Práctica 2 de JUnit

Ejercicio 1: Realiza con Eclipse el resto de pruebas y comenta los resultados.

Para realizar las pruebas de calcularSalarioBruto es necesario añadir el control de excepciones a dicha función en la clase EmpleadoBR.

```
public float calculaSalarioBruto(String tipo, float ventasMes, float horasExtra) throws Exception {
    float base = 0;
    float salarioBruto;
    float extra;

    // Exception handling
    if (tipo == null)
    {
        throw new Exception("El tipo de empleado no puede ser null.");
    }
    if (ventasMes < 0)
    {
        throw new Exception("Las ventas del mes no pueden ser negativas.");
    }
    if (horasExtra < 0)
    {
        throw new Exception("Las horas extra no pueden ser negativas.");
    }
```

Además, se deberá añadir “throws Exception” al final de la declaración de la función. Al añadirlo en EmpleadoBR, también deberá añadirse a todas las funciones “testCalculaSalarioBrutoX” esa misma línea de “throws Exception”, para evitar errores.

```
import static org.junit.Assert.*;

public class EmpleadoBRTest {

    @Test
    public void testCalculaSalarioBruto1() throws Exception {
        EmpleadoBR empleado = new EmpleadoBR();
        float resultadoReal = empleado.calculaSalarioBruto("vendedor", 2000.0f, 8.0f);
        float resultadoEsperado = 1360.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }

    @Test
    public void testCalculaSalarioBruto2() throws Exception {
        EmpleadoBR empleado = new EmpleadoBR();
        float resultadoReal = empleado.calculaSalarioBruto("vendedor", 1500.0f, 3.0f);
        float resultadoEsperado = 1260.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }

    @Test
    public void testCalculaSalarioBruto3() throws Exception {
        EmpleadoBR empleado = new EmpleadoBR();
        float resultadoReal = empleado.calculaSalarioBruto("vendedor", 1499.99f, 0.0f);
        float resultadoEsperado = 1100.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }

    @Test
    public void testCalculaSalarioBruto4() throws Exception {
        EmpleadoBR empleado = new EmpleadoBR();
        float resultadoReal = empleado.calculaSalarioBruto("encargado", 1250.0f, 8.0f);
        float resultadoEsperado = 1760.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }

    @Test
    public void testCalculaSalarioBruto5() throws Exception {
        EmpleadoBR empleado = new EmpleadoBR();
        float resultadoReal = empleado.calculaSalarioBruto("encargado", 1000.0f, 0.0f);
        float resultadoEsperado = 1600.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }

    @Test
    public void testCalculaSalarioBruto6() throws Exception {
        EmpleadoBR empleado = new EmpleadoBR();
        float resultadoReal = empleado.calculaSalarioBruto("encargado", 999.99f, 3.0f);
        float resultadoEsperado = 1560.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
```

Todo esto es para poder usar “@Test(expected=Exception.class)” en los tests de calculaSalarioBruto, que se necesitará usar en los últimos tres casos de prueba de testCalcularSalarioBrutoX.

```java
@Test
public void testCalculaSalarioBruto7() throws Exception {
    EmpleadoBR empleado = new EmpleadoBR();
    float resultadoReal = empleado.calculaSalarioBruto("encargado", 500.0f, 0.0f);
    float resultadoEsperado = 1500.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}

@Test
public void testCalculaSalarioBruto8() throws Exception {
    EmpleadoBR empleado = new EmpleadoBR();
    float resultadoReal = empleado.calculaSalarioBruto("encargado", 0.0f, 8.0f);
    float resultadoEsperado = 1660.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}

@Test(expected=Exception.class)
public void testCalculaSalarioBruto9() throws Exception {
    EmpleadoBR empleado = new EmpleadoBR();
    float resultadoReal = empleado.calculaSalarioBruto("vendedor", -1.0f, 8.0f);
    float resultadoEsperado = 1260.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}

@Test(expected=Exception.class)
public void testCalculaSalarioBruto10() throws Exception {
    EmpleadoBR empleado = new EmpleadoBR();
    float resultadoReal = empleado.calculaSalarioBruto("vendedor", 1500.0f, -1.0f);
    float resultadoEsperado = 1260.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}

@Test(expected=Exception.class)
public void testCalculaSalarioBruto11() throws Exception {
    EmpleadoBR empleado = new EmpleadoBR();
    float resultadoReal = empleado.calculaSalarioBruto(null, 1500.0f, 8.0f);
    float resultadoEsperado = 1260.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}
```

