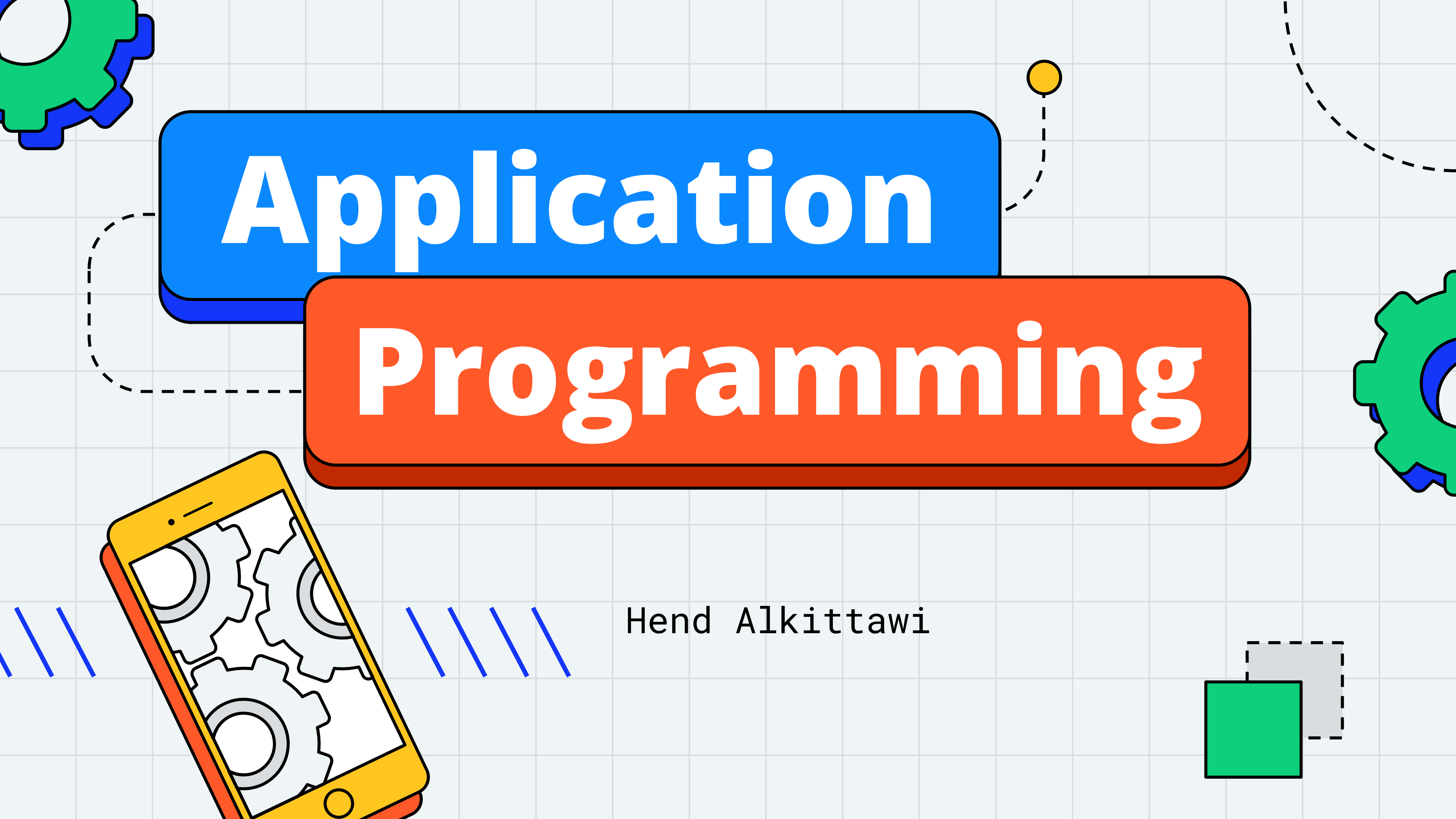
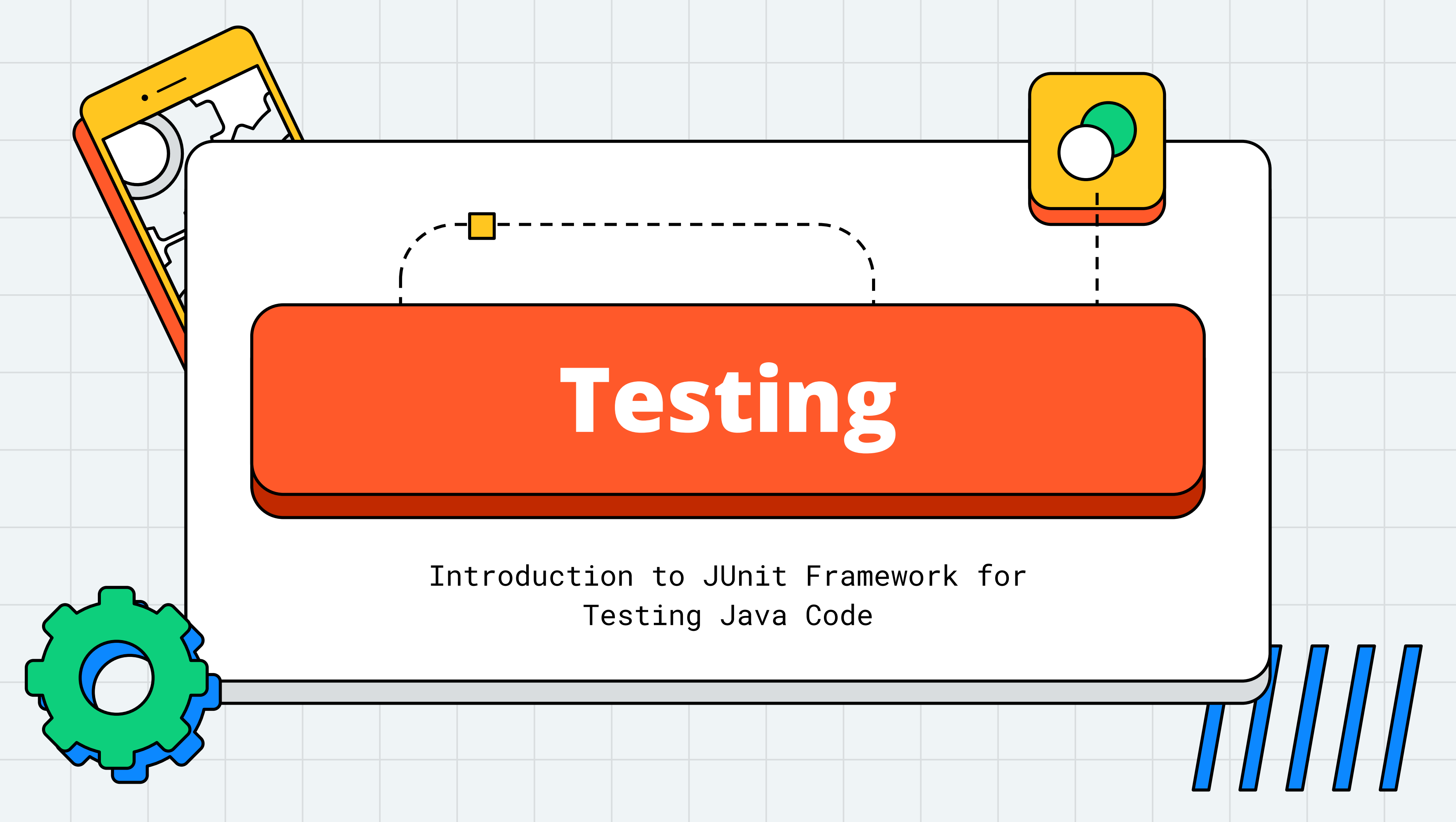
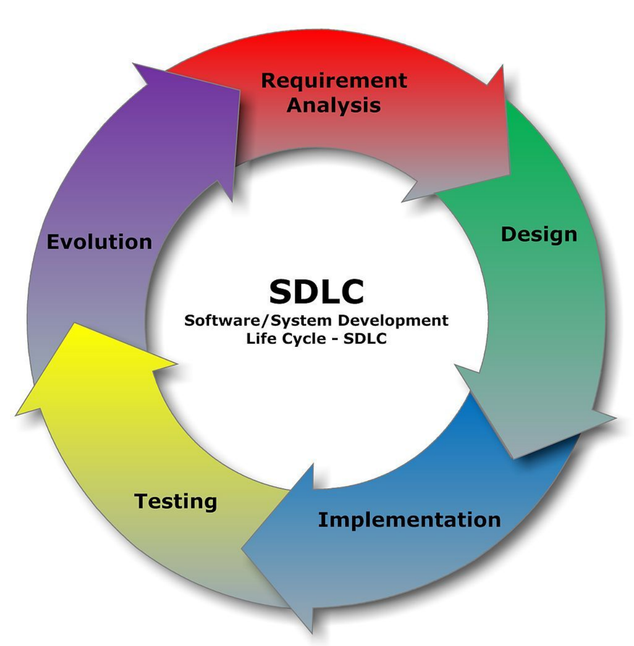

