INVOICE

Bill to:
Mr. John Doe
215 New Street, City, State
Country, Postal Code

Ship to:
Mr. John Doe
215 New Street, City, State
Country, Postal Code

Qty	Product Description	Price	Total
02	Books	$49	$98
01	Cosmetics	$99	$99
05	Apparel	$29	$145
02	Footwear	$99	$198

Subtotal $500
Tax Rate 0.00%

Thank You! Total $450

If you want more information or need more assistance, go to Help.

Thank you for shopping with us

Check the Invoice

- What do you think is wrong with this invoice?

Display Screen

- Is there any problem with the display of this app?

Credit Card

- How is this credit card different from other credit cards?

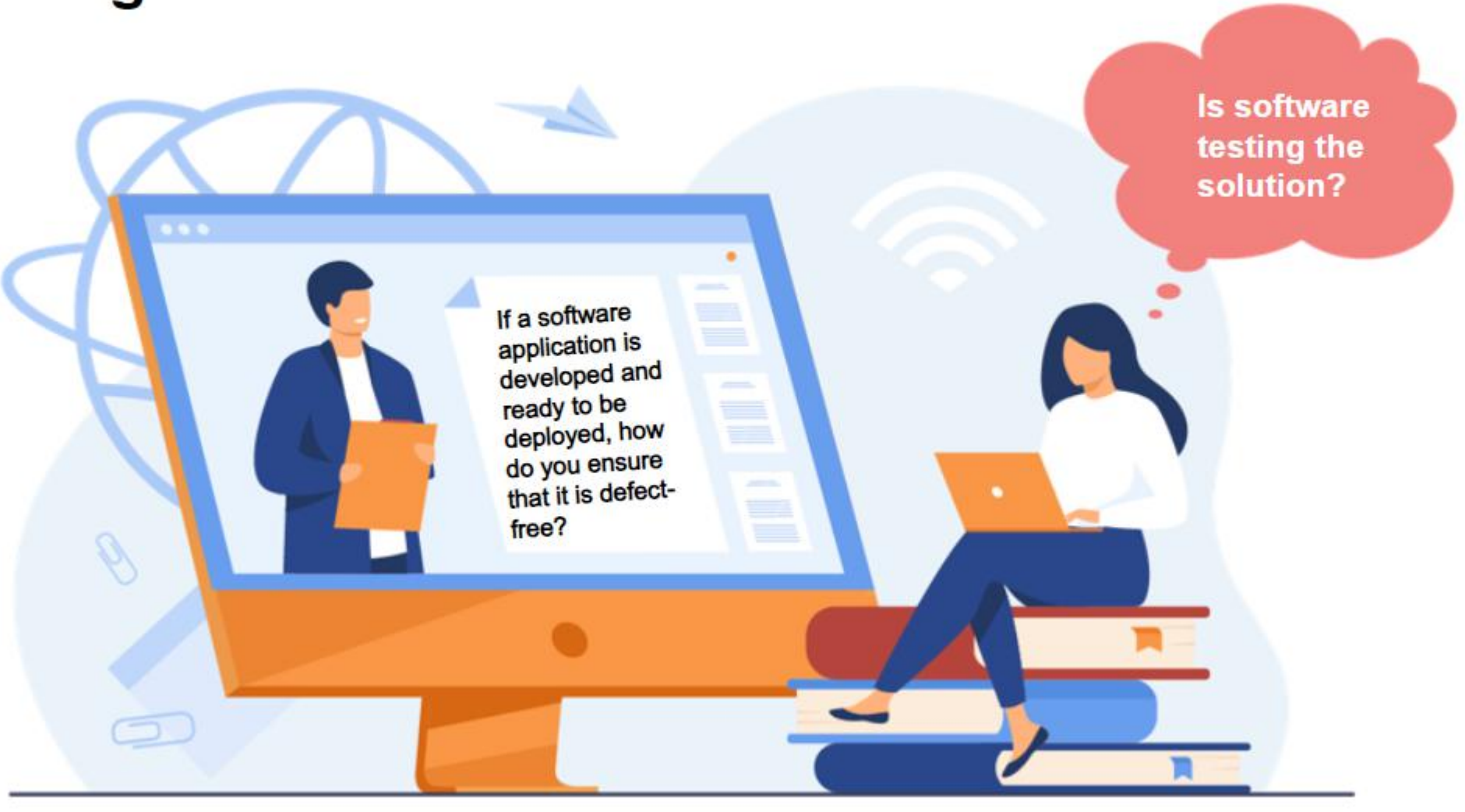
Think and Tell

How do these problems impact consumers and businesses?

