- setup the environment to works with junit library :
- Example :
  - Syntax:
    - Parameters:
    - Returns:
  - Example Usage:
    - Basic Example:
    - Explanation:
  - When Should You Use assertThrows?
  - Comparison with try-catch Blocks:
  - Tips:
  - Syntax:
    - Parameters:
    - Returns:
  - Example Usage:
    - Basic Example:
    - Explanation:
  - Advanced Example with a Return Value:
  - When Should You Use assertTimeout?
  - assertTimeout vs. assertTimeoutPreemptively:
  - Comparison with Manual Timing:
  - Summary:
  - Explanation of the Code:
  - Key Concepts:
    - 1. @TestInstance(TestInstance.Lifecycle.PER_CLASS)
    - 2. Lifecycle Methods:
    - 3. Constructor:
    - 4. Test Methods:
  - Flow of Execution:
  - Output Example:
  - Why Use PER_CLASS Lifecycle?
  - 1. Test Independence:
  - 2. Reliability:
  - 3. Simplicity:
  - 4. Default Behavior:
  - When to Use PER_METHOD:
  - Example of PER_METHOD:

setup the environment to works with junit library :

1. Create a Marvin project
2. Create a class inside the `src/main/java/org.example`
3. right mouse click at the class name,then press *Show context Actions*, then press *Create Test*, then press ok

Example :

```
// src/main/java/org.example/Service
package org.example;

public class Service {
    private String name;

    Service(String name) {
        this.name = name;

    }

    Service() {

    }

    String getName() {
        return name;
    }

    void setName(String name) {
        this.name = name;
    }

}
```

```
// src/test/java/org.example/ServiceTest
package org.example;

import org.junit.jupiter.api.Test;
```

```
import static org.junit.jupiter.api.Assertions.*;
// create a serviceTest class :
class ServiceTest {

    // a unit test for the getName function :
    @Test
    void getNameTest(){
        Service service = new Service("LoadingService");
        // Compare the expected value with the real one. if they are equal, the
test will pass; if it's not the case it will fail.
        assertEquals("LoadingService", service.getName());
    }
    @Test
    void getNameTestSupplier(){
        Service service = new Service("LoadingService");
        // add a custom fail message by implementing the supplier interface :
        assertEquals("LoadingService", service.getName(),()->"Wrong value");
    }
}
```

// Test for Arrays :

```
 @Test
    void test(){
        int []expected = {2,4,6,8};
        int[] actual = {4,8,6,2};
        Arrays.sort(actual);
        // compare the value of the first array with the second array
        assertArrayEquals(expected,actual);

        // compare the reference of the first array with the second array :
         assertEquals(expected,actual);

    }
```

// handling exceptions (try ,catch)

```
 @Test
    void test(){

        try {

            int[] expected = null;

            Arrays.sort(expected);
            int[] actual = {4, 8, 6, 2};
            Arrays.sort(actual);
        assertArrayEquals(expected,actual);
        }catch (Exception e){
            System.out.println("error :");
            fail("null pointer exception");
```

```
        }
    }
```

The `assertThrows` function in JUnit is used to assert that a specific piece of code throws a particular exception. It is typically used in unit tests to verify that methods behave as expected when invalid input or other error conditions occur.

Syntax:

In JUnit 5, the `assertThrows` method is part of the `org.junit.jupiter.api.Assertions` class. The syntax is as follows:

```
Assertions.assertThrows(expectedType, executable);
```

Parameters:

1. **expectedType**: The class of the exception you expect (e.g., `IllegalArgumentException.class`).
2. **executable**: A lambda expression or method reference that contains the code you expect to throw the exception.

Returns:

`assertThrows` returns the exception that was thrown, allowing further assertions on it if needed.

Example Usage:

Basic Example:

```
@Test
void testExceptionThrowing() {
    Exception exception = Assertions.assertThrows(IllegalArgumentException.class,
() -> {
        // Code that should throw the exception
        Integer.parseInt("abc");
    });
```

```
    // Optional: Assert on the exception message or other details
    Assertions.assertEquals("For input string: \"abc\"", exception.getMessage());
}
```

Explanation:

1. The test expects `IllegalArgumentException` to be thrown.
2. If no exception is thrown, or an exception of a different type is thrown, the test will fail.
3. You can also verify the exception message or other details.

When Should You Use `assertThrows`?

Use `assertThrows` in the following scenarios:

1. **Exception Verification**: To verify that a method throws the correct exception for invalid inputs or states.
    - Example: Testing input validation or boundary conditions.
2. **Error Handling Logic**: To confirm your code correctly handles exceptional cases.
3. **Negative Testing**: To ensure certain operations fail as expected under specific conditions.

