QC-2 Testing

What is Testing?

Testing is the process of evaluating a software application or system to ensure that it meets the required specifications, works as expected, and is free from defects or bugs. The primary goal of testing is to identify and report defects or bugs, so that they can be fixed before the software is released to the end-users.

-Unit Testing -Feature Branch

-Integration Testing - SIT BRANCH

-User Acceptance Testing -UAT Branch

-Production Branch

Why is Testing Necessary?

Testing is necessary for several reasons:

1. **Ensures Quality**: Testing helps to ensure that the software meets the required quality standards, is reliable, and performs as expected.
2. **Reduces Costs**: Identifying and fixing defects early in the development cycle reduces the overall cost of software development and maintenance.
3. **Improves User Experience**: Testing helps to ensure that the software is user-friendly, intuitive, and provides a good user experience.
4. **Reduces Risk**: Testing helps to identify potential risks and defects that could lead to system failures, data breaches, or other security issues.

Manual Testing

Manual testing involves testing a software application or system manually, using human testers to execute test cases and report defects. Manual testing is typically used for:

- Exploratory testing
- Usability testing
- User acceptance testing (UAT)
- Testing complex business logic

Automation Testing (you write a code to test a code)

Automation testing involves using software tools to execute test cases and report defects. Automation testing is typically used for:

- Regression testing
- Functional testing
- Performance testing
- Load testing

Types of Testing

There are several types of testing, each with its own objectives, scope, and methodologies. Here are some of the most common types of testing:

1. Unit Testing

Unit testing involves testing individual units of code, such as functions, methods, or classes, to ensure they work as expected. The goal is to isolate each unit and verify its behavior.

Example: Testing a single function that calculates the area of a rectangle.

2. Integration Testing

Integration testing involves testing how different units of code work together to ensure they integrate seamlessly. The goal is to verify that the interactions between units are correct.

Example: Testing a login feature that involves multiple components, such as authentication, database, and UI.

3. System Testing

System testing involves testing the entire software system, from end to end, to ensure it meets the specified requirements. The goal is to verify that the system works as expected in a real-world scenario.

Example: Testing a complete e-commerce website, including user registration, login, shopping cart, payment gateway, and order processing.

4. Acceptance Testing

Acceptance testing involves testing the software system to ensure it meets the acceptance criteria defined by the customer or end-user. The goal is to verify that the system meets the user's expectations.

Example: Testing a software system to ensure it meets the requirements specified in the contract or agreement.

5. Regression Testing

Regression testing involves re-executing a series of tests to ensure that changes made to the software system have not introduced new bugs or affected existing functionality. The goal is to verify that the changes have not broken existing functionality.

Example: Testing a software system after a new feature is added to ensure that the existing features still work as expected.

6. Exploratory Testing

Exploratory testing involves testing the software system without a pre-defined set of test cases or scripts. The goal is to discover new defects or issues that may not be caught through scripted testing.

Example: Testing a software system by trying out different scenarios, inputs, and workflows to see how it behaves.

7. Usability Testing

Usability testing involves testing the software system to ensure it is user-friendly, intuitive, and easy to use. The goal is to verify that the system meets the user's expectations in terms of usability.

Example: Testing a software system to ensure that the user interface is easy to navigate, and the user can complete tasks efficiently.

8. Performance Testing

Performance testing involves testing the software system to ensure it meets the required performance criteria, such as response time, throughput, and resource utilization. The goal is to verify that the system can handle the expected load and traffic.

Example: Testing a software system to ensure it can handle 1000 concurrent users and respond within 2 seconds.

9. Security Testing

Security testing involves testing the software system to ensure it is secure and protected from unauthorized access, data breaches, and other security threats. The goal is to verify that the system is secure and meets the required security standards.

Example: Testing a software system to ensure it is protected from SQL injection attacks and cross-site scripting (XSS) attacks.

10. Compatibility Testing

Compatibility testing involves testing the software system to ensure it works correctly on different environments, platforms, and devices. The goal is to verify that the system is compatible with different configurations.

Example: Testing a software system to ensure it works correctly on different browsers, operating systems, and mobile devices.

11. Interface Testing

Interface testing involves testing the interactions between different components or systems to ensure they communicate correctly. The goal is to verify that the interfaces are correctly implemented.