When should these defects be identified?

What steps must companies take to avoid such occurrences?

Testing for Defects



Think and Tell

- To test the outcome of the code, `println()` methods are used to print the results on the console.
- In complex applications, results of the Java methods are not always printed on the console. They are provided as inputs to other functions that will require it.
- How can you verify whether the output from one function is valid and can be passed on to the other functions?

Unit Testing With JUnit

Learning Objectives

- Explore software testing
- Explain unit testing and its significance
- Define JUnit architecture
- Test a Java class
- Describe JUnit annotations
- Demonstrate assertion statements
- Write test cases

Software Testing

What Is Software Testing?

- Software testing is a way to check that the finished software meets expectations.
- It also ensures that the software is defect free.
- The purpose of software testing is to identify errors, gaps, or missing requirements in contrast to the actual requirements.
- A properly tested software product ensures reliability, security, and high performance. Consequently, it saves time, is cost effective, and increases customer satisfaction.
- Functional, non-functional and maintenance testing are the different types of Software testing.

Functional Testing

- Functional testing is used to test each function or method of a software application by providing appropriate input and verifying the output.
- Two types of functional testing
  - Unit Testing
  - Integration Testing

In this learning sprint, you will only learn about Unit Testing.

Unit Testing and Its Significance

Unit Testing and Its Significance

- Unit Testing is one of the best development practices used to test smaller units of a code.
- It tests individual units of an application. A unit can be:
  - a method of a class, or
  - a complete class
- It ensures that even the smallest unit of a code is bug-free and reusable.
- It verifies that the code functions efficiently on each unit of application.
- It helps bugs early in the development cycle and saves time and money.
- Unit testing in Java is done by using **JUnit**.

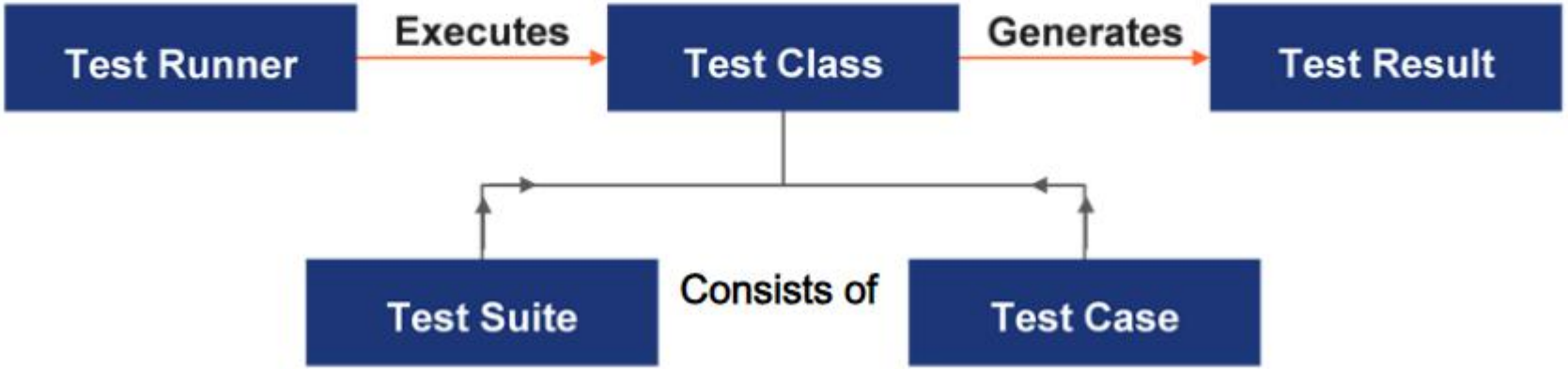
JUnit

- JUnit is an open-source testing tool.
- It is used to write test cases for Java.
- It emphasizes the implementation of Test-Driven Development (TDD).