Application
Programming
Hend Alkittawi


Testing
Introduction to JUnit Framework for
Testing Java Code

INTRODUCTION



- Testing in application programming/software development

Image Source: https://commons.wikimedia.org/wiki/File:SDLC_-_Software_Development_Life_Cycle.jpg

INTRODUCTION

- **Unit testing** is a software testing method where individual components of a software application, known as "units", are tested in isolation from the rest of the application.
- A **unit** is typically the smallest testable part of an application, such as a function, method, or class.
- **Unit tests** are designed to validate that each unit of the software performs as expected. These tests are usually **automated** and are written and run by software developers as part of the development process.

UNIT TESTING

- Unit testing is a systematic attempt to reveal errors

I DON'T ALWAYS TEST CODE

BUT WHEN I DO, I DO IT IN PRODUCTION

UNIT TESTING

- Importance of Unit Testing in Software Development
  - Early Detection of Issues
  - Improved Code Quality
  - Reduces Debugging Time
  - Promotes Confidence and Reliability
  - Cost Efficiency
  - Supports Continuous Integration and Continuous Deployment
  - Documentation

UNIT TESTING

- For each method implemented, consider the following when creating the test cases
  - Preconditions: Assumptions/requirements made on the parameters or class variables to be used in the method.
  - Postconditions: Assumptions/requirements made on the returned value (or updated class variables) at the end of the method.