En el caso de textCalcularSalarioNeto9 (es decir, el último), lanza una excepción, y para poder detectarla, hay que añadir “throws Exception” al declarar la función y además el decorativo “@Test(exception=Exception.class)”.

```java
public float calculaSalarioNeto(float salarioBruto) throws Exception {
    float salarioNeto = salarioBruto;

    if (salarioBruto < 0)
        throw new Exception("El salario no puede ser un valor negativo");
    if (salarioBruto >= 1000 & salarioBruto < 1500) {
        salarioNeto = (float) (salarioBruto * (1 - 0.16));
    }
    if (salarioBruto >= 1500) {
        salarioNeto = (float) (salarioBruto * (1 - 0.18));
    }
    return salarioNeto;
}
```

Tras realizar estos ajustes los casos de prueba se ejecutarán sin problemas.

```java
@Test
public void testCalculaSalarioNeto1() {
    EmpleadoBR empleado = new EmpleadoBR();
    try {
        float resultadoReal = empleado.calculaSalarioNeto(2000.0f);
        float resultadoEsperado = 1640.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
    catch(Exception error){
        System.out.println("El sueldo bruto no puede ser menor que 0");
    }
}

@Test
public void testCalculaSalarioNeto2() {
    EmpleadoBR empleado = new EmpleadoBR();
    try {
        float resultadoReal = empleado.calculaSalarioNeto(1500.0f);
        float resultadoEsperado = 1230.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
    catch(Exception error){
        System.out.println("El sueldo bruto no puede ser menor que 0");
    }
}

@Test
public void testCalculaSalarioNeto3() {
    EmpleadoBR empleado = new EmpleadoBR();
    try {
        float resultadoReal = empleado.calculaSalarioNeto(1499.99f);
        float resultadoEsperado = 1259.9916f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
    catch(Exception error){
        System.out.println("El sueldo bruto no puede ser menor que 0");
    }
}
```

```java
@Test
public void testCalculaSalarioNeto4() {
    EmpleadoBR empleado = new EmpleadoBR();
    try {
        float resultadoReal = empleado.calculaSalarioNeto(1250.0f);
        float resultadoEsperado = 1050.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
    catch(Exception error){
        System.out.println("El sueldo bruto no puede ser menor que 0");
    }
}

@Test
public void testCalculaSalarioNeto5() {
    EmpleadoBR empleado = new EmpleadoBR();
    try {
        float resultadoReal = empleado.calculaSalarioNeto(1000.0f);
        float resultadoEsperado = 840.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
    catch(Exception error){
        System.out.println("El sueldo bruto no puede ser menor que 0");
    }
}

@Test
public void testCalculaSalarioNeto6() {
    EmpleadoBR empleado = new EmpleadoBR();
    try {
        float resultadoReal = empleado.calculaSalarioNeto(999.99f);
        float resultadoEsperado = 999.99f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
    catch(Exception error){
        System.out.println("El sueldo bruto no puede ser menor que 0");
    }
}
```

