

# PRUEBAS UNITARIAS

Básico



INGENIERÍA DE SOFTWARE

## Registro



# ¿Qué vamos a ver?

- Introducción
  - Pruebas unitarias
  - Unidades
  - JUnit
- Configuración
- Casos básicos
  - Arrange, Act, Assert (AAA)
- Condicionales
- Aserciones básicas
- Excepciones
- Anotaciones básicas

# Introducción

Es el  
momento  
de  
todos

**Bancolombia** 

# Pruebas unitarias

## ¿Qué son pruebas unitarias?

Son pruebas sobre las partes más pequeñas de una aplicación: las **unidades**.

## ¿Qué es una unidad?

Una unidad es la parte *testable* más pequeña de un software.

## ¿Para qué se hacen las pruebas unitarias?

Se hacen para asegurar que las unidades (**funciones**) se comporten como se espera.

# Unidades

En programación orientada a objetos, una función es una unidad.

Unidad

```
@Override
public JSONAPIBody<BalanceResponse> doPost(JSONAPIBody<BalanceRequest> request) {
    SslVerification.disableSslVerification();
    try {
        response = doPostForByDataRequest(request);
    } catch (SystemExceptionMsg systemExceptionMsg) {
        logService.setStackTrace(systemExceptionMsg);
        response = doPostCatchLog(systemExceptionMsg,
            systemExceptionMsg.getFaultInfo().getGenericException().getCode(),
            systemExceptionMsg.getFaultInfo().getGenericException().getDescription(), this.path);
    } catch (BusinessExceptionMsg eMsg) {
        logService.setStackTrace(eMsg);
        response = doPostCatchLog(eMsg, eMsg.getFaultInfo().getGenericException().getCode(),
            eMsg.getFaultInfo().getGenericException().getDescription(), this.path);
    } catch (BusinessExceptionMS e) {
        logService.setStackTrace(e);
        response = doPostCatchLog(null, e.getCode(), e.getMessage(), e.getPath());
    } catch (DatatypeConfigurationException e) {
        logService.setStackTrace(e);
        response = doPostCatchLog(e, constants.getEXCEPTION_ERRORCODE(), e.getMessage(), this.path);
    } catch (WebServiceException e) {
        logService.setStackTrace(e);
        response = doPostCatchLog(e, constants.getError_WebServiceCode(), e.getMessage(), this.path);
    } catch (RESTClientException e) {
        logService.setStackTrace(e);
        response = doPostCatchLog(e, constants.getError_Mapper_Document_Code(), e.getMessage(), this.pat
    } catch (Exception exception) {
        logService.setStackTrace(exception);
        response = doPostCatchLog(exception, constants.getEXCEPTION_ERRORCODE(), exception.getMessage(),
    }

    return response;
}
```

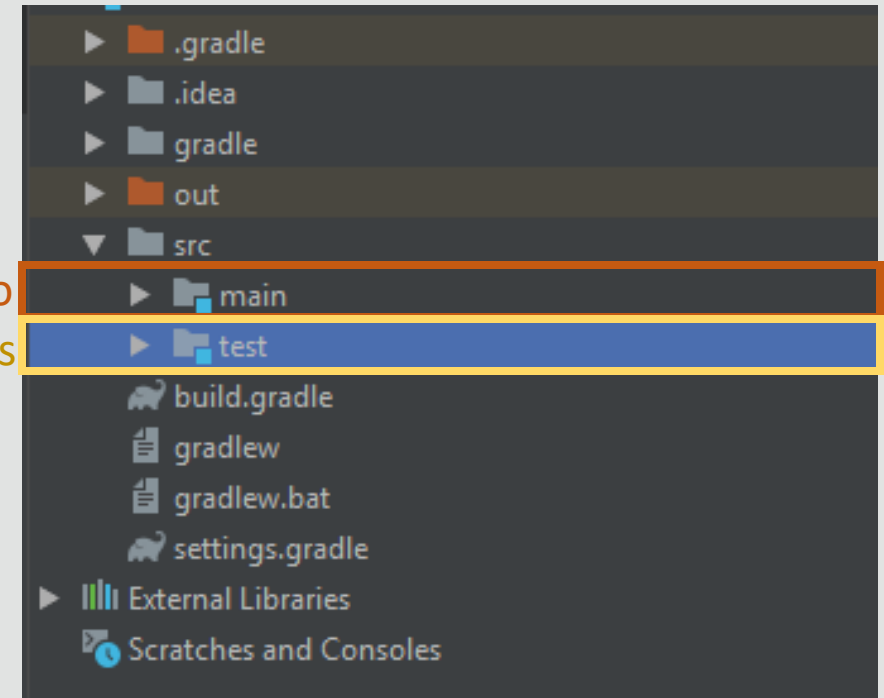
Es el  
momento  
de  
todos

Bancolombia 

# JUnit

- Las pruebas unitarias son **código** escrito en el mismo lenguaje que el software.
- Para correr pruebas unitarias, usualmente se usan frameworks y herramientas.
- El framework más conocido para java es **JUnit**

código  
pruebas



# JUnit

- JUnit es el framework para pruebas unitarias en Java. Viene como un jar que se usa en tiempo de compilación.
- Para usar JUnit, se agrega como dependencia testCompile a la configuración de build de gradle.

```
plugins {  
    id 'java'  
}  
  
group 'IngenieriaSoftware'  
version '1.0-SNAPSHOT'  
  
sourceCompatibility = 1.8  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testCompile group: 'junit', name: 'junit', version: '4.12'  
}
```