JUNIT TESTING

- JUnit is a widely used open-source testing framework for Java programming language. It provides an easy-to-use framework for writing and running repeatable tests.
- JUnit has become the de facto standard for unit testing in Java, integrated with various development environments and build tools.
- JUnit is linked as a JAR at compile-time
- The framework resides under package org.junit

JUNIT TESTING

- JUnit uses Java **annotations** to define test methods and manage test life cycle events. Key annotations include:
  - @Test: Marks a method as a test method.
  - @Before: Runs before each test method to perform setup.
  - @After: Runs after each test method to perform cleanup.
  - @BeforeClass: Runs once before any of the test methods in the class.
  - @AfterClass: Runs once after all the test methods in the class.
  - @Ignore: Ignores the marked test method.

JUNIT ANNOTATIONS

Annotation	Description
`@Test` `public void method()`	The `Test` annotation indicates that the public void method to which it is attached can be run as a test case.
`@Before` `public void method()`	The `Before` annotation indicates that this method must be executed before each test in the class, so as to execute some preconditions necessary for the test.
`@BeforeClass` `public static void method()`	The `BeforeClass` annotation indicates that the static method to which is attached must be executed once and before all tests in the class. That happens when the test methods share computationally expensive setup (e.g. connect to database).
`@After` `public void method()`	The `After` annotation indicates that this method gets executed after execution of each test (e.g. reset some variables after execution of every test, delete temporary variables etc)
`@AfterClass` `public static void method()`	The `AfterClass` annotation can be used when a method needs to be executed after executing all the tests in a JUnit Test Case class so as to clean-up the expensive set-up (e.g disconnect from a database). Attention: The method attached with this annotation (similar to `BeforeClass`) must be defined as static.
`@Ignore` `public static void method()`	The `Ignore` annotation can be used when you want temporarily disable the execution of a specific test. Every method that is annotated with `@Ignore` won't be executed.