Comparison with `try-catch` Blocks:

Advantages of `assertThrows`:

- **Conciseness**: No need to write boilerplate `try-catch` blocks.
- **Readability**: Clearly communicates that the focus is on exception handling.
- **Validation**: Built-in assertions on the exception type and message.

Example Without `assertThrows` (using try-catch):

```
@Test
void testExceptionThrowingWithoutAssertThrows() {
    try {
        Integer.parseInt("abc");
        Assertions.fail("Expected an IllegalArgumentException to be thrown");
    } catch (IllegalArgumentException e) {
        Assertions.assertEquals("For input string: \"abc\"", e.getMessage());
```

```
    }
}
```

Using `assertThrows` is cleaner and reduces boilerplate code.

Tips:

- Use specific exception types to make your tests precise and avoid false positives.
- Avoid asserting on the exact exception message unless it's a critical part of your logic, as messages might change over time. The `assertTimeout` function in JUnit is used to verify that a block of code executes within a specified duration. If the code takes longer than the specified time, the test fails. It is helpful for ensuring performance and responsiveness of certain operations.

Syntax:

In JUnit 5, `assertTimeout` is part of the `org.junit.jupiter.api.Assertions` class. The syntax is as follows:

```
Assertions.assertTimeout(Duration timeout, Executable executable);
```

Parameters:

1. **timeout**: The maximum duration allowed for the code execution (e.g., `Duration.ofSeconds(2)`).
2. **executable**: A lambda expression or method reference containing the code to test.

Returns:

The value returned by the executable (if any), or void if none.

Example Usage:

Basic Example:

```
@Test
void testExecutionTime() {
    Assertions.assertTimeout(Duration.ofSeconds(2), () -> {
        // Code that should complete within 2 seconds
        Thread.sleep(1000);
    });
}
```

Explanation:

1. The test expects the code to finish within 2 seconds.
2. If the code takes longer, the test fails.

Advanced Example with a Return Value:

```
@Test
void testExecutionTimeWithReturnValue() {
    String result = Assertions.assertTimeout(Duration.ofMillis(500), () -> {
        // Code that computes and returns a value
        Thread.sleep(300);
        return "Completed";
    });

    Assertions.assertEquals("Completed", result);
}
```

Here, the test ensures the operation finishes within 500ms and asserts the return value.

When Should You Use `assertTimeout`?

Use `assertTimeout` when:

1. **Performance Testing**: To ensure methods execute within acceptable limits.
2. **Preventing Infinite Loops**: To catch unresponsive or hanging code during testing.
3. **Testing Real-Time Systems**: For operations with strict timing constraints.

`assertTimeout` vs. `assertTimeoutPreemptively`:

- `assertTimeout`: The test waits for the code to complete. If it exceeds the duration, the test fails afterward.
- `assertTimeoutPreemptively`: The test aborts the code execution if it exceeds the duration, making it more suitable for long-running tasks.

Example of `assertTimeoutPreemptively`:

```
@Test
void testExecutionWithPreemption() {
    Assertions.assertTimeoutPreemptively(Duration.ofSeconds(1), () -> {
        Thread.sleep(2000); // Test will abort here after 1 second
    });
}
```

Comparison with Manual Timing:

Using `assertTimeout` is better than manually measuring execution time because:

- It integrates directly with JUnit, making the intent clear.
- It simplifies code and improves readability.

Summary:

- **Purpose**: Ensure code executes within a specific duration.
- **Syntax**: `assertTimeout(Duration.ofSeconds(n), () -> code);`
- **Use Cases**: Performance validation, preventing hanging code.
- **Tip**: Use `assertTimeoutPreemptively` for tasks that might hang.

Explanation of the Code:

This code is a JUnit 5 test class for testing the `Shapes` class, which presumably contains methods for computing the area of shapes like squares and circles. The test demonstrates the usage of the `@TestInstance` annotation and JUnit lifecycle annotations such as `@BeforeAll`, `@AfterAll`, `@BeforeEach`, and `@AfterEach`.

Key Concepts:

1. `@TestInstance(TestInstance.Lifecycle.PER_CLASS)`

- By default, JUnit creates a new test instance for every test method (`PER_METHOD` lifecycle).
- With `PER_CLASS`, JUnit creates a single instance of the test class for all tests, reducing overhead and allowing easier sharing of state between tests.
- **Impact**: The constructor (`ShapesTest`) is called **once** for all test methods, not for each.