Gradle:

testCompile group: 'junit', name: 'junit', version: '4.12'

Es el  
momento  
de  
todos

**Bancolombia**



# JUnit

- Las pruebas unitarias terminan siendo un conjunto de clases que no se usan en tiempo de ejecución del programa
- Están ubicadas en una subcarpeta del Proyecto.
- Terminan usando partes de JUnit y otras herramientas para correr todos los casos de prueba que se programan.
- JUnit a su vez es la herramienta para correr las pruebas.

```
plugins {  
    id 'java'  
}  
  
group 'IngenieriaSoftware'  
version '1.0-SNAPSHOT'  
  
sourceCompatibility = 1.8  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testCompile group: 'junit', name: 'junit', version: '4.12'  
}
```

# Configuración

*git clone* [https://grupobancolombia.visualstudio.com/Vicepresidencia%20Servicios%20de%20Tecnolog%C3%ADa/\\_git/IngSW\\_DojoUnitTestBasic](https://grupobancolombia.visualstudio.com/Vicepresidencia%20Servicios%20de%20Tecnolog%C3%ADa/_git/IngSW_DojoUnitTestBasic)

Es el  
momento  
de  
todos



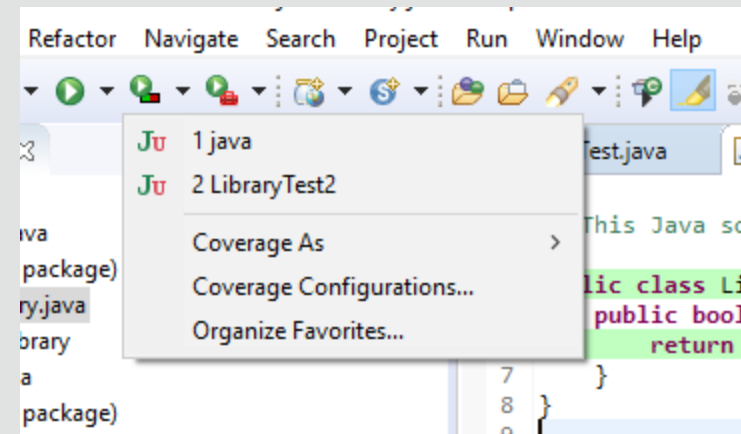
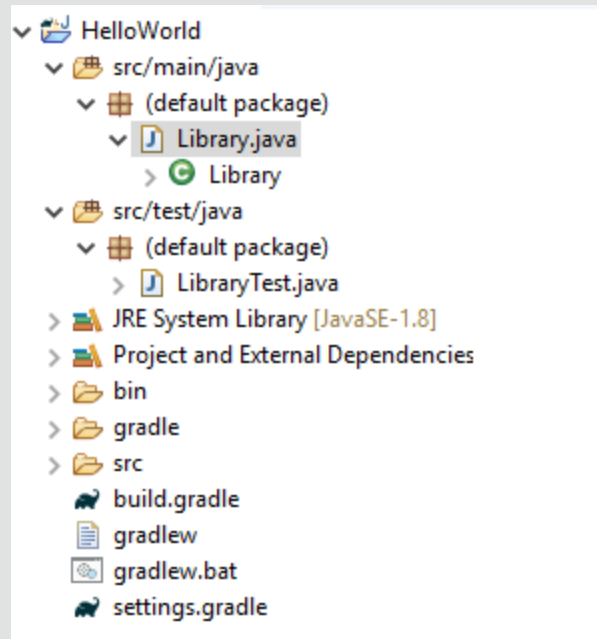
# Eclipse

Al crear un proyecto Java gradle nuevo, Eclipse incluye a JUnit en las dependencias.

```
8
9 plugins {
10     // Apply the java-library plugin to add support for Java Library
11     id 'java-library'
12 }
13
14 dependencies {
15     // This dependency is exported to consumers, that is to say found on their compile classpath
16     api 'org.apache.commons:commons-math3:3.6.1'
17
18     // This dependency is used internally, and not exposed to consumers on their own compile classpath
19     implementation 'com.google.guava:guava:23.0'
20
21     // Use JUnit test framework
22     testImplementation 'junit:junit:4.12'
23 }
24
25 // In this section you declare where to find the dependencies of your project
26 repositories {
27     // Use jcenter for resolving your dependencies.
28     // You can declare any Maven/Ivy/file repository here.
29     jcenter()
30 }
31
```