Example: Testing the interface between a web application and a database to ensure data is correctly exchanged.

12. End-to-End Testing

End-to-end testing involves testing the entire software system, from start to finish, to ensure it works as expected. The goal is to verify that the system works correctly from the user's perspective.

Example: Testing a complete e-commerce transaction, from adding items to the cart to completing the payment.

13)Smoke Testing

Smoke testing is a type of testing that involves testing the most critical and essential functionalities of a software system to ensure it is stable and functional enough to continue testing. The goal is to quickly identify major defects or issues that would prevent further testing.

14) Sanity Testing

Sanity testing is a type of testing that involves testing a software system to ensure it meets the basic requirements and works as expected. The goal is to verify that the system is sane and functional enough to continue testing.

1. What is unit testing, and why is it important in software development?

Answer:
Unit testing is the process of testing the smallest part of an application, typically a single function or method, in isolation from the rest of the application. This ensures that each unit of the code performs as expected.

Real-Life Example:
Imagine a car manufacturing process where each component, like the engine, brakes, and steering, is tested individually before assembling the car. Unit testing is like testing each component separately to ensure it works perfectly on its own.

Technical Example:
In a function that calculates the sum of two numbers, a unit test would check if

the function returns the correct sum for various inputs (e.g., `sum(2, 3)` should return `5`).

2. Explain the difference between a unit test and an integration test.

Answer:
A unit test focuses on testing a single piece of functionality in isolation, while an integration test ensures that different modules or components of the application work together correctly.

Real-Life Example:
Testing a car's engine individually (unit test) versus testing how the engine works with the transmission system (integration test).

Technical Example:
For a shopping cart application:

- **Unit Test:** Testing a function that adds an item to the cart.
- **Integration Test:** Testing the interaction between the cart, inventory, and payment systems to ensure an item is correctly added, inventory is updated, and the total price is calculated.

3. How does unit testing help in isolating defects in source code?

Answer:
Unit testing helps identify defects by focusing on a specific section of code in isolation. When a unit test fails, it points directly to the area of the code that needs attention, making it easier to isolate and fix the defect.

Real-Life Example:
If a car's headlight isn't working, testing the headlight independently helps determine if the issue is with the bulb, wiring, or switch, rather than checking the entire electrical system at once.

Technical Example:
If a test for a function that calculates tax fails, the developer knows that the issue is within that specific tax calculation function, not in other parts of the application.

4. What are mocks and stubs, and how are they used in unit testing?

Answer:
Mocks and stubs are types of test doubles used to simulate the behavior of complex objects or external dependencies during unit testing.

- **Mock:** An object that records its interactions and can verify that expected actions occurred.
- **Stub:** A simple object that provides predefined responses to method calls.

Real-Life Example:
In a science experiment, using a model of the solar system (mock) instead of the actual solar system allows for easy observation and testing.

Technical Example:
In a unit test for a function that sends an email, a stub might simulate the email service to return a success response, while a mock would ensure that the email service was called with the correct parameters.

5. Why is it important to test source code in isolation?

Answer:
Testing code in isolation ensures that the unit test is not influenced by external factors or dependencies. This leads to more reliable and consistent test results, making it easier to identify and fix defects.

Real-Life Example:
Testing a car's braking system on its own ensures that any issues with braking are not influenced by other systems, like the engine or steering.

Technical Example:
Testing a function that calculates discounts should be done without involving the entire shopping cart system, ensuring that any issues with discount calculation are identified without external interference.

6. Can you describe the benefits of unit testing?

Answer:

- **Encourages Writing Testable Code:** Developers write cleaner, modular code that can be easily tested.
- **Isolate Defects in Source Code:** Helps in quickly identifying where a defect is located.
- **Cost-Effective:** Early detection of bugs reduces the cost of fixing them later in the development process.
- **Improves Overall Code Quality:** Encourages good coding practices, leading to more reliable and maintainable code.
- **Confidence When Refactoring:** Unit tests ensure that changes to the codebase don’t introduce new bugs.

7. How does unit testing improve overall code quality?

Answer:
Unit testing improves code quality by encouraging developers to write code that is modular, well-structured, and adheres to the principles of good software design. It also ensures that code changes do not introduce new bugs, as tests will quickly catch any issues.

