



Diseño de Software

Pruebas Unitarias

Grupo #5

Repositorio git:

<https://github.com/JerePoveda/grupo-5-Pruebas-Unitarias.git>

Integrantes:

- Shirley Yamel Aragón Intriago (saragon@espol.edu.ec)
- Iván Andrés Salinas Castillo (ivansali@espol.edu.ec)
- Jeremias Davide Poveda Guerra (jdpoveda@espol.edu.ec)
- Eloy Ranses Rojas San Andres (erojas@espol.edu.ec)

PAO I - 2025

| | |
|-------------------------------------|----------|
| Sección A | 4 |
| Implementación de la lógica | 4 |
| Casos de prueba (Lógica inicial) | 4 |
| Errores de lógica y Corrección | 6 |
| Casos de pruebas (Lógica corregida) | 7 |
| Sección B | 9 |
| Método #1 Cs() | 9 |
| Método #2 CalculateYearBonus() | 13 |

Sección A

Implementación de la lógica

La lógica principal ya está explicada en el archivo del taller; esta permite que el primer caso de prueba se ejecute con éxito, aunque el segundo no lo haga. No obstante, dicha lógica contiene errores que se evidencian más adelante y que impiden el correcto funcionamiento de la función. A continuación, se presentan los casos de prueba realizados.

```
1 import org.junit.Test;
2 import static org.junit.Assert.assertEquals;
3
4 public class Prueba {
5     @Test
6     public void testFindMax() {
7         assertEquals(10000, Calculation.findMax(new int[] {10000,0,1,-9687,100}));
8     }
9 }
```

```
1
2 public class Calculation {
3     public static int findMax(int arr[]) {
4         int max=0;
5         for(int i=1;i<arr.length;i++) {
6             if(max<arr[i])
7                 max=arr[i];
8         }
9         return max;
10    }
11 }
```

Casos de prueba (Lógica inicial)

| Caso #1 | |
|------------------------|--|
| ID del caso | TC001 |
| Entrada | [-3,-2,-1,0,1,2,3] |
| Salida esperada | 3 |
| Descripción | Se utiliza un arreglo que incluye números positivos, negativos y el valor cero. Además, dicho arreglo está ordenado de forma ascendente, por lo que se analiza el comportamiento de la función al trabajar con una lista previamente ordenada. |
| Resultado de la prueba | El resultado fue exitoso, se obtuvo el valor esperado. |

Finished after 0,114 seconds

Runs: 1/1 Errors: 0 Failures: 0

Prueba [Runner: JUnit 4] (0,000 s)

testFindMax (0,000 s)

```
1 import org.junit.Test;
2 import static org.junit.Assert.assertEquals;
3
4 public class Prueba {
5     @Test
6     public void testFindMax() {
7         assertEquals(3, Calculation.findMax(new int[] {-3,-2,-1,0,1,2,3}));
8     }
9 }
```

| Caso #2 | |
|------------------------|--|
| ID del caso | TC002 |
| Entrada | [-45,87,-100,0,-97,-200,50] |
| Salida esperada | 87 |
| Descripción | Se emplea un arreglo que contiene números negativos, positivos y el cero, dispuesto sin ningún orden específico. En este caso, se evalúa el comportamiento de la función frente a un arreglo de tipo convencional. |
| Resultado de la prueba | El resultado fue exitoso, se obtuvo el valor esperado. |

```

1 import org.junit.Test;
2 import static org.junit.Assert.assertEquals;
3
4 public class Prueba {
5     @Test
6     public void testFindMax() {
7         assertEquals(87, Calculation.findMax(new int[] {-45,87,-100,0,-97,-200,50}));
8     }
9 }