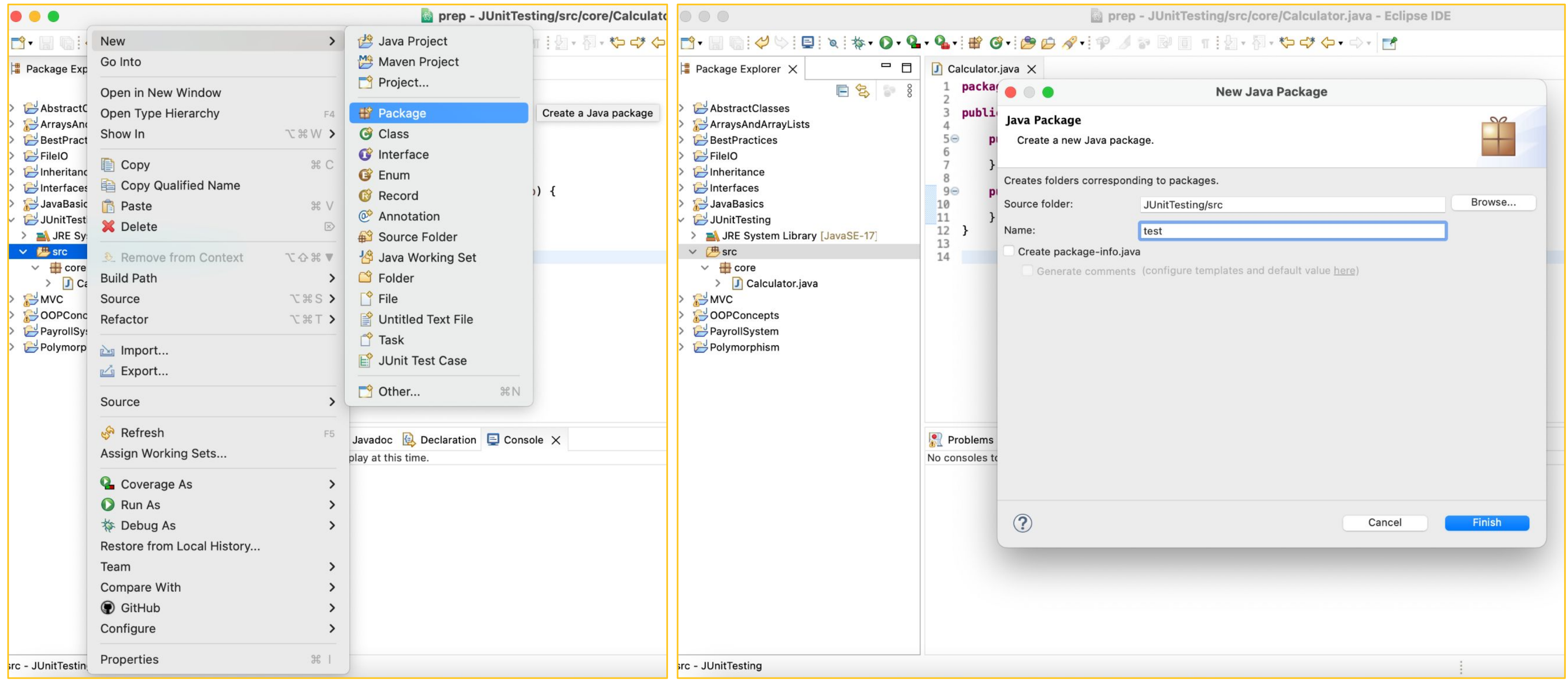
JUNIT TESTING

- JUnit provides a set of **assertion methods** to verify expected outcomes, such as:
  - assertEquals(expected, actual)
  - assertNotEquals(unexpected, actual)
  - assertTrue(condition)
  - assertFalse(condition)
  - assertNull(object)
  - assertNotNull(object)
  - fail(message)

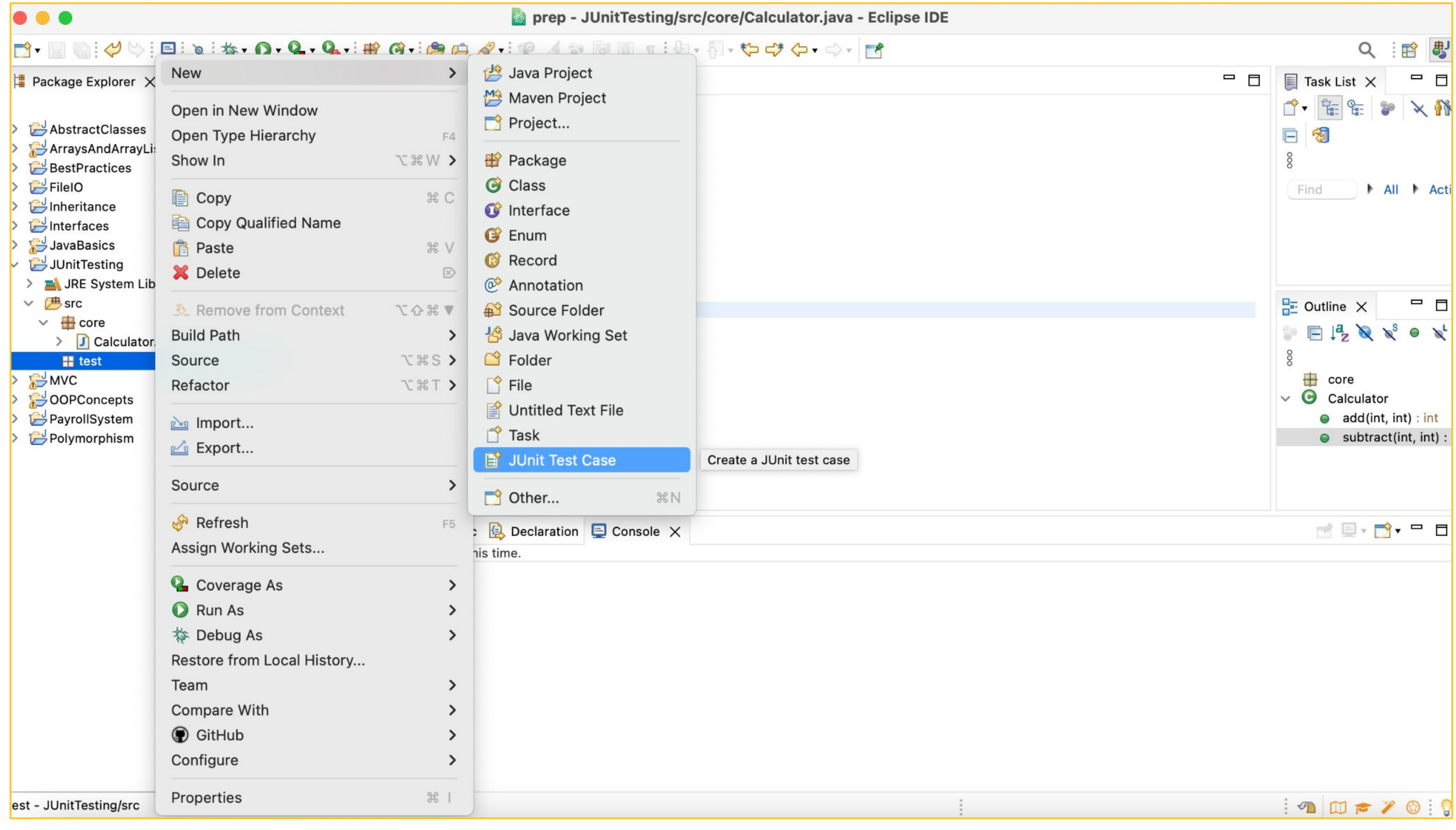
JUNIT IN ECLIPSE

- JUnit integrates seamlessly with IDEs like Eclipse
- To create a JUnit test in Eclipse
  - Create a new package, name it test, under your project (New > Package)
  - Right-click on the test package > New JUnit Test Case
  - On the New JUnit Test Case wizard, select JUnit 4 and fill the fields
  - Select which methods to be tested in the generated class
  - Add the JUnit library to the build path
  - Create the test methods and run the test

Create a new package under your project (New > Package)



Right-click on the test package > New JUnit Test Case



On the New JUnit Test Case wizard, select JUnit 4 and fill the highlighted fields > Next

New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. Specify the class under test to select methods to be tested on the next page.