JUnit Architecture

JUnit Architecture

- The architecture of JUnit refers to the process used by the JUnit framework to execute the tests and display the results.
- **Test Class:** In Java, all code is written inside the class. To test a class JUnit provides a convention to write a separate class and write all the test cases inside it.
- **Test Case:** A single method that basically checks the code logic.
- **Test Suite:** Collection of multiple test cases.
- **Test Runner:** JUnit uses a test runner to automatically run the test case or test suite.
- **Test Result:** Verifies the correctness of test cases and produces a test report.



How Can You Test a Method?

- Given the two input values, the method should be tested in a way that it always returns the sum of the input values.

```
int add (int num1, int num2);
```

- The below table displays some of the input parameters the above method takes, and the output value that the above method returns. It also shows the expected output based on the logic written inside the method. This is an assumption by the programmer.

Input parameters	Output the Method returns	Expected Output	Test
100 , 200	300	300	Pass
-100,-200	-300	-300	Pass
100000, 100000	200000	200000	Pass
500, -500	1000	0	Fail

Testing a Java Class

Steps to Test a Class

- Create the test class.
- Write test cases inside the test class.
- Mark the methods with annotations provided by JUnit.
- Create the object of the class to be tested.
- Call the methods to be tested and verify them inside the specific test cases.
- Use assertion to match the expected and actual output.

Create the Test Class

- The `EvenNumberDemo` class and all its methods need to be tested.

```
public class EvenNumberDemo {
    public boolean isEven(int number) {
        if (number % 2 == 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

- As per convention, the class that will test `EvenNumberDemo` class should have the same name suffixed with the word Test to it.
- So, `EvenNumberDemoTest` will be the name of the testing class.

```
public class EvenNumberDemoTest {

}
```

Annotations

Annotations

- Java Annotation is a tag that represents additional instructions for the compiler.
- Annotations are not part of the code or the logic, but an instruction for the compiler that gives additional information about the class or methods where the annotations are used.
- The extra information includes
  - Name of the class/method/variable
  - Value that the methods of the class will return
  - Type of variable
- The additional instructions can be used by the Java compiler and JVM.
- To indicate that a method is a test case, we will need to provide adequate information to the JVM.
- This is done through annotations provided by the JUnit.

JUnit Annotations

- Some of the annotations provided by JUnit are:
  - `@Test` – This annotation implies that the marked method is a test case.
  - `@Before` – This annotation executes statements before each test case is executed.
  - `@After` – This annotation executes statements after each test case completes its execution.

Assertions

Assert Statements

- When writing test cases there must be one expected output i.e., the assumed output what the method must return based on the logic written inside it. and one actual output i.e., what the method will actually return.

- Assertion helps us to compare the expected output to the actual output and evaluate whether a given test case will pass or fail.
  - **Expected output**

```
public int add(int number1,int number2)
{
return number1+number2;
}
```

    Consider the above method, we expect that if we pass 3, 4 to the method it will return 7. Hence 7 is the expected output, even though the method is not being called.

  - **Actual output**

    The actual output is what the method returns when a call is made to it and parameters are passed.

Assert Statements (contd.)

- JUnit Assert Statements
  - Junit provides a class named Assert, which provides several assertion methods useful in writing test cases.
  - These assertion methods allow you to verify the output of the function being tested with the expected output.
  - The assertion methods are useful in determining the Pass or Fail status of a test case.
  - Some of the assert methods are
    - `assertTrue()`- Tests a single variable to see if its value is either true or false
    - `assertEquals(int expected, int actual)`- Compares two objects for equality
  - More assert methods can be explored in the documentation.

Writing Test Cases

```java
public class EvenNumberDemo {
    public boolean isEven(int number) {
        if (number % 2 == 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

```java
public class EvenNumberDemoTest {
    @Test
    public void isEvenSuccess() {
        EvenNumberDemo evenNumberDemo = new EvenNumberDemo();
        boolean flag = evenNumberDemo.isEven( number: 10);
        assertTrue(flag);
    }
}
```