Real-Life Example:
Maintaining a detailed checklist for a car's maintenance ensures that all parts are regularly checked and serviced, leading to a well-maintained vehicle.

Technical Example:
A codebase with comprehensive unit tests will have fewer bugs and be easier to maintain, as each function is tested independently, and any issues are quickly identified and fixed.

8. What are some challenges you might face when writing unit tests?

Answer:

- **Complex Dependencies:** Testing code with many external dependencies can be challenging.

- **Time-Consuming:** Writing and maintaining unit tests can be time-consuming, especially for large projects.
- **Changing Requirements:** Frequent changes in requirements may lead to test cases that quickly become outdated.
- **Over-Mocking:** Excessive use of mocks can make tests harder to understand and maintain.

Real-Life Example:
Testing a complex machine with many interconnected parts can be difficult because isolating each part for testing might be challenging.

Technical Example:
Writing unit tests for a function that interacts with a third-party API can be challenging, as it requires setting up mocks or stubs to simulate the API responses.

9. What is the role of automated frameworks in unit testing?

Answer:
Automated frameworks, like JUnit for Java, facilitate the writing, organizing, and running of unit tests. They provide tools to execute tests automatically, generate reports, and integrate with continuous integration systems, making unit testing more efficient and consistent.

Real-Life Example:
Using a diagnostic tool in a car to automatically check all systems rather than manually inspecting each part.

Technical Example:
JUnit automatically runs all the test cases in a project and generates a report showing which tests passed or failed, saving developers the time and effort of running tests manually.

10. How would you handle a situation where a unit test fails?

Answer:
When a unit test fails, the first step is to examine the test and the code under test to understand the cause of the failure. The developer should then fix the

issue in the code and re-run the test. If the failure was due to an incorrect test case, the test should be corrected.

Real-Life Example:
If a car's engine fails a diagnostic test, the mechanic would inspect the engine to identify and fix the issue, then re-run the test to ensure the problem is resolved.

Technical Example:
If a test for a function that calculates the total price of items in a shopping cart fails, the developer would review the function’s logic, correct any mistakes, and re-run the test to confirm that the issue is fixed.

1. What is Test-Driven Development (TDD), and how does it differ from traditional development methods?

Answer:
Test-Driven Development (TDD) is a software development approach where tests are written before the actual code. In TDD, developers first write a unit test for a specific feature, then write just enough code to make the test pass, and finally refactor the code while ensuring that it continues to pass the test.

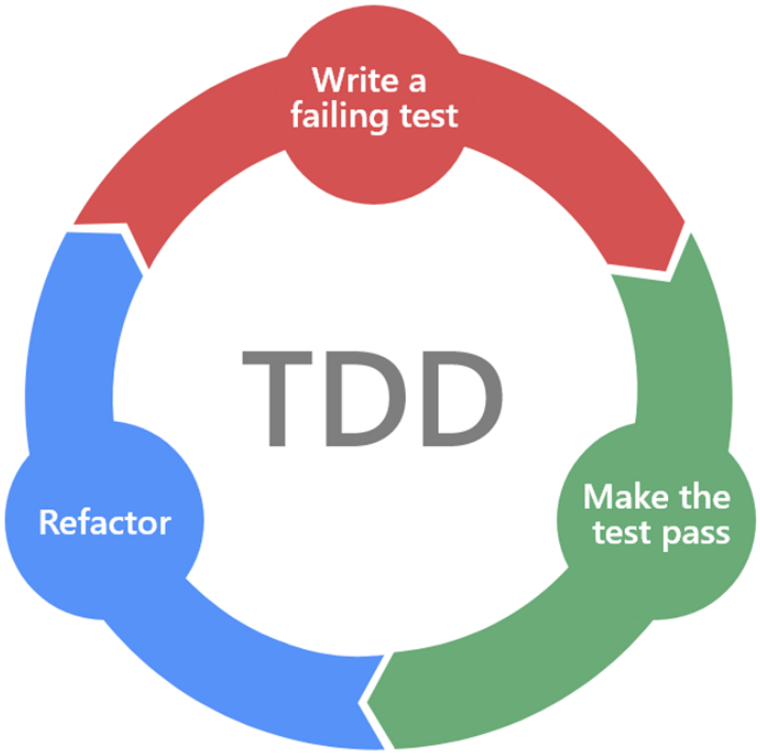
Difference from Traditional Development:

- **Traditional Development:** Code is written first, and tests are written afterward or sometimes not at all.
- **TDD:** Tests are written before writing the code, ensuring that the code meets the specified requirements from the beginning.