New JUnit 3 test | New JUnit 4 test | New JUnit Jupiter test

Source folder: JUnitTesting/src | Browse...

Package: test | Browse...

Name: CalculatorTest

Superclass: java.lang.Object | Browse...

Which method stubs would you like to create?

@BeforeClass setUpBeforeClass() | @AfterClass tearDownAfterClass()

@Before setUp() | @After tearDown()

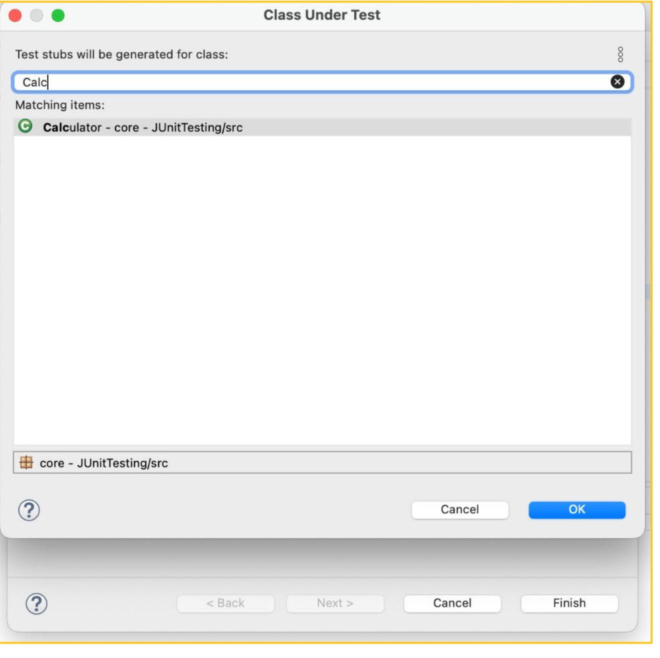
constructor

Do you want to add comments? (Configure templates and default value here)