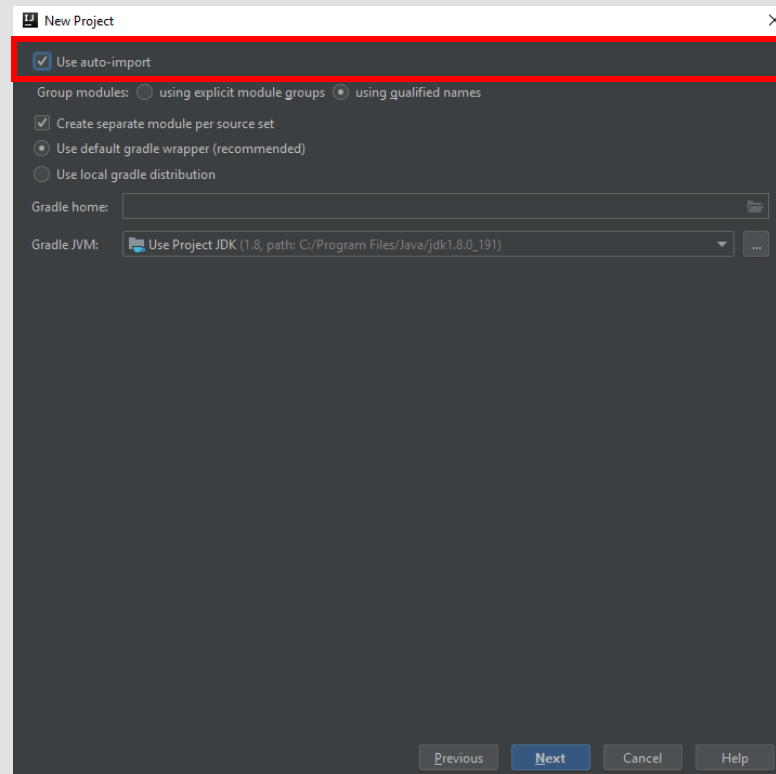
# Eclipse

Adicionalmente, Eclipse se integra con JUnit de forma que crea una estructura de carpetas por defecto para los tests y tiene configurada la tarea que corre todos los tests en esta carpeta.



# IntelliJ IDEA

Al crear un proyecto Java gradle nuevo, se debe seleccionar “Use auto-import” para asegurar que se cree la carpeta src con la estructura por defecto.

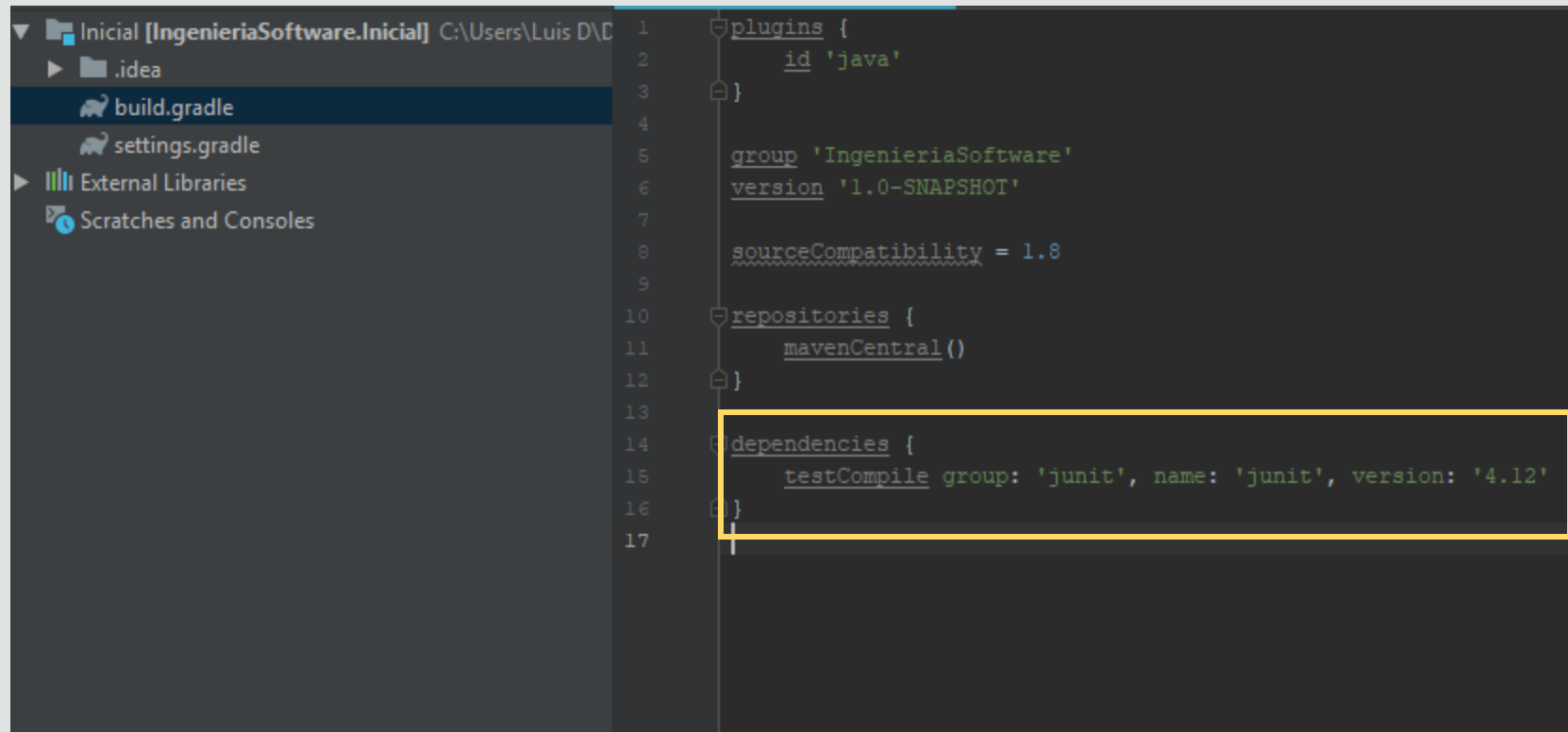


Es el  
momento  
de  
todos

Bancolombia

# IntelliJ IDEA

IDEA genera por defecto el build.gradle, con la dependencia testCompile junit:



```
1 plugins {  
2     id 'java'  
3 }  
4  
5 group 'IngenieriaSoftware'  
6 version '1.0-SNAPSHOT'  
7  
8 sourceCompatibility = 1.8  
9  
10 repositories {  
11     mavenCentral()  
12 }  
13  
14 dependencies {  
15     testCompile group: 'junit', name: 'junit', version: '4.12'  
16 }  
17
```