```

| Caso #3 | |
|------------------------|--|
| ID de entrada | TC003 |
| Entrada | [20,-10000,-8878,-9687,-1500] |
| Salida esperada | 20 |
| Descripción | Se utiliza un arreglo que incluye números positivos, negativos y el valor cero. Además, dicho arreglo está ordenado de forma ascendente, por lo que se analiza el comportamiento de la función al trabajar con una lista previamente ordenada. |
| Resultado de la prueba | Se utiliza un arreglo en el que el valor máximo se ubica al inicio. Este tipo de disposición tiende a evidenciar los errores presentes en la lógica del método. |

```

1 import org.junit.Test;
2 import static org.junit.Assert.assertEquals;
3
4 public class Prueba {
5     @Test
6     public void testFindMax() {
7         assertEquals(20, Calculation.findMax(new int[] {20,-10000,-8878,-9687,-1500}));
8     }
9 }

```

Failure Trace

```

java.lang.AssertionError: expected:<20> but was:<0>
    at org.junit.Assert.fail(Assert.java:89)
    at Prueba.testFindMax(Prueba.java:7)

```

```

1 import org.junit.Test;
2 import static org.junit.Assert.assertEquals;
3
4 public class Prueba {
5     @Test
6     public void testFindMax() {
7         assertEquals(20, Calculation.findMax(new int[] {20,-10000,-8878,-9687,-1500}));
8     }
9 }
10

```

Errores de lógica y Corrección

Error 1

El primer problema de la función es que el bucle for arranca con un índice incorrecto. Al empezar en la posición 1, omite el primer elemento de cada arreglo, lo que puede provocar fallos en la ejecución.

Solución: iniciar la variable `i` en 0 en lugar de 1.

Error 2

Otro problema está en la inicialización de la variable max, que comienza en 0. Si el arreglo incluye valores negativos, esta elección introduce inconsistencias lógicas y puede producir resultados incorrectos, sobre todo al manejar números negativos.

Solución: inicializar max con un elemento existente del arreglo; en este caso, usar el primer elemento.

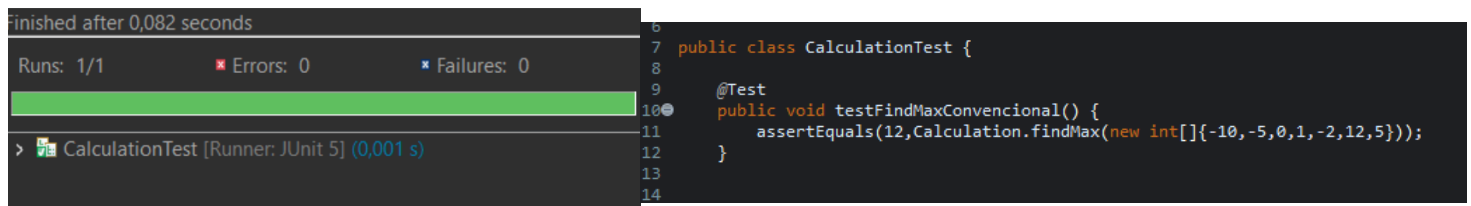
```

2
3 public class Calculation {
4
5     public static int findMax(int[] arr) {
6         int max=arr[0];
7         for(int i=0;i<arr.length;i++) {
8             if(max<arr[i])
9                 max=arr[i];
10        }
11        return max;
12    }
13
14
15 }
16

```

Casos de pruebas (Lógica corregida)

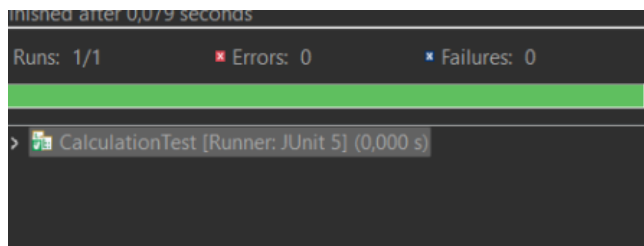
| Caso #4 | |
|------------------------|--|
| ID del caso | TC004 |
| Entrada | [-10,-5,0,1,-2,12,5] |
| Salida esperada | 12 |
| Descripción | Se utiliza un arreglo convencional con el fin de verificar que la lógica continúe operando correctamente, tal como sucedía antes de aplicar la corrección. |
| Resultado de la prueba | El resultado fue exitoso, se obtuvo el valor esperado. |



```
finished after 0,082 seconds
Runs: 1/1      Errors: 0      Failures: 0
> CalculationTest [Runner: JUnit 5] (0,001 s)

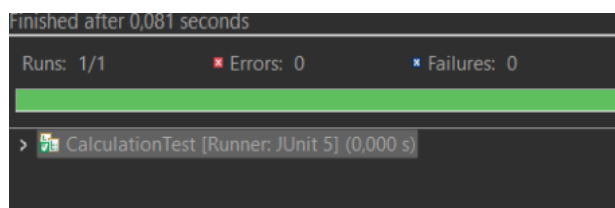
6
7 public class CalculationTest {
8
9     @Test
10    public void testFindMaxConvencional() {
11        assertEquals(12, Calculation.findMax(new int[]{-10, -5, 0, 1, -2, 12, 5}));
12    }
13
14 }
```