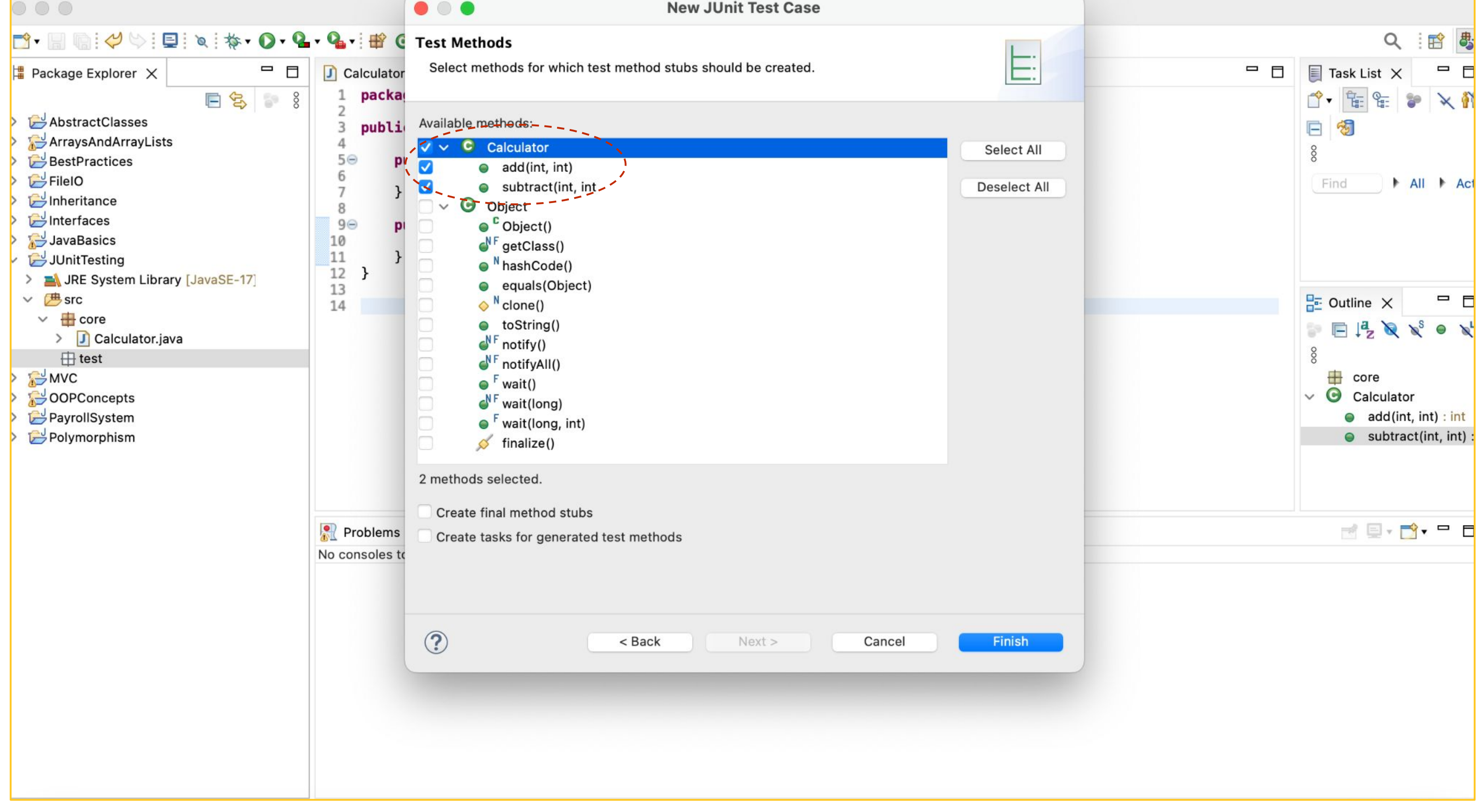
Generate comments

Class under test: | Browse...

