**Q-1) What are Junit Test Cases?**

JUnit is a popular testing framework for Java programming language. It is used to write and run repeatable automated tests, to ensure that your code runs as expected. JUnit framework provides annotations to identify test methods and gives assertions for testing expected results.

A JUnit test case is a Java class that contains methods to test a specific piece of your application. Each test method usually tests a single method of the target class. JUnit provides a set of assertions methods that are used to test the application's invariants. These methods can be used to check whether a method returns the expected results, whether a method changes the state of an object in the expected way, or whether a method calls other methods in the expected way.

Here is an example of a simple JUnit test case:

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class TestJunit {

@Test

public void testAdd() {

String str = "Junit is working fine";

assertEquals("Junit is working fine",str);

}

}

In this example, the @Test annotation tells JUnit that the public void method() is a test method. The assertEquals() method is a statically imported method, which comes from the org.junit.Assert class, and it checks that two primitives/objects are equal.

**Q-2) Why do Java projects need Junit test cases?**

JUnit test cases are an essential part of software development, especially in Java projects, for several reasons:

1. **Ensure Code Quality:** JUnit tests help to check if the logic of the written code is correct and if it behaves as expected. They allow developers to ensure that code modifications haven't introduced new bugs in existing functionality.
2. **Facilitate Changes and Simplify Integration:** With a robust suite of unit tests, developers can make changes to their code and quickly know if the change has broken anything. This can be especially useful in an environment where multiple developers or teams are working on the same codebase.
3. **Documentation:** JUnit test cases can serve as a form of documentation to show how a method is supposed to work. New developers on the team can look at the test cases to understand the project requirements and the functionality of methods.
4. **Design:** Writing test cases often helps developers with the design of their code. To write testable code, developers are often encouraged to write small, single-purpose methods, which leads to higher-quality designs and more maintainable code.
5. **Efficiency:** Automated testing with JUnit is much more efficient than manual testing. Automated tests can be easily replicated and they ensure that the same features are tested the same way every time.
6. **Regression Testing:** Once a bug is fixed, JUnit tests can be written to ensure that the same bug doesn't reoccur in the future.

**Q-3) A simple java project with Junit test cases.**

// Main class

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Division by zero is not allowed.");

}

return a / b;

}

}

And here's the updated CalculatorTest class:

// JUnit test class

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calculator = new Calculator();

assertEquals(5, calculator.add(2, 3));

}

@Test

public void testSubtract() {

Calculator calculator = new Calculator();

assertEquals(-1, calculator.subtract(2, 3));

}

@Test

public void testMultiply() {

Calculator calculator = new Calculator();

assertEquals(6, calculator.multiply(2, 3));

}

@Test

public void testDivide() {

Calculator calculator = new Calculator();

assertEquals(2, calculator.divide(6, 3));

}

@Test(expected = IllegalArgumentException.class)

public void testDivideByZero() {

Calculator calculator = new Calculator();

calculator.divide(6, 0);

}

}

In the testDivideByZero test case, we're expecting an IllegalArgumentException to be thrown, which is why we've added the expected = IllegalArgumentException.class parameter to the @Test annotation.