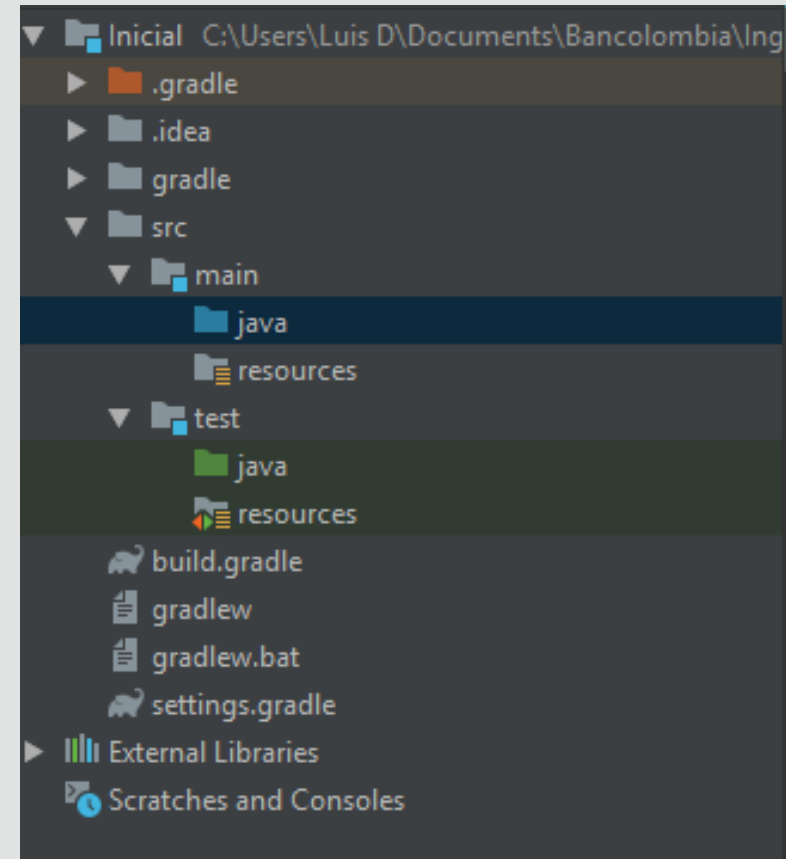
Es el  
momento  
de  
todos



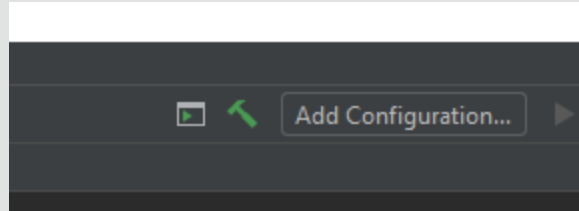
# IntelliJ IDEA

Después de terminar de sincronizar y hacer el build inicial, IDEA crea las carpetas *main* y ***test***, y marca esta última como carpeta principal de pruebas.

Resta crear la configuración para correr con JUnit todas las pruebas unitarias bajo este directorio.

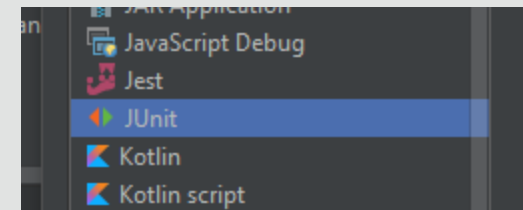
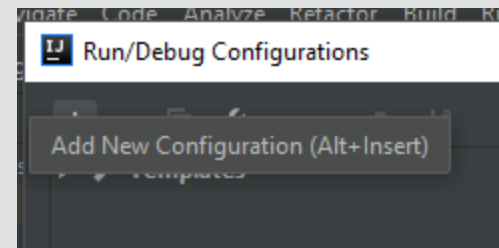


# IntelliJ IDEA



Para esto:

1. Add configuration
2. Add new configuration
3. JUnit



Es el  
momento  
de  
todos

**Bancolombia**



# IntelliJ IDEA

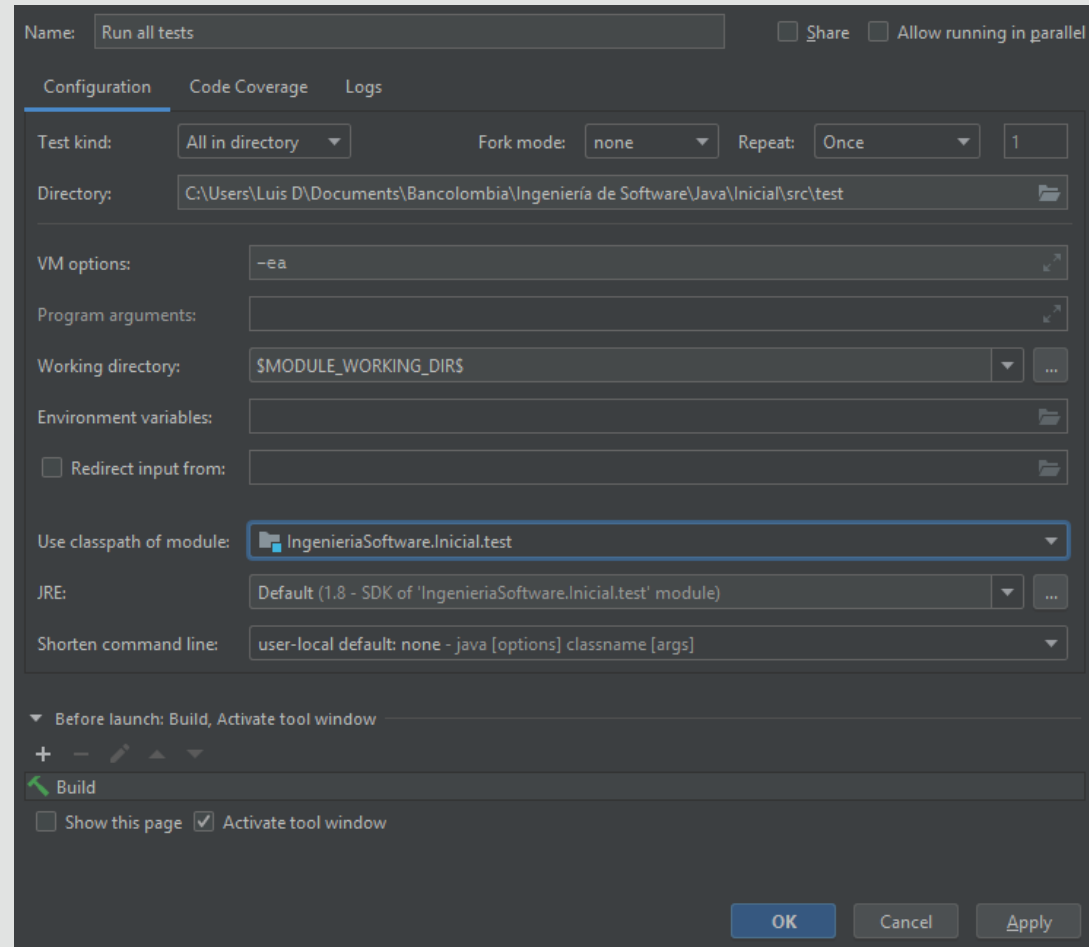
En el formulario:

Name: **Run all tests**

Test kind: **all in directory**

Directory: carpeta **test** del directorio del Proyecto.

Classpath of module:  
***[nombre\_del\_Proyecto].test***



The screenshot shows the 'Run' configuration dialog in IntelliJ IDEA. The 'Name' field is set to 'Run all tests'. The 'Test kind' is 'All in directory'. The 'Directory' is 'C:\Users\Luis D\Documents\Bancolombia\Ingeniería de Software\Java\Inicial\src\test'. The 'VM options' are '-ea'. The 'Working directory' is '\$MODULE\_WORKING\_DIRS'. The 'Use classpath of module' is set to 'IngenieriaSoftware.Inicial.test'. The 'JRE' is 'Default (1.8 - SDK of \'IngenieriaSoftware.Inicial.test\' module)'. The 'Shorten command line' is 'user-local default: none - java [options] classname [args]'. The 'Before launch' section is expanded, showing 'Build' and 'Activate tool window' checked.

Name: Run all tests ☐ Share ☐ Allow running in parallel

Configuration Code Coverage Logs

Test kind: All in directory Fork mode: none Repeat: Once 1

Directory: C:\Users\Luis D\Documents\Bancolombia\Ingeniería de Software\Java\Inicial\src\test

VM options: -ea

Program arguments:

Working directory: \$MODULE\_WORKING\_DIRS

Environment variables:

☐ Redirect input from:

Use classpath of module: IngenieriaSoftware.Inicial.test

JRE: Default (1.8 - SDK of 'IngenieriaSoftware.Inicial.test' module)

Shorten command line: user-local default: none - java [options] classname [args]

▼ Before launch: Build, Activate tool window

+ - [icon] [icon] [icon]