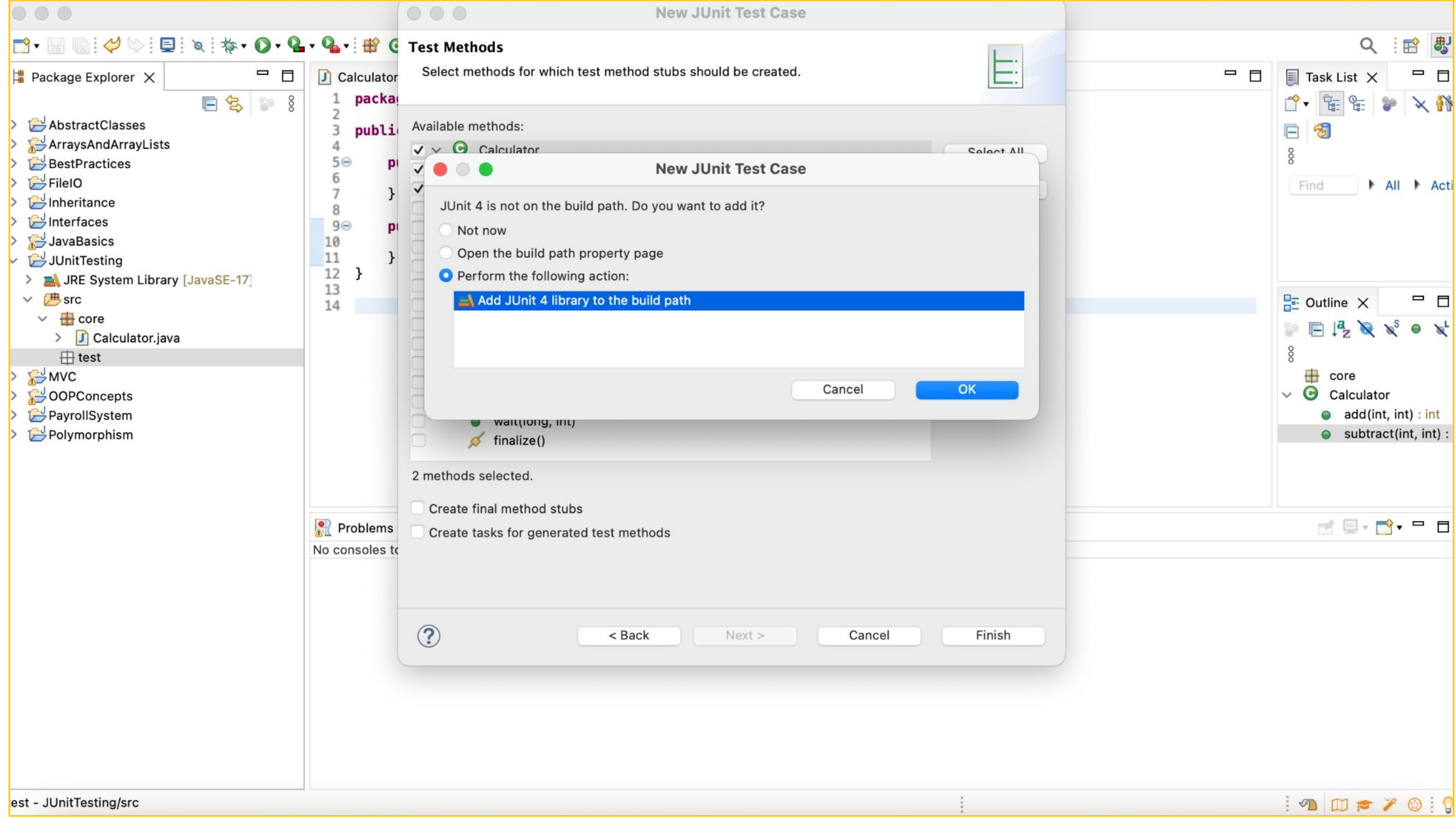
< Back | Next > | Cancel | Finish



Select which methods are to be tested in the generated class > Finish

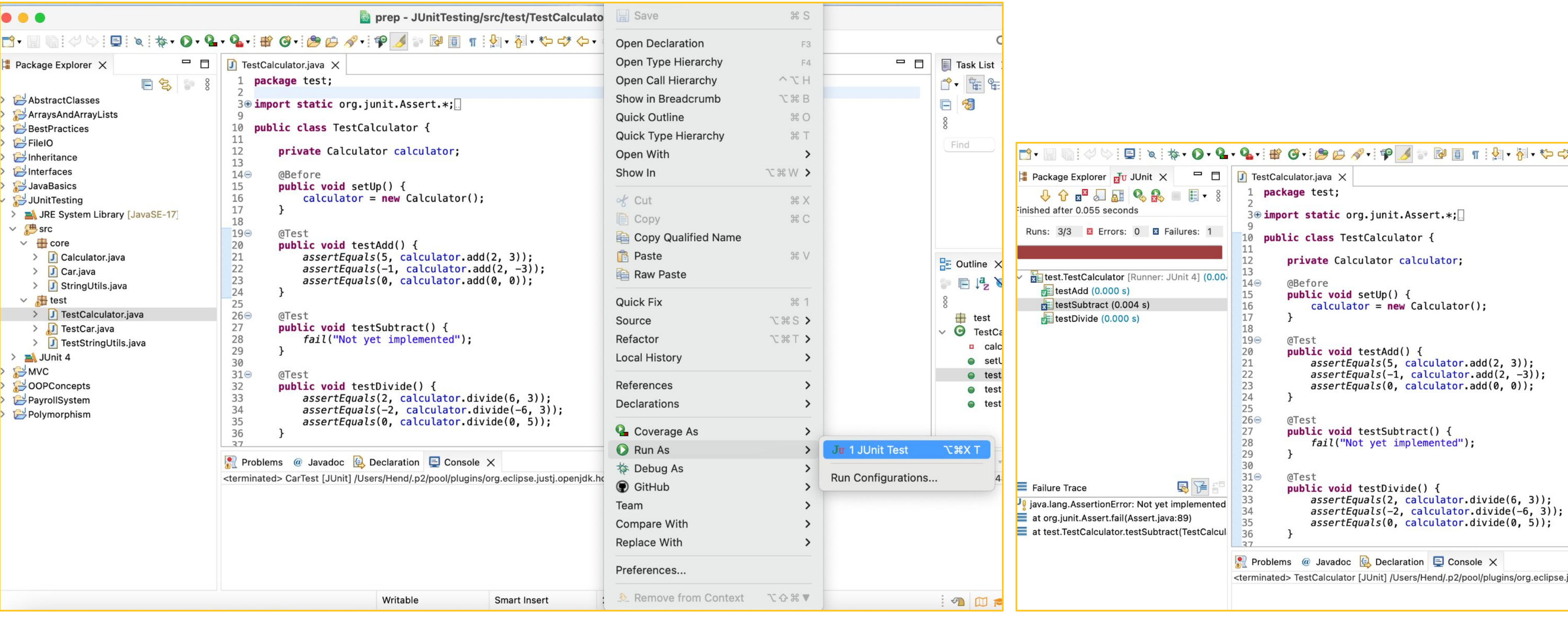


Add the JUnit library to the build path



Create the test methods and run the test

right-click in the text editor > Run As > JUint Test



```java
package core;

public class StringUtils {

  public static String capitalize(String input) {
    return input.toUpperCase();
  }

  public static boolean isPalindrome(String str) {
     if (str == null)
       throw new IllegalArgumentException("Input string cannot be null");

     str = str.toLowerCase();
     int left = 0;
     int right = str.length() - 1;
     while (left < right) {
       if (str.charAt(left++) != str.charAt(right--)) {
         return false;
       }
     }
     return true;
   }
}
```

```java
package test;

public class TestStringUtils {

  @Ignore
  public void testCapitalize() {
    fail("Not yet implemented");
  }

  @Test
   public void testIsPalindromePalindromeString() {
     boolean result = StringUtils.isPalindrome("racecar");
     assertTrue(result);
   }


  @Test
   public void testIsPalindromeNonPalindromeString() {
     boolean result = StringUtils.isPalindrome("hello");
     assertFalse(result);
   }
}
```

```java
package core;

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("Cannot divide by zero");
    return a / b;
  }
}
```

```java
package test;

public class TestCalculator {

  private Calculator calculator;

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

  @Test
  public void testAdd() {
     assertEquals(5, calculator.add(2, 3));
     assertEquals(-1, calculator.add(2, -3));
     assertEquals(0, calculator.add(0, 0));
  }

  @Test
  public void testSubtract() {
    fail("Not yet implemented");
  }

  @Test
   public void testDivide() {
     assertEquals(2, calculator.divide(6, 3));
     assertEquals(-2, calculator.divide(-6, 3));
     assertEquals(0, calculator.divide(0, 5));
   }

   @Test(expected = IllegalArgumentException.class)
   public void testDivisionByZero() {
     calculator.divide(10, 0);
   }
}
```

```java
package core;

public class Car {

  private String make;
  private String model;
  private int year;
  private double fuelLevel;

  public Car(String make, String model, int year) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.fuelLevel = 0.0;
  }

  // getters, setters, and other methods

  public void drive(double distance) {
    if (fuelLevel > 0) {
       fuelLevel -= distance / 10; // Assuming fuel consumption
                                   // rate of 10 units per mile
    }
  }
}
```