| Caso #5 | |
|------------------------|--|
| ID del caso | TC006 |
| Entrada | [-10,-5,0,3,7] [7,3,0,-5,-10] |
| Salida esperada | Para ambos 7 |
| Descripción | Se evalúan dos arreglos ordenados: uno en forma ascendente y otro en forma descendente. Con ello, se contemplan tanto el escenario en el que el valor máximo aparece al final del arreglo como aquel en el que se ubica al inicio. Este último corresponde al caso #3, en el cual se detectó un fallo. |
| Resultado de la prueba | El resultado fue exitoso, se obtuvo el valor esperado para ambos arreglos. |



```
15 @Test
16 public void testFindMaxOrdenados() {
17     assertEquals(7, Calculation.findMax(new int[]{-10, -5, 0, 3, 7}));
18     assertEquals(7, Calculation.findMax(new int[]{7, 3, 0, -5, -10}));
19 }
20
```

| Caso #6 | |
|------------------------|--|
| ID del caso | TC006 |
| Entrada | [9,9,9,9,9,9] |
| Salida esperada | 9 |
| Descripción | Se utiliza un arreglo en el que todos los elementos son idénticos. Con esta prueba se busca confirmar que la lógica del método funcione adecuadamente incluso en presencia de valores repetidos. |
| Resultado de la prueba | El resultado fue exitoso, se obtuvo el valor esperado. |



```
@Test
public void testFindMaxRepetido() {
    assertEquals(9, Calculation.findMax(new int[]{9, 9, 9, 9, 9}));
}
```


Sección B

En el programa propuesto hay dos métodos con la capacidad de ser probados

1. Cs()
2. CalculateYearBonus()

A continuación, sus pruebas:

Método #1 Cs()

Primero creamos 6 instancias de Employee para luego hacer uso de la anotación `@BeforeEach` para darle valor a los atributos de estas instancias, y crear un método que determine si el mes es par o impar; lo cual nos permitió crear estos 5 casos de prueba:

```
class CsTest {  
  
    private Employee Trabajador1;  
    private Employee Trabajador2;  
    private Employee Trabajador3;  
    private Employee Manager1;  
    private Employee Supervisor1;  
    private Employee Supervisor2;  
  
    @BeforeEach  
    void inicializarEmpleados() {  
        Trabajador1 = new Employee(1200, "USD", 5, EmployeeType.Worker);  
        Trabajador2 = new Employee(1000, "EUR", 7, EmployeeType.Worker);  
        Trabajador3 = new Employee(-500, "USD", 10, EmployeeType.Worker);  
        Manager1 = new Employee(2000, "USD", 15, EmployeeType.Manager);  
        Supervisor1 = new Employee(1500, "USD", 10, EmployeeType.Supervisor);  
        Supervisor2 = new Employee(1800, "CAD", 12, EmployeeType.Supervisor);  
    }  
  
    private boolean esMesImpar() {  
        // Determina si el mes actual es impar  
        int mesActual = LocalDate.now().getMonthValue();  
        return mesActual % 2 != 0;  
    }  
}
```