Test Case - assertTrue

- `EvenNumberDemo` contains an `isEven()` method, the method will return true if the passed integer is even otherwise, it is false.
- `EvenNumberDemoTest` is the testing class that contains `isEvenSuccess()` method to test the `isEven()` method.
- The `isEvenSuccess()` method is annotated with `@Test` to signify that it is a test case.
- The test case performs the following tasks:
  - Creates object of `EvenNumberDemo`.
  - Calls `isEven()` method and stores the value in a boolean variable.
  - Calls the `assertTrue(flag)` method to check the value of the flag.
  - If the flag is true, `assertTrue` is evaluated to be true and the `isEvenSuccess` test case is passed.

```java
public class Calculator {
    1 usage
    public int add(int num1,int num2){
        int result = num1 + num2;
        return result;
    }
}
```

```java
public class CalculatorTest {
    @Test
    public void addSuccess() {
        Calculator calculator = new Calculator();
        int actual = calculator.add(10, 20);
        assertEquals( expected: 30, actual);
    }
}
```

Test Case – assertEquals

- `CalculatorTest` is the testing class that contains the `addSuccess()` method to test the `add()` method of the `Calculator` class
- The `addSuccess()` method is annotated with `@Test` to signify that it is a test case.
- The testing method `addSuccess()` performs the following task
  - Creates object of `Calculator` class.
  - Calls `add()` method passes the parameters and stores the value in an integer variable.
  - As the expected value and actual value are the same, assertEquals() evaluates to be true and hence addSuccess() test case will pass.

```java
public class Calculator {
    1 usage
    public int add(int num1,int num2){
        int result = num1 + num2;
        return result;
    }
}
```

```java
public class CalculatorTest {
    @Test
    public void addSuccess() {
        Calculator calculator = new Calculator();
        int actual = calculator.add(10, 20);
        assertEquals( expected: 30, actual);
    }

    @Test
    public void addFailure() {
        Calculator calculator = new Calculator();
        int actual = calculator.add(10, 20);
        assertNotEquals( unexpected: 25, actual);
    }
}
```

Test Case – assertNotEquals

- `CalculatorTest` is the testing class that contains `addSuccess()` and `addFailure()` method to test the `add()` method of the `Calculator` class.
- Both testing methods are annotated with @Test to signify that they are test cases.
- The test case `addFailure()` performs the following:
  - Creates object of the `Calculator` class.
  - Calls the `add()` method, passes the parameters, and stores the value in an integer variable.
  - As the expected value and actual value are **NOT THE SAME**, `assertNotEquals()` evaluates to true and the `addFailure()` test case will pass.

Think and Tell

- The `addSucess()` and `addFailure()` method perform the repetitive task of initializing the Calculator class object.
  - If we write a method `multiply()` in the Calculator class, should the object of the class Calculator be created multiple times inside the test cases?
  - Can the common feature of initializing the object of the Calculator class be kept separately and all test cases be utilized the same way?

```java
Calculator calculator;
@Before
public void setUp(){
    calculator = new Calculator();

}
@Test
public void addSuccess() {
    int actual = calculator.add(10, 20);
    assertEquals( expected: 30, actual);
}
@Test
public void addFailure() {
    int actual = calculator.add(10, 20);
    assertNotEquals( unexpected: 25, actual);
}
```

Basic setUp Before Testing

The CalculatorTest class contains the following

- `setUp()` –
  - This method is annotated with @Before and will be called before the execution of every test case.
  - This method does precondition work like initializing objects, initializing some values.
- `addSuccess()` and `addFailure()` test cases now only focuses on testing of methods by calling respective assert methods.

