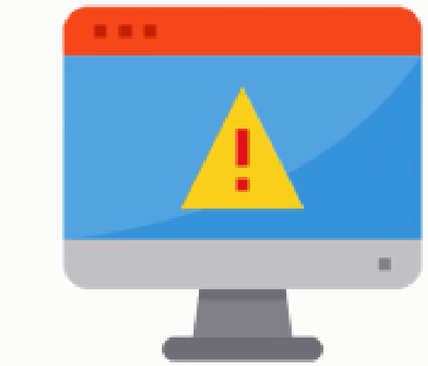
Ignore Test

Ignore Test

If the code is not ready completely while executing a test case, it results in test failures.

@Ignore annotation

- A test method containing @Ignore will not be executed.
- A test class annotated with @ignore will not execute any method.



Ignore Test

Create a Java class that needs to be tested, say Register.java, in C:\> YourTestDirectory:

```
public class Register {
 private String username;
 private String password;
   //Constructor
   //@param message to be printed
   public Message(String username, String password){
      this.username = username;
      this.password = password;
   }
   // prints username
   public String printUserName(){
      System.out.println(username);
      return username;
   }
   // add "Welcome!" to the User
   public String registerUser(){
      String message = "Welcome!" + username;
      System.out.println(message);
      return message;
   }}
```

Ignore Test

Create a class of Java, say Test.java, in C:\>YourTestDirectory.

testPrintUserName()

Ignore Test

Add @Ignore annotation to method printMessage():

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;
public class Test {
   String username = "leo@example.com";
   String password = "leo123";
   Register register = new Register(username, password);
   @Ignore
   @Test
   public void testPrintUserName() {
      System.out.println("Inside testPrintUserName()");
      String message = "hello";
      assertEquals(message,register.printUserName());
   }
}
```

Ignore Test

Create a Java class named TestRun.java in C:\>YourTestDirectory to execute the test case:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TestRun {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(Test.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Ignore Test

Run the test runner, which will not invoke the printMessage() test

01

Compile the message, test case, and test runner classes

02

03

Verify the output

Ignore Test

Update the test in C:\>YourTestDirectory to ignore all test cases and add @Ignore at the level of class:

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;
@Ignore
public class Test {
//….
 }
```

Ignore Test

Compile the test case with the help of the javac command and keep the test runner:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TesT {
  public static void main(String[] args) {
   Result result = JUnitCore.runClasses(TestJunit.class);

   for (Failure failure : result.getFailures()) {
     System.out.println(failure.toString());
   }
   System.out.println(result.wasSuccessful());
  }
}
```

Ignore Test

Finally:

Execute the test runner

Verify the output

Time Test

Time Test

If a test case takes more than milliseconds, it will automatically mark it as failed.

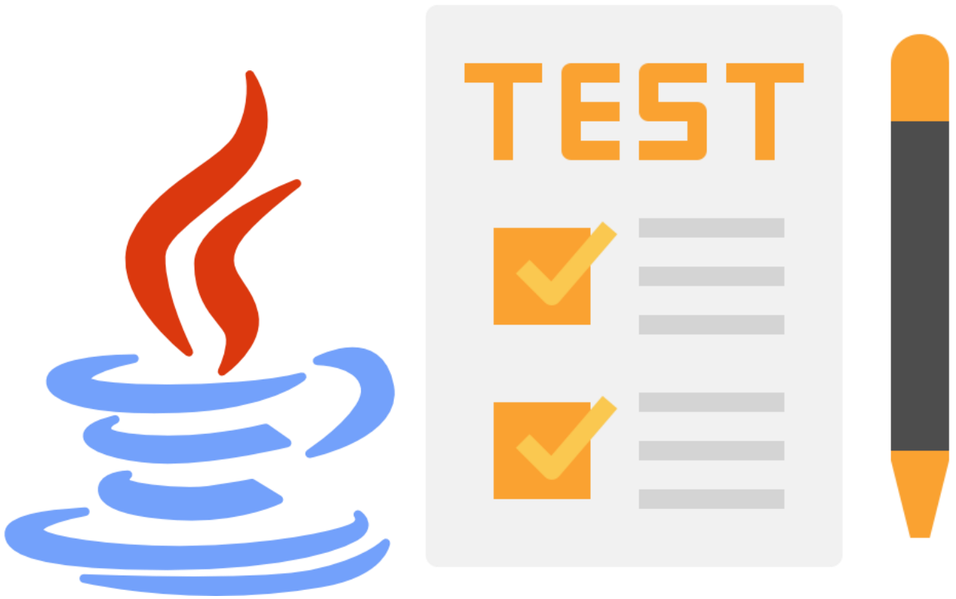
Timeout

Note

The timeout parameter is used with @Test annotation.

Time Test

Create a Java class to be tested, such as Complaint.java, in C:\>YourTestDirectory



Time Test

Add an infinite while loop within the printMessage() method:

```
public class Complaint {
  private String message;
   //@param message to be printed
   public Complaint(String message){
      this.message = message;
   }
   // prints the complaint message
   public void printComplaintMessage(){
      System.out.println(message);
      while(true);
   }
   // add "Thank You!" to the message
   public String acknowledgementForMessage(){
      message = "Thank You!" + message;
      System.out.println(message);
      return message;
   }
}
```

Time Test

Create a Java test class, such as Test.java, in C:\>YourTestDirectory. Add a timeout of 1000:

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;
public class Test {
   String message = "Power Outage in my Home";
   Complaint complaint = new Complaint(message);
   @Test(timeout = 1000)
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      complaint.printComplaintMessage();
   }
   @Test
   public void testAcknowledgementMessage() {
      System.out.println("Inside testAcknowledgementMessage()");
      message = "Thank You!" + message;
      assertEquals(message,complaint.acknowledgementForMessage());
   }
```

Time Test

Create a Java class named TestRun.java in C:\>YourTestDirectory to run the test cases:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class Test {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Time Test

- Compile the message, test case, and test runner classes
- Run the test runner, which will execute the test cases
- Verify the output

Writing Test Cases with Various eCommerce Scenarios

Problem Statement:

You have been asked to write unit test cases with various eCommerce scenarios and configure Junit5 in the project.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Creating a new Maven project
2. Creating an algorithmic method
3. Writing a test case to create an algorithmic method
4. Implementing JUnit5 and configuring the project with a newer version
5. Creating more test cases with output values
6. Packaging application

Key Takeaways

- JUnit is a unit testing framework, and it is crucial for test-driven development.
- Fixtures are fixed states of objects employed as a baseline for running tests.
- The Assert class is used to validate the steps during execution.
- An assertion is used to check if a particular condition or logic returns true or false.
- Parameterized testing allows the execution of the same test repeatedly using different values.

Thank You