| Caso #1 | |
|----------------------------|--|
| ID del caso | TC001 |
| Nombre del caso | salarioMesUSDTest() |
| Entrada | [Trabajador.1cs() Supervisor1.cs() Manager1.cs()] |
| Salida esperada (mes par) | 1200.0f (Trabajador1) 1503.5f (Supervisor1) 2010.5f (Manager1) |
| Salida esperada(mes impar) | 1264.33f (Trabajador1) 1568.33f (Supervisor1) 2100.83f (Manager1) |
| Descripción | Se evalúan empleados con moneda USD en meses impares, calculando salario, décimo y bono según el tipo de empleado. |
| Resultado de la prueba | El resultado fue exitoso, y sus cálculos correctos |

```

@Test
void salarioMesUSDTest() {
    if (esMesImpar()) {
        // Mes impar
        float salarioEsperadoTrabajador1 = 386.0f / 6 + 1200;
        float salarioEsperadoSupervisor1 = 386.0f / 6 + 1500 + (10 * 0.35f);
        float salarioEsperadoManager1 = 386.0f / 6 + 2000 + (15 * 0.7f);

        assertEquals(salarioEsperadoTrabajador1, Trabajador1.cs(), "Salario mes impar trabajador 1 incorrecto");
        assertEquals(salarioEsperadoSupervisor1, Supervisor1.cs(), "Salario mes impar supervisor 1 incorrecto");
        assertEquals(salarioEsperadoManager1, Manager1.cs(), "Salario mes impar manager 1 incorrecto");
    } else {
        // Mes par
        assertEquals(1200, Trabajador1.cs(), "Salario mes par trabajador 1 incorrecto");
        assertEquals(1500 + (10 * 0.35f), Supervisor1.cs(), "Salario mes par supervisor 1 incorrecto");
        assertEquals(2000 + (15 * 0.7f), Manager1.cs(), "Salario mes par manager 1 incorrecto");
    }
}

```

| Caso #2 | |
|----------------------------|--|
| ID del caso | TC002 |
| Nombre del caso | salarioMesNoUSDTest() |
| Entrada | Trabajador.2cs() Supervisor2.cs() |
| Salida esperada (mes par) | 950.0f (Trabajador2) 1761.5f (Supervisor2) |
| Salida esperada(mes impar) | 1047.33f (Trabajador2) 1791.33f (Supervisor2) |
| Descripción | Se evalúan empleados con moneda distinta a USD en meses impares, considerando conversión monetaria, salario, décimo y bono según el tipo de empleado |
| Resultado de la prueba | El resultado fue exitoso, y sus cálculos correctos |

```

@Test
void salarioMesNoUSDTest() {
    if (esMesImpar()) {
        // Mes impar
        float salarioEsperadoTrabajador2 = (1000 * 0.95f) + (386.0f / 6);
        float salarioEsperadoSupervisor2 = (1800 * 0.95f) + (12 * 0.35f) + (386.0f / 6);

        assertEquals(salarioEsperadoTrabajador2, Trabajador2.cs(), "Salario mes impar trabajador 2 incorrecto");
        assertEquals(salarioEsperadoSupervisor2, Supervisor2.cs(), "Salario mes impar supervisor 2 incorrecto");
    } else {
        // Mes par
        float salarioEsperadoTrabajador2 = 1000 * 0.95f;
        float salarioEsperadoSupervisor2 = (1800 * 0.95f) + (12 * 0.35f);

        assertEquals(salarioEsperadoTrabajador2, Trabajador2.cs(), "Salario mes par trabajador 2 incorrecto");
        assertEquals(salarioEsperadoSupervisor2, Supervisor2.cs(), "Salario mes par supervisor 2 incorrecto");
    }
}

```

| Caso #3 | |
|------------------------|--|
| ID del caso | TC003 |
| Nombre del caso | salarioConSueldoNegativo() |
| Entrada | Trabajador3.cs() |
| Salida esperada | 0.0f |
| Descripción | Se evalúa un empleado con salario negativo para verificar que el método retorna 0 como salario esperado. |
| Observaciones | El test ha sido desactivado debido a situación improbable. |
| Resultado de la prueba | El test fue omitido de testeo. |

```


@Test
@Disabled
void salarioConSueldoNegativoTest() {
    assertEquals(0.0f, Trabajador3.cs(), "El salario debería ser 0 para sueldos negativos");
}