Real-Life Example:
Imagine writing a recipe for a cake before baking it. You define the steps (test) before executing them (code), ensuring you don't miss any ingredients or steps.

Technical Example:
In TDD, if you're developing a function to calculate the area of a rectangle, you would first write a test like

`assertEquals(20, calculateArea(4, 5));` before actually implementing the `calculateArea()` function.



2. Describe the steps involved in implementing TDD.

Answer:

The steps in TDD are often summarized as "Red, Green, Refactor":

1. **Write a Test (Red):** Write a failing unit test for a new feature or functionality.
2. **Make the Test Pass (Green):** Write just enough code to make the test pass.
3. **Refactor the Code (Refactor):** Clean up the code while ensuring that it still passes the test

3. What are the advantages of using TDD in software development?

Answer:

- **Fewer Defects:** TDD helps catch bugs early by ensuring that all code is tested from the start.
- **Improved Design:** Writing tests first encourages developers to think about the design and interface before implementation.
- **Confidence in Code Changes:** Developers can refactor code confidently, knowing that the tests will catch any regressions.
- **Documentation:** Tests serve as a form of documentation, showing how the code is expected to behave.

4. What potential hazards might arise when using TDD, and how can they be mitigated?

Answer:

- **Writing Too Many Tests at Once:** This can lead to complex and hard-to-manage test cases. Mitigation: Focus on one small test at a time.
- **Creating Trivial Tests:** Tests that are too simple may not add value. Mitigation: Write meaningful tests that genuinely validate functionality.
- **Broad Tests:** Writing tests that cover too much functionality can make them brittle. Mitigation: Keep tests focused on small, isolated units of functionality.

Real-Life Example:

If a chef skips tasting the soup because it's "just soup," they might miss that it's under-seasoned. Similarly, skipping tests for "simple" code can lead to unnoticed issues.

Real-Life Example:
In a cooking competition, if a chef tries to prepare all dishes at once, they may not focus on the quality of each dish. It's better to prepare one dish at a time with full attention.

Technical Example:
In a login system, writing a broad test that checks both user authentication and

database interaction might fail if any one component has a minor issue. Instead, test authentication and database interaction separately.

5. How does TDD help in reducing the number of defects introduced into the source code?

Answer:
TDD reduces defects by ensuring that every piece of code is written to satisfy a pre-written test. This means that the code is always validated against the test, catching issues early in the development process.

Real-Life Example:
A manufacturer who tests each component of a product before assembly is less likely to encounter defects in the final product.

Technical Example:
In developing a calculator app, writing tests for each operation (addition, subtraction, etc.) before implementing them ensures that each operation works correctly from the start, reducing the likelihood of bugs.

6. Why is it important to write tests before writing the source code in TDD?

Answer:
Writing tests before the code ensures that the code is written with the requirements in mind from the beginning. It also prevents over-engineering, as the developer writes just enough code to pass the test.

Real-Life Example:

If an architect plans the layout of a house before construction, the builders will know exactly what to build, reducing the chance of errors.

What is Junit?

-Automation Testing

JUnit is a open source test automation framework for the Java programming language. JUnit is often used for unit testing. Java Developers use this framework to write and execute automated tests.

Architecture of JUnit 5

JUnit 5 consists of three main components:

1. **JUnit Platform:** The foundation for launching testing frameworks on the JVM. It defines the `TestEngine` API, which is used to discover and execute tests.
2. **JUnit Jupiter:** The combination of new programming and extension models for writing tests and extensions in JUnit 5.
3. **JUnit Vintage:** Provides backward compatibility for running JUnit 3 and JUnit 4 tests on the JUnit 5 platform.

JUnit 4 to JUnit 5 Transition:

1. `@Test`
   - **JUnit 4**: Used to indicate that a method is a test method.
   - **JUnit 5**: Still used in JUnit 5 with the same purpose, but now it's part of the JUnit Jupiter API.
2. `@Before` → `@BeforeEach`
   - **JUnit 4**: `@Before` is used to execute code before each test method.
   - **JUnit 5**: `@BeforeEach` replaces `@Before` and serves the same purpose, but the name reflects that the annotated method is executed before each test method.
3. `@After` → `@AfterEach`
   - **JUnit 4**: `@After` is used to execute code after each test method.
   - **JUnit 5**: `@AfterEach` replaces `@After` and is used to clean up after each test method.