Build

☐ Show this page ☒ Activate tool window

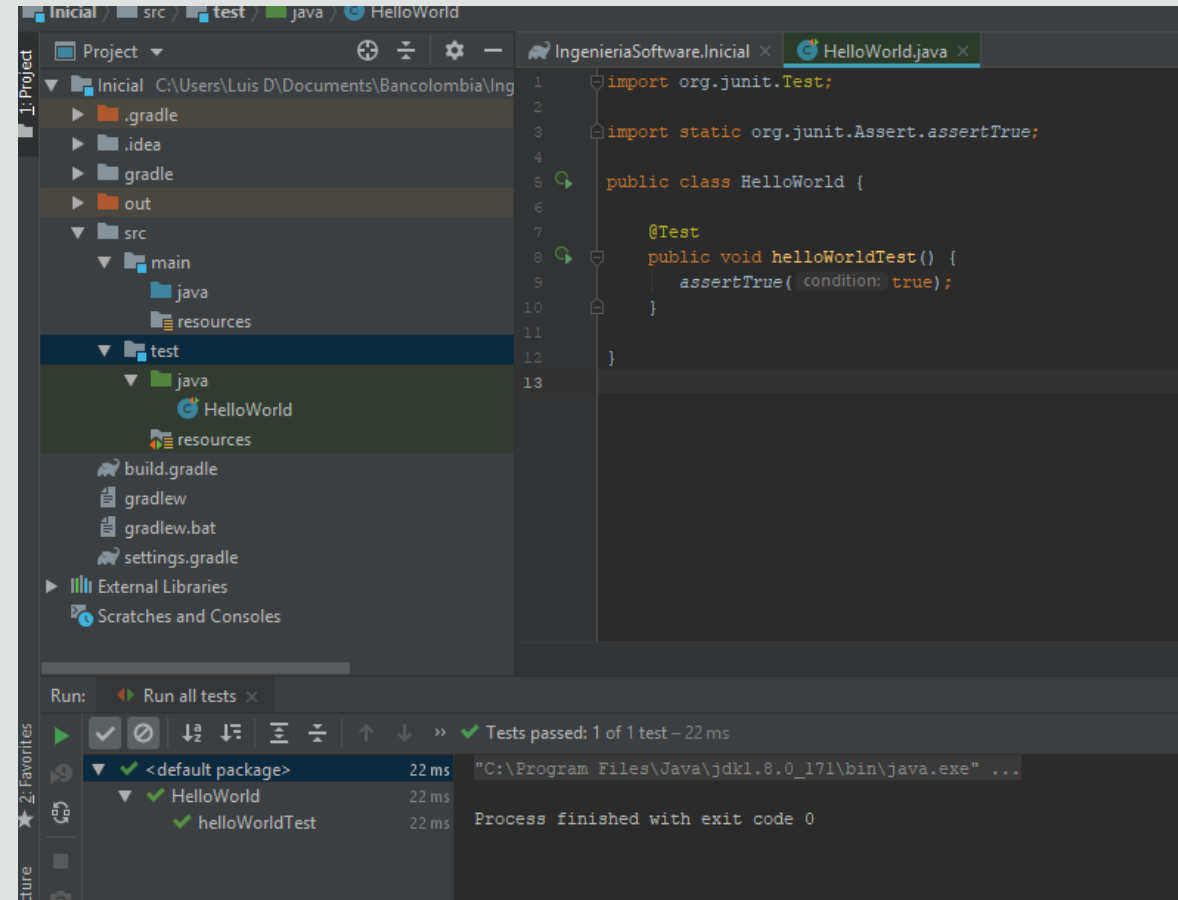
OK Cancel Apply

Es el  
momento  
de  
todos

Bancolombia

# IntelliJ IDEA

Al ejecutar esta configuración, se corren todas las pruebas en el directorio **test**.



Es el  
momento  
de  
todos



# Configurar ambientes locales para correr una clase de pruebas

Es el  
momento  
de  
todos

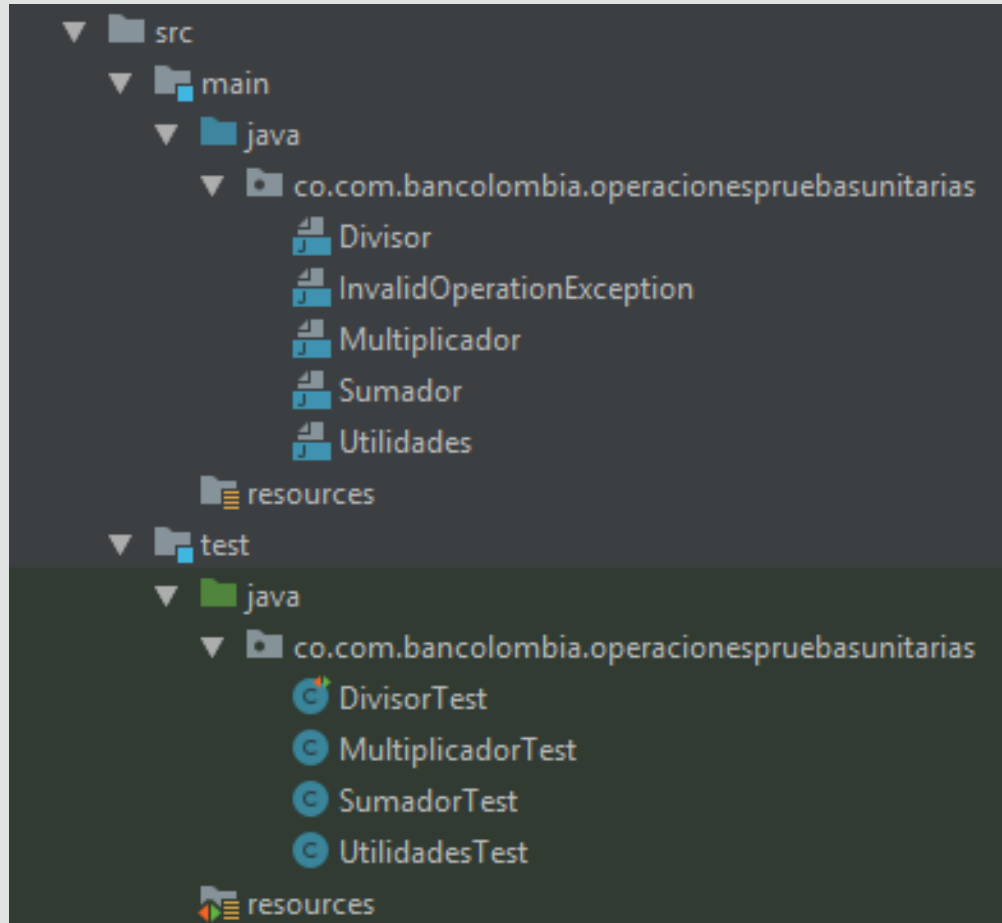
**Bancolombia** 

# Casos básicos

Es el  
momento  
de  
todos

**Bancolombia** 

# Casos de prueba



Un buen comienzo es hacer una estructura de carpetas análoga a la del proyecto, y para cada clase del proyecto, tener al menos una clase con sus casos de pruebas unitarias.

La clase de pruebas unitarias tendría el mismo package que la clase a probar, pero estaría en diferente directorio:

`src/main/java/co/com/.../proyecto/MiClase.java`

`src/test/java/co/com/.../proyecto/MiClaseTest.java`

Es el  
momento  
de  
todos

**Bancolombia**

# Casos de prueba

- En las clases de pruebas, cada caso de prueba es un método con la anotación @Test de JUnit.
- Un caso de prueba se caracteriza por tener una **entrada conocida** y un **resultado esperado**.
- Lo ideal es que en una prueba unitaria se pruebe **el resultado de una sola operación**.