```

Aquí se prueba el éxito de los 3 casos propuestos:

Finished after 0,071 seconds

Runs: 3/3 (1 skipped) ❌ Errors: 0 ❏ Failures: 0

▼  CsTest [Runner: JUnit 5] (0,016 s)

-  salarioMesNoUSDTest() (0,012 s)
-  salarioMesUSDTest() (0,001 s)
-  salarioConSueldoNegativoTest() (0,000 s)

Método #2 CalculateYearBonus()

Primero se realizó la creación de 6 elementos. Para esto se crearon los objetos fuera de un método. Sin embargo, para hacer uso de otras anotaciones se usa `@BeforeEach` para su llenado. Cabe recalcar que, aunque hubiera sido mas acertado usar `@BeforeAll`, su uso involucraba otros elementos que no cubren el objetivo del taller así que se optó por este

```
class CalculateYearBonusTest {
    Employee Trabajador1, Trabajador2, Trabajador3, Manager1, Supervisor1, Supervisor2;

    @BeforeEach
    void llenarEmpleados() {
        Trabajador1 = new Employee(1000, "USD", 5, EmployeeType.Worker);
        Trabajador2 = new Employee(980, "EUR", 5, EmployeeType.Worker);
        Trabajador3 = new Employee(-386.0f, "USD", 5, EmployeeType.Worker);
        Manager1 = new Employee(3600, "USD", 15, EmployeeType.Manager);
        Supervisor1 = new Employee(1950, "USD", 8, EmployeeType.Supervisor);
        Supervisor2 = new Employee(2000, "CAD", 8, EmployeeType.Supervisor);
    }
}
```

Tras eso se crearon 3 casos de prueba.

| Caso #1 | |
|------------------------|--|
| ID del caso | TC001 |
| Nombre del caso | NormalWorkersTest() |
| Entrada | Trabajador1.CalculateYearBonus() Supervisor1.CalculateYearBonus() Manager1.CalculateYearBonus() |
| Salida esperada | 386.0f 3986.0f 2143f |
| Descripción | Se evalúan a empleados normales según la lógica del algoritmo. Aquí también se evalúa el switch para los tipos de empleados. |
| Resultado de la prueba | El resultado fue exitoso, y sus cálculos correctos. |

```
@Test
void NormalWorkersTest() {
    assertEquals(386.0f, Trabajador1.CalculateYearBonus(), "Empleado comun");
    assertEquals(3986.0f, Manager1.CalculateYearBonus(), "Manager");
    assertEquals(2143f, Supervisor1.CalculateYearBonus(), "Supervisor normal");
}
```

| Caso #2 | |
|------------------------|--|
| ID del caso | TC002 |
| Nombre del caso | NoUSDTest() |
| Entrada | Trabajador2.CalculateYearBonus() Supervisor2.CalculateYearBonus() |
| Salida esperada | 386.0f 2093f |
| Descripción | Se evalúan a empleados que no son pagados en dólares, ya que ellos cuentan con otra declaración de variables |
| Resultado de la prueba | El resultado fue exitoso, y sus cálculos correctos. |

```

@Test
void NoUSDTest() {
    assertEquals(386.0f, Trabajador2.CalculateYearBonus(), "Empleado pagado en Euros");
    assertEquals(2093f, Supervisor2.CalculateYearBonus(), "Supervisor pagado en dolares canadienses");
}

```

| Caso #3 | |
|------------------------|--|
| ID del caso | TC003 |
| Nombre del caso | EmpleadoSueldoNegativoTest () |
| Entrada | Trabajador3.CalculateYearBonus() |
| Salida esperada | 386.0f |
| Descripción | Se evalúan a un empleado con sueldo negativo. |
| Observaciones | El test ha sido desactivado debido a situación improbable. |
| Resultado de la prueba | El test fue omitido de testeo. |

```

@Test
@Disabled
void salarioConSueldoNegativoTest() {
    assertEquals(0.0f, Trabajador3.cs(), "El salario debería ser 0 para sueldos negativos");
}

```

Aquí se prueba el éxito de los 3 casos propuestos

