1. **What is Unit Testing? How is it Different from Functional Testing?**

**Unit testing** is the process of testing individual components or small units of code—such as methods or functions—in isolation to ensure they perform as intended. It allows developers to verify the internal logic of a specific unit without interference from the rest of the system. For example, if a calculator has an Add() function, unit testing would check whether this specific method returns the correct sum for given inputs. The main goal is to catch errors early, during the development phase, before integrating the component into the larger application.

On the other hand, **functional testing** evaluates whether the application as a whole performs its intended operations based on the specified business requirements. It checks complete user scenarios and end-to-end workflows, such as logging into an application, performing a transaction, and receiving confirmation. This type of testing ensures that the system behaves correctly from the user's perspective.

The key difference between the two lies in their scope and purpose: **unit testing focuses on the correctness of individual code units**, while **functional testing validates the application's overall behavior and user-facing features**. Unit testing is typically faster and more technical, used by developers, whereas functional testing often involves testers and simulates real-world use.

**2. Types of Testing**

Unit Testing – Test small, individual methods.

Functional Testing – Test whether full features or business flows work.

Automated Testing – Let the computer run your tests without manual effort.

Performance Testing – See how fast your code runs under pressure or heavy load.  
  
  
 **Why Automated Testing Helps**

**Automated testing** is a powerful approach that allows tests to run automatically whenever changes are made to the codebase. This eliminates the need for repetitive manual testing, saving time and effort, especially during frequent development cycles. Automated tests provide quick feedback, making it easier to catch bugs early before they reach production. They also ensure consistency, as the same tests can be executed repeatedly without variation. This improves reliability, reduces human error, and increases overall confidence in the system. As a result, automated testing plays a crucial role in maintaining high-quality, efficient, and maintainable software.  
  
**What is Loosely Coupled & Testable Design?**

A **loosely coupled design** means that different parts of the code are not tightly connected, allowing each part to function independently. This makes the code easier to test, modify, and reuse. In contrast, tightly coupled code is harder to manage because changes in one part can affect others.

A **testable design** allows individual components to be tested easily, often by using techniques like **dependency injection**, where external dependencies are passed into a class rather than being created inside it. This way, during testing, you can replace real dependencies (like a database or email service) with mock objects. For example, a calculator that receives its input through arguments is more testable than one that fetches data from a database directly. Overall, loosely coupled and testable designs lead to more maintainable and reliable code.

**Write Your First Unit Test – Calculator Add Operation**

public class CalculatorTests

{

[Test] // Marks this method as a test

public void Add\_ReturnsCorrectSum()

{

var calc = new Calculator();

Assert.That(calc.Add(2, 3), Is.EqualTo(5));

}

}

**Q.Why Use [SetUp], [TearDown], [Ignore]?**

[SetUp]: Runs before each test. Good for setting up your calculator.

[TearDown]: Runs after each test. Good for cleanup/reset.

[Ignore]: Temporarily skip a test without deleting it.

Code:

[SetUp]

public void Init() => calc = new Calculator();

[TearDown]

public void Cleanup() => calc = null;

[Test, Ignore("Not ready yet")]

public void Subtract\_WillBeImplementedLater() {}

**Q. Why Use Parameterized Test Cases?**

[TestCase(2, 3, 5)]

[TestCase(-1, -1, -2)]

[TestCase(0, 0, 0)]

public void Add\_WorksForMultipleInputs(int a, int b, int expected)

{

Assert.That(calc.Add(a, b), Is.EqualTo(expected));

}