---

2. Lifecycle Methods:

- `@BeforeAll` **and** `@AfterAll`:
  - These are executed **once** before and after all test methods in the class.
  - In the `PER_CLASS` lifecycle, these methods don't need to be `static` since the test instance is shared.
  - Example:

```
@BeforeAll
void setUpBeforeClass() {
    System.out.println("\n--------shapes Test ---------------\n");
}
@AfterAll
static void setUpAfterClass() {
    System.out.println("\n--------------------------\n");
}
```

- `@BeforeEach` **and** `@AfterEach`:
  - These are executed **before and after each test method**.
  - Used for setting up and cleaning resources specific to each test.
  - Example:

```
@BeforeEach
void init() {
    shapes = new Shapes(); // Initialize Shapes object
    System.out.println("Hello world");
}

@AfterEach
void destroy() {
    shapes = null; // Cleanup
}
```

---

3. Constructor:

The constructor of the test class:

```
ShapesTest() {
    System.out.println("creating new shapes");
}
```

- Demonstrates that with the `PER_CLASS` lifecycle, the constructor is called **once**, unlike the default `PER_METHOD`, where it would be called for every test method.

---

4. Test Methods:

The test methods use the `Shapes` class to test specific functionality:

- `testComputeSquareArea`:

  ```
  @Test
  void testComputeSquareArea() {
      assertEquals(25, shapes.computeSquareArea(5));
  }
  ```

  - Verifies that the `computeSquareArea` method correctly calculates the area of a square with side 5.

- `testComputeCircleArea`:

  ```
  @Test
  void testComputeCircleArea() {
  ```

```
        assertEquals(25, shapes.computeCircleArea(5));
    }
```

- Verifies that the `computeCircleArea` method correctly calculates the area of a circle with radius 5.

Both use the `assertEquals` assertion to compare the expected and actual results.

Flow of Execution:

1. **Test Class Initialization**:
   - Constructor `ShapesTest()` is called once, printing `"creating new shapes"`.
2. **Before All Tests**:
   - `@BeforeAll` method `setUpBeforeClass()` is executed, printing the test header.
3. **For Each Test Method**:
   - `@BeforeEach` method `init()` is executed to initialize the `shapes` object.
   - The test method (`testComputeSquareArea` or `testComputeCircleArea`) runs.
   - `@AfterEach` method `destroy()` is executed to clean up the `shapes` object.
4. **After All Tests**:
   - `@AfterAll` method `setUpAfterClass()` is executed, printing the test footer.

Output Example:

When running the test class, the output will look something like this:

```
creating new shapes

--------shapes Test ---------------
```

```
Hello world
Hello world
--------------------------
```

Why Use `PER_CLASS` Lifecycle?

- **When State Is Shared**: Useful when tests need to share expensive-to-create resources or maintain consistent state.
- **Performance Optimization**: Reduces the overhead of creating and destroying test instances for every method. The default `PER_METHOD` lifecycle in JUnit creates a **new instance of the test class for each test method**. This behavior ensures that **each test is isolated** and independent, making it the preferred lifecycle for most scenarios.

Here's why `PER_METHOD` is commonly used:

---

1. Test Independence:

- Each test starts with a **clean state**, free from any leftover data or side effects from other tests.
- This minimizes unintended interference between tests.
- Example:

```
@Test
void testA() {
    shapes.setColor("red");
}
@Test
void testB() {
    // No risk of "red" from testA affecting this test
    assertNull(shapes.getColor());
}
```

---

2. Reliability:

- Tests remain **reliable** because they don't depend on the state or behavior of other tests.
- If one test fails, it doesn't influence the results of others.

---

3. Simplicity:

- With `PER_METHOD`, you don't need to manually reset shared state after each test.
- The framework automatically ensures each test starts with a fresh instance.

---

4. Default Behavior:

- It's the default lifecycle in JUnit 5, as it aligns with the principle that **tests should not depend on one another**.

When to Use `PER_METHOD`:

- **Stateless Tests**: When each test has its own independent logic.
- **Shared State Is Unnecessary**: If tests do not require shared resources or state.
- **Avoiding Test Pollution**: When avoiding state contamination is critical.

Example of `PER_METHOD`:

```
@TestInstance(TestInstance.Lifecycle.PER_METHOD) // default
class ShapesTest {
    Shapes shapes;

    ShapesTest() {
        System.out.println("Creating new Shapes instance");
    }

    @BeforeEach
    void init() {
        shapes = new Shapes(); // New instance for each test
    }

    @Test
    void testComputeSquareArea() {
```

```
        assertEquals(25, shapes.computeSquareArea(5));
    }

    @Test
    void testComputeCircleArea() {
        assertEquals(78.54, shapes.computeCircleArea(5), 0.01);
    }
}
```

Output for `PER_METHOD`:

```
Creating new Shapes instance
Creating new Shapes instance
```

- A **new instance** is created for each test method.

When to Avoid `PER_METHOD`:

If creating test instances is **expensive** (e.g., initializing large objects or resources), or the tests depend on shared state, consider using `PER_CLASS`.