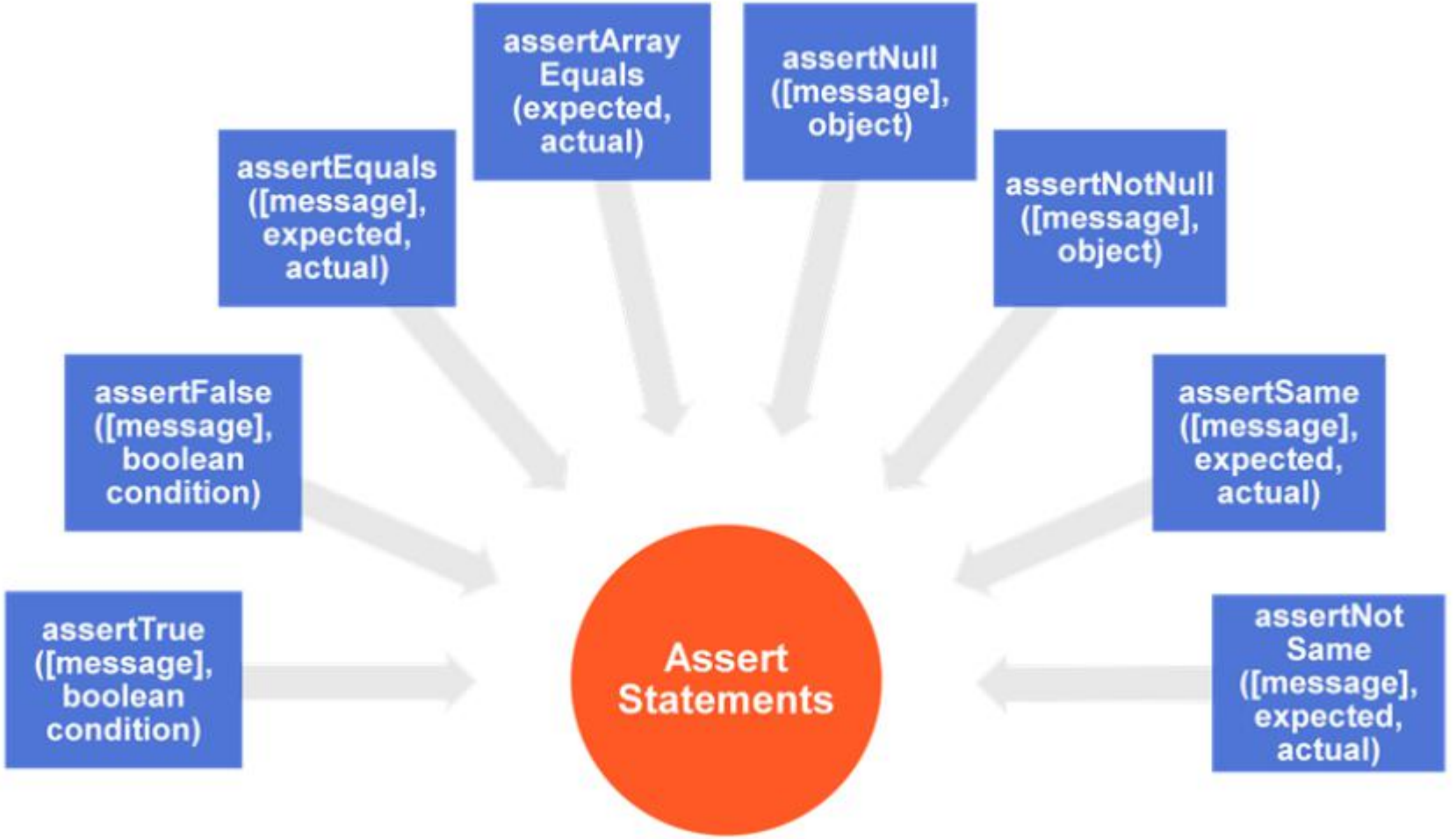
```java
@Before
public void setUp(){
    calculator = new Calculator();

}
@Test
public void addSuccess() {
    int actual = calculator.add(10, 20);
    assertEquals( expected: 30, actual);
}
@Test
public void addFailure() {
    int actual = calculator.add(10, 20);
    assertNotEquals( unexpected: 25, actual);
}
@After
public void tearDown(){
    calculator = null;
}
```

Tear Down After Testing

- `tearDown()` –
  - This method is annotated with @After and is executed after every test case.
  - This method will delete all -temporary files, variables and deallocate the object by setting their reference to null.

A Few More JUnit Assert Statements



Calculator Demo

Write a program to develop a standard calculator that performs the following basic operations:

- Add
- Subtract
- Multiply
- Divide

Write test cases to test each method. For solutions, click here.

The workbench must be used for the demonstration. Execute the test cases provided in the test folder.

DEMO

Even Odd

Write a Java program that has the following functionalities:

1) To test whether the number is even or odd.
2) To check whether the number is a palindrome or not.

Write test cases for both functionalities.

Click here for the solution.
The workbench must be used for the demonstration. Execute the test cases provided in the test folder.

DEMO