4. `@BeforeClass` → `@BeforeAll`
    - **JUnit 4**: `@BeforeClass` is used to execute code before all test methods in a class. The method must be static.
    - **JUnit 5**: `@BeforeAll` replaces `@BeforeClass`, but the method does not need to be static if the test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)`.
5. `@AfterClass` → `@AfterAll`
    - **JUnit 4**: `@AfterClass` is used to execute code after all test methods in a class. The method must be static.
    - **JUnit 5**: `@AfterAll` replaces `@AfterClass`, with similar requirements as `@BeforeAll`.
6. `@Ignore` → `@Disabled`
    - **JUnit 4**: `@Ignore` is used to skip or ignore a test method or class.
    - **JUnit 5**: `@Disabled` replaces `@Ignore` and serves the same purpose, but with a more descriptive name.
7. `@Category` → `@Tag`
    - **JUnit 4**: `@Category` is used to categorize tests into different groups.
    - **JUnit 5**: `@Tag` replaces `@Category` and allows for the grouping and filtering of tests.

The
`@Test` annotation in JUnit 5 marks a method as a test method that should be executed by the testing framework

@BeforeEach The annotated method will be run before each test method in the test class.

@AfterEach The annotated method will be run after each test method in the test class.

@BeforeAll
The annotated method will be run before all test methods in the test class. This method must be static.

@AfterAll
The annotated method will be run after all test methods in the test class. This method must be static.

`@TestFactory` is used to create dynamic tests, which are generated at runtime. Dynamic tests are not fixed and can be determined based on input, configuration, or other runtime factors

`@Nested` is used to group test cases within a test class. It allows for better organization and readability, especially when testing different aspects of the same subject.

JUnit 5 provides various ways to filter tests, such as using `@Tag` annotations and command-line options.
Mark test methods or test classes with tags for test discovering and filtering

Test Lifecycle in JUnit 5

The test lifecycle includes methods annotated with `@BeforeAll`, `@BeforeEach`, `@AfterEach`, and `@AfterAll` to control the initialization and cleanup processes.

`@TestMethodOrder` is used to define the order of test execution within a test class. The `@Order` annotation specifies the order of individual test methods.

-By default order is arbitrary so that any dependency should not affect any other

`@Disabled` is used to disable a test method or class temporarily. This is useful for tests that are known to fail or are under development.

Using JUnit 5 test suites, you can run tests spread into multiple test classes and different packages. JUnit 5 provides two annotations: @SelectPackages and @SelectClasses to create test suites.

Additionally, you can use following annotations for filtering test packages, classes or even test methods.
@IncludePackages and @ExcludePackages to filter packages
@IncludeClassNamePatterns and @ExcludeClassNamePatterns to filter test classes
@IncludeTags and @ExcludeTags to filter test methods

Assertion:

-Assertions are used to verify that the expected results match the actual results.

-To keep things simple, all JUnit Jupiter assertions are static methods

Assertions.assertEquals() and Assertions.assertNotEquals()
- Assertions.assertArrayEquals()
- Assertions.assertIterableEquals()
- Assertions.assertLinesMatch()
- Assertions.assertNotNull() and Assertions.assertNull()
- Assertions.assertNotSame() and Assertions.assertSame()
- Assertions.assertTimeout() and Assertions.assertTimeoutPreemptively()
- Assertions.assertTrue() and Assertions.assertFalse()

▌Assertions.assertThrows()
▌Assertions.fail()

@BeforeAll
static void setupAll() {
System.out.println("Setting up resources before all tests.");
// Example: Connecting to a database, starting a server, etc.
}

```
// This method is executed once after all tests in this class
@AfterAll
static void tearDownAll() {
    System.out.println("Cleaning up resources after all test
s.");
    // Example: Closing database connections, stopping a serv
er, etc.
}

// This method is executed before each test method in this cl
ass
@BeforeEach
void setup() {
    System.out.println("Setting up before each test.");
    calculator = new Calculator(); // Creating a new instance
for each test
}

// This method is executed after each test method in this cla
ss
@AfterEach
void tearDown() {
    System.out.println("Cleaning up after each test.");
    calculator = null; // Nullifying the instance to clean up
}
```

Assumption :

Assumptions in JUnit 5 are used to set conditions under which a test will be executed. If the assumption fails (i.e., the condition is not met), the test is skipped rather than failed. This is particularly useful when certain tests are only relevant under specific conditions or environments.