# Casos de prueba

```
package co.com.bancolombia.operacionespruebasunitarias;

/**
 * La clase Sumador permite hacer operaciones de suma.
 */
public class Sumador {

    /**
     * Retorna la suma de los parámetros. Los números
     * especificados pueden ser reales.
     * @param a Sumando
     * @param b Sumando
     * @return El resultado de la suma de a y b
     */
    public double sumar(double a, double b) {
        return a + b;
    }
}
```

```
package co.com.bancolombia.operacionespruebasunitarias;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class SumadorTest {

    @Test
    public void sumarDosNumeros() {
        Sumador sumador = new Sumador();
        double param1 = 4;
        double param2 = 2.5;
        double resultadoEsperado = 6.5;

        double resultado = sumador.sumar(param1, param2);

        assertEquals(resultado, resultadoEsperado, 0D);
    }
}
```

## Casos de prueba

Se usó otro elemento de JUnit, *assertEquals*. En una aserción (*assert*) se verifica que se cumpla un criterio.

`assertEquals` verifica que los parámetros pasados sean iguales.



# AAA (*Arrange, Act, Assert*)

Es un patrón para escribir pruebas unitarias.

Consiste en que cada método de prueba unitaria tiene tres secciones:

- *Arrange* (preparar): Inicializar objetos y preparar los valores de los parámetros que se van a pasar al método que se quiere probar.
- *Act* (actuar): Invocar al método que se quiere probar con los parámetros preparados.
- *Assert* (asegurar): Verificar que el método se comporte de la manera esperada.

# AAA (Arrange, Act, Assert)

```
package co.com.bancolombia.operacionespruebasunitarias;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class SumadorTest {

    @Test
    public void sumarDosNumeros() {
        // Arrange
        Sumador sumador = new Sumador();
        double param1 = 4;
        double param2 = 2.5;
        double resultadoEsperado = 6.5;

        // Act
        double resultado = sumador.sumar(param1, param2);

        // Assert
        assertEquals(resultado, resultadoEsperado, 0D);
    }
}
```

# Práctica

- Implementar este método en *Sumador*:

```
/**
 * Retorna la suma de los números del arreglo
 * especificado. Los números especificados pueden ser
 * reales.
 * @param a Arreglo a sumar
 * @return El resultado de la suma de todos los elementos
 * de a.
 */
public double sumar(double a[])
```

- Agregar la clase *Multiplicador* e implementar:

```
/**
 * Retorna el resultado de multiplicar dos números reales.
 * @param a Primer factor
 * @param b Segundo factor
 * @return El resultado de multiplicar el primer factor por
 * el segundo.
 */
public double multiplicar(double a, double b)
```

- Agregar la clase *Divisor* e implementar:

```
/**
 * Retorna la división del primer parámetro entre el segundo.
 * Los números especificados pueden ser reales.
 * @param a Divisor
 * @param b Dividendo
 * @return El resultado de la división de a entre b
 */
public double dividir(double a, double b)
```

Implementar pruebas unitarias para los tres nuevos métodos y correrlas con éxito.

# Condicionales

Es el  
momento  
de  
todos

**Bancolombia** 

# Condicionales y ramificaciones

Los condicionales y ramificaciones determinan el resultado de los métodos.

Se debe hacer pruebas unitarias para los **posibles resultados** dados por los diferentes caminos que pueda tomar la ejecución de un método.

# Condicionales y ramificaciones

```
package co.com.bancolombia.operacionespruebasunitarias;
```

```
/**  
 * Clase con verificaciones y utilidades  
 */
```

```
public class Utilidades {
```

```
/**  
 * Revisa si un entero es primo  
 * @param n Entero a verificar  
 * @return resultado de la verificación  
 */
```

```
public boolean esPrimo(int n) {  
    if(n <= 1) {  
        return false;  
    }  
    if(n % 2 == 0 && n > 2) {  
        return false;  
    }  
    for(int i = 3; i*i <= n; i += 2) {  
        if(n % i == 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

```
}
```

UtilidadesTest.java

☐ Inherited members (Ctrl+F12) ☐ Anonymous class

UtilidadesTest

- esPrimoFalseImparGrande(): void
- esPrimoFalseNegativo(): void
- esPrimoFalsePar(): void
- esPrimoTrueDos(): void
- esPrimoTrueNumeroGrande(): void

Es el  
momento  
de  
todos

Bancolombia

# Aserciones básicas

Es el  
momento  
de  
todos

**Bancolombia** 

# Aserciones

Una aserción, por definición, es una afirmación en la que se dice que **un predicado específico siempre es verdadero**.

Aplicado en pruebas unitarias, las aserciones son funciones normalmente provistas por el framework de pruebas cuyo propósito es evaluar el predicado y en caso de que este no se cumpla, dar información sobre la razón del fallo.

Las pruebas unitarias se deben escribir de manera que si una aserción falla, esto indique que la unidad probada no se está comportando como se espera.

La cantidad de aserciones en una prueba unitaria corresponde con la cantidad de resultados que se quieran validar. Lo importante es que una prueba unitaria corresponda a una sola **acción**.



# Aserciones

JUnit provee aserciones para casos comunes:

*assertEquals*

*assertArrayEqual*

*assertTrue*

*assertFalse*

*assertNull*

*assertNotNull*

*assertSame*

*assertNotSame*

*fail*

Es el  
momento  
de  
todos

**Bancolombia** 

# Aserciones

## assertEquals

Sirve para verificar que el valor esperado es igual al valor obtenido.