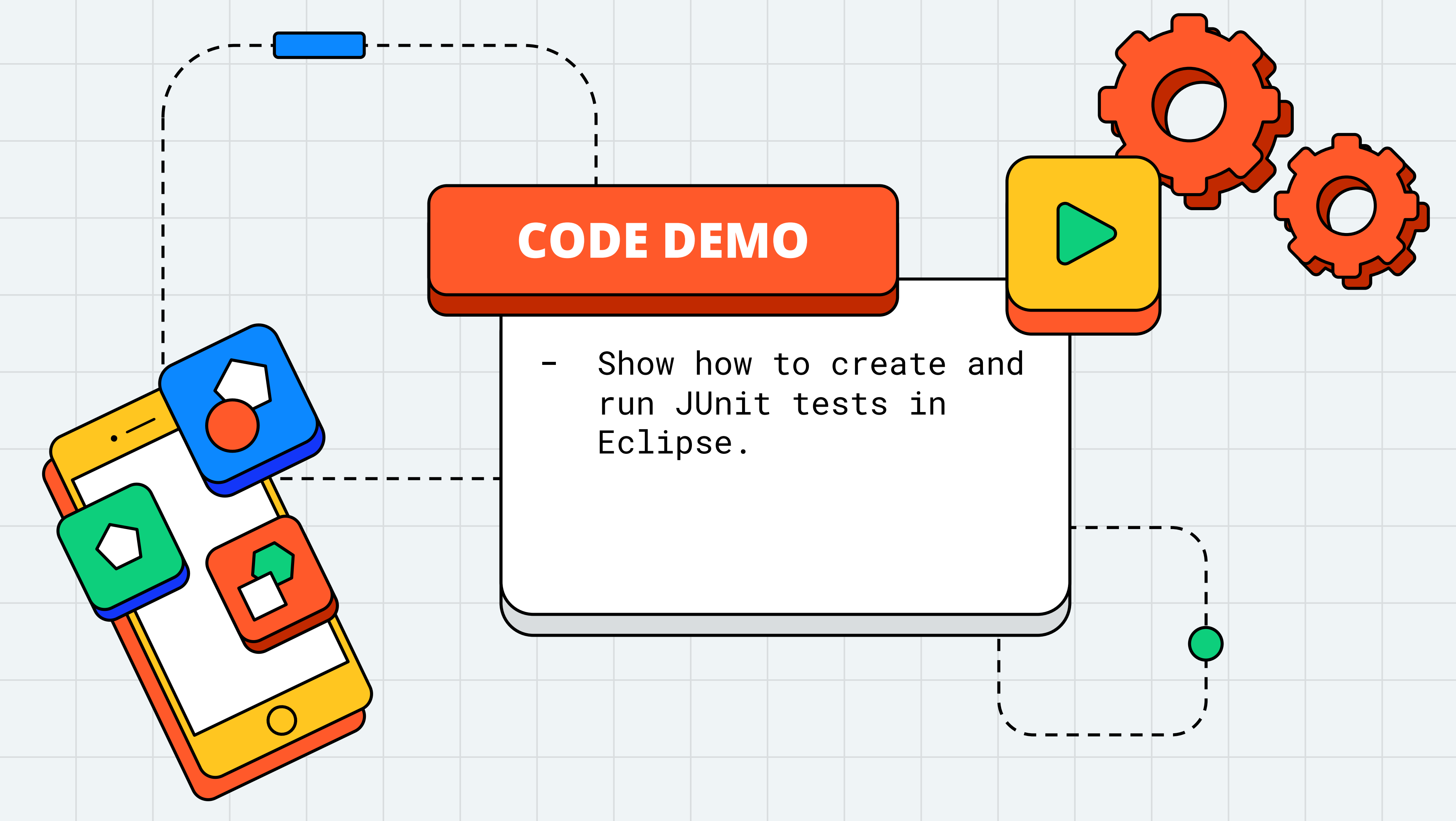
```java
package test;

public class TestCar {

  @Test
  public void testAddFuel() {
    Car car = new Car("Toyota", "Camry", 2022);
    car.addFuel(20.0);
    assertEquals(20.0, car.getFuelLevel(), 0.0);
  }

  @Test
  public void testDriveWithEnoughFuel() {
    Car car = new Car("Honda", "Accord", 2023);
    car.addFuel(30.0);
    car.drive(150.0); // Assuming 150 miles drive
    assertEquals(15.0, car.getFuelLevel(), 0.0);
  }

  @Test
  public void testDriveWithInsufficientFuel() {
    Car car = new Car("Ford", "Focus", 2021);
    car.addFuel(10.0);
    car.drive(150.0); // Assuming 150 miles drive
    assertEquals(10.0, car.getFuelLevel(), 0.0);
  }
}
```


CODE DEMO
- Show how to create and run JUnit tests in Eclipse.

THANK
YOU!
DO YOU HAVE ANY QUESTIONS?
@
hend.alkittawi@utsa.edu
By Appointment
Online