Real-Life Scenario:

Imagine you're developing an application that interacts with a specific web service, but the service is only available in the production environment. You might want to run certain tests only when the application is deployed in production.

The **AAA** (Arrange, Act, Assert) pattern is a standard framework for structuring unit tests in a clear and organized way. It helps ensure that tests are easy to read, understand, and maintain by dividing the test logic into three distinct sections:

1. **Arrange:** Set up the initial conditions and inputs needed for the test. This step involves creating and configuring objects, preparing data, and setting up any dependencies or mocks.
2. **Act:** Perform the actual operation that you want to test. This usually involves calling the method or function under test with the arranged inputs.
3. **Assert:** Verify that the outcome of the operation is as expected. This step checks the results and compares them with the expected values using assertions.

Real-Life Analogy:

Imagine you're baking a cake:

- **Arrange:** You gather all the ingredients and prepare the kitchen.
- **Act:** You follow the recipe, mixing the ingredients and baking the cake.

- **Assert:** You check if the cake looks, smells, and tastes as it should.

Technical Example:

Let's say you are testing a `Calculator` class with a method `add(int a, int b)` that returns the sum of two integers.

```
javaCopy code
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @Test
    void testAddition() {
        // Arrange: Set up the calculator and inputs
        Calculator calculator = new Calculator();
        int a = 2;
        int b = 3;

        // Act: Perform the operation
        int result = calculator.add(a, b);

        // Assert: Verify the outcome
        assertEquals(5, result, "2 + 3 should equal 5");
    }
}
```

Explanation:

- **Arrange:** The test begins by setting up the necessary objects and variables (`Calculator calculator = new Calculator();`, `int a = 2;`, `int b = 3;`). This is the preparation phase where you arrange all the elements needed for the test.
- **Act:** The actual method under test is called (`int result = calculator.add(a, b);`). This is the execution phase where the operation you want to test is performed.

- **Assert:** Finally, the test verifies that the result of the operation is correct (`assertEquals(5, result, "2 + 3 should equal 5");`). This is the verification phase where you assert that the outcome matches your expectations.

Benefits of AAA Pattern:

1. **Readability:** The test's structure is clear, making it easier to understand what is being tested, how it is being tested, and what the expected outcome is.
2. **Maintainability:** By separating the different phases of a test, it becomes easier to maintain and modify the tests as the codebase evolves.
3. **Consistency:** Using a consistent structure across tests helps standardize how tests are written and reviewed, leading to fewer errors and better quality tests.

1. Stubs

A **stub** is a simplified version of a method or function that returns predefined responses, rather than interacting with the actual implementation. Stubs are typically used when the code you're testing depends on some other component that you don't want to test at the moment.

2. Mocks

A **mock** is more advanced than a stub. Mocks not only provide predefined responses like stubs, but they also record how they were called. This allows you to assert that certain methods were called with specific parameters, or that they were called a certain number of times.

How do you handle exceptions in junit?

In JUnit, handling exceptions in tests is important to ensure that your code behaves as expected in exceptional conditions. You can handle exceptions in JUnit tests in a few different ways, depending on whether you want to verify that an exception is thrown or handle exceptions within your test code. Here’s how you can handle exceptions in JUnit:

1. Assert Exception Using `assertThrows` (JUnit 5)

In JUnit 5, you can use the `assertThrows` method to assert that a specific exception is thrown during the execution of a block of code. This is the recommended way to test for exceptions in JUnit 5.

2. Using `ExpectedException` Rule (JUnit 4)

In JUnit 4, you can use the `ExpectedException` rule to specify that a certain exception should be thrown by the test method.

3. Try-Catch Block (General Exception Handling)

You can also use a try-catch block within your test method to manually handle exceptions. This approach is less common for asserting exceptions but can be useful in certain situations where you want to perform additional checks or cleanup.