```
@Test
public void asercionIgualdad() {
    String valorEsperado = "cualquier valor";
    String valorObtenido = "cualquier valor";
    assertEquals(valorEsperado, valorObtenido);
}
```

## assertFalse

Sirve para verificar que el valor esperado es igual a false.

```
@Test
public void asercionFalse() {
    boolean valorObtenido = false;
    assertFalse(valorObtenido);
}
```

## assertTrue

Sirve para verificar que el valor esperado es igual a true.

```
@Test
public void asercionTrue() {
    boolean valorObtenido = true;
    assertTrue(valorObtenido);
}
```

# Práctica

- Agregar la clase *Utilidades* e implementar:

```
/**
 * Revisa si un entero es primo
 * @param n Entero a verificar
 * @return resultado de la verificación
 */
public boolean esPrimo(int n)
```

```
/**
 * Retorna si un número es negativo
 * @param n número a verificar
 * @return resultado de la verificación
 */
public boolean esNegativo(double n)
```

```
/**
 * Retorna si un número es positivo
 * @param n número a verificar
 * @return resultado de la verificación
 */
public boolean esPositivo(double n)
```

```
public boolean esPrimo(int n) {
    if(n <= 1) {
        return false;
    }
    if(n % 2 == 0 && n > 2) {
        return false;
    }
    for(int i = 3; i*i <= n; i += 2) {
        if(n % i == 0) {
            return false;
        }
    }
    return true;
}
```

Implementar pruebas unitarias para los tres nuevos métodos y correrlas con éxito.

# Excepciones

Es el  
momento  
de  
todos

**Bancolombia** 

# Excepciones

Es muy común encontrar el lanzamiento de una excepción específica entre los comportamientos posibles de un método.

Para estos casos, se plantea usar un bloque try/catch y hacer aserciones sobre la excepción atrapada. En caso de no producirse excepción, usar la aserción fail, que hace que la prueba falle con el mensaje especificado.

```
@Test
public void excepcionArreglosDiferenteLongitudTryCatch() {
    Multiplicador multiplicador = new Multiplicador();
    double param1[] = {1.2, 3, 10.1};
    double param2[] = {2, 3, 4, 5};
    String mensajeEsperado = "Los arreglos no tienen la misma cantidad de elementos";

    try {
        multiplicador.multiplicar(param1, param2);
        fail("Excepción InvalidOperationException no lanzada");
    } catch (InvalidOperationException e) {
        assertEquals(mensajeEsperado, e.getMessage());
    }
}
```

# Práctica

- Agregar la clase *InvalidOperationException*:

⚡	InvalidOperationException	
f	operation	String
m	InvalidOperationException(String, String)	
m	getOperation()	String
m	setOperation(String)	void

Implementar prueba unitaria para el método *dividir* con esta excepción.

- Lanzar una *InvalidOperationException* en el método *dividir* de la clase Divisor:

```
public double dividir(double a, double b) throws InvalidOperationException{
    if(b == 0) {
        throw new InvalidOperationException("División", "El divisor es cero");
    }
    return a / b;
}
```

# Anotaciones

Es el  
momento  
de  
todos

**Bancolombia** 

# Anotaciones

JUnit provee algunas funcionalidades útiles a través de anotaciones:

***@Before***

***@BeforeClass***

***@After***

***@AfterClass***

***@Test***

***@Ignore***

***@Test(timeout=500)***

***@Test(expected=IllegalArgumentException.class)***

***@Rule***

Es el  
momento  
de  
todos

**Bancolombia** 



# Anotaciones

## *@Before*

Los métodos anotados con *@Before* se ejecutan antes de cada caso de prueba, es útil cuando se tiene código común antes de todos los casos de una clase.

## *@After*

Los métodos anotados con *@After* se ejecutan después de cada caso de prueba, es útil para limpiar después de la ejecución de cada uno de los casos.

```
import org.junit.Test;

import static org.junit.Assert.assertTrue;

public class HelloWorld {

    // ...

    private List<String> lista;

    @Before
    public void init() {
        lista = new ArrayList<>(
            Arrays.asList("test1", "test2")
        );
    }

    @After
    public void finalize() {
        lista.clear();
    }

}
```

# Anotaciones

## *@BeforeClass*

Los métodos anotados con `@BeforeClass` se ejecutan antes de todos los casos de prueba de una clase. Deben ser estáticos.

## *@AfterClass*

Los métodos anotados con `@AfterClass` se ejecutan después de todos los casos de prueba de una clase. Deben ser estáticos.

```
import org.junit.Test;

public class BeforeClassAndAfterClassAnnotationsTest {

    // ...

    @BeforeClass
    public static void setup() {
        LOG.info("Antes de todos");
    }

    @AfterClass
    public static void tearDown() {
        LOG.info("Después de todos");
    }
}
```

# Anotaciones

Uno de los usos de `@Rule` es probar excepciones de una manera más elegante:

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void excepcionArreglosDiferenteLongitud() throws InvalidOperationException{
    // Arrange
    Multiplicador multiplicador = new Multiplicador();
    double param1[] = {1.2, 3, 10.1};
    double param2[] = {2, 3, 4, 5};
    String mensajeEsperado = "Los arreglos no tienen la misma cantidad de elementos";

    // Assert
    thrown.expect(InvalidOperationException.class);
    thrown.expectMessage(mensajeEsperado);

    // Act
    multiplicador.multiplicar(param1, param2);
}
```

# GRACIAS!



INGENIERÍA DE SOFTWARE

## Calificación