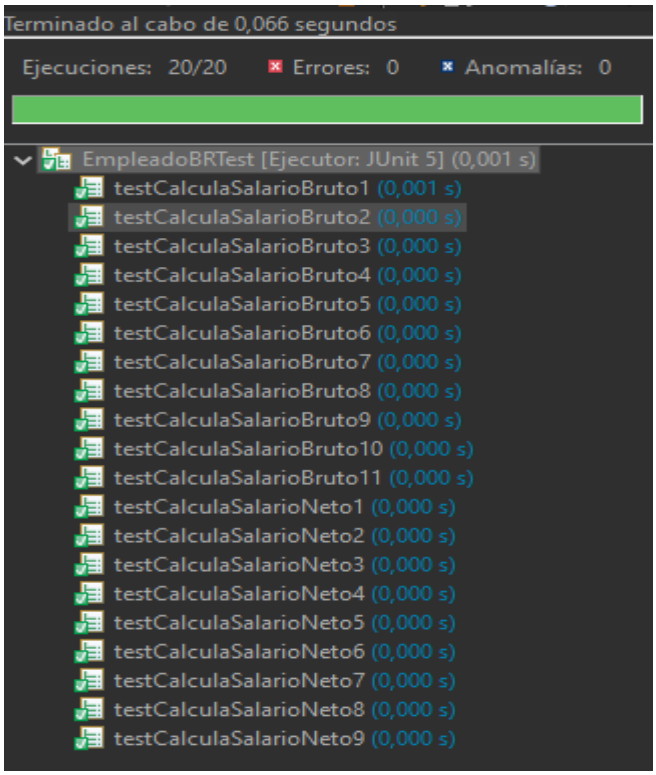
```java
@Test
public void testCalculaSalarioNeto7() {
    EmpleadoBR empleado = new EmpleadoBR();
    try {
        float resultadoReal = empleado.calculaSalarioNeto(500.0f);
        float resultadoEsperado = 500.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
    catch(Exception error){
        System.out.println("El sueldo bruto no puede ser menor que 0");
    }
}

@Test
public void testCalculaSalarioNeto8() {
    EmpleadoBR empleado = new EmpleadoBR();
    try {
        float resultadoReal = empleado.calculaSalarioNeto(0.0f);
        float resultadoEsperado = 0.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
    catch(Exception error){
        System.out.println("El sueldo bruto no puede ser menor que 0");
    }
}

@Test(expected=Exception.class)
public void testCalculaSalarioNeto9() throws Exception {
    EmpleadoBR empleado = new EmpleadoBR();
    float resultadoReal = empleado.calculaSalarioNeto(-1.0f);
    float resultadoEsperado = 0.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}
```

Test 9, que espera una excepción.

Resultados:



Ejercicio 2: Realiza los test mediante pruebas parametrizadas.

Se deberán realizar dos clases de tests parametrizados, una para CalcularSalarioBruto y otra para CalcularSalarioNeto.

En el caso del cálculo del salario bruto, se deben usar parámetros que tengan las siguientes características: una cadena para el tipo, un float para las ventas del mes, y otro float para las horas extra. Además, se debe añadir otro dato, un float, para el resultado esperado. Por desgracia, desconozco como probar las excepciones con pruebas parametrizadas, así que se omitirán en las dos clases de tests parametrizados.

EmpleadoBR: Tests parametrizados del cálculo del salario bruto:

Empezamos con: `@RunWith(Parameterized.class)`

La clase necesita un constructor:

```
public EmpleadoBRParamSalBrutoTest (String tipo, float ventasMes, float horasExtra, float expected) {
    this.tipo = tipo;
    this.ventasMes = ventasMes;
    this.horasExtra = horasExtra;
    this.expected = expected;
}
```

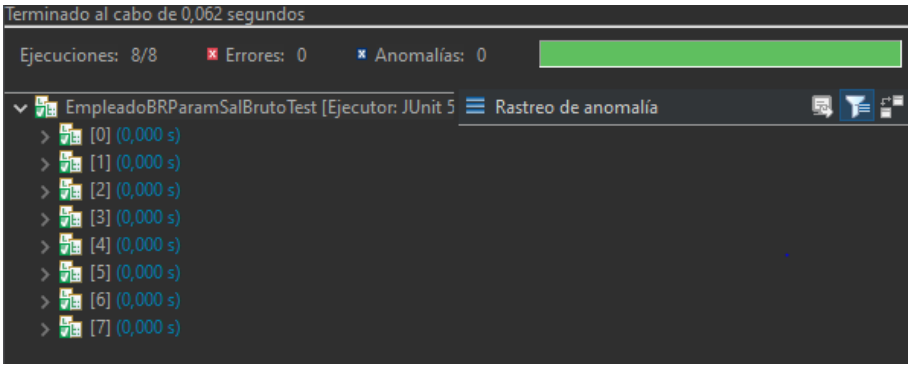
Luego, debemos inicializar el conjunto de parámetros:

```
@Parameters
public static Collection<Object[]> numeros() {
    return Arrays.asList(new Object[][] {{"vendedor", 2000.0f, 8.0f, 1360.0f},
                                         {"vendedor", 1500.0f, 3.0f, 1260.0f},
                                         {"vendedor", 1499.99f, 0.0f, 1100.0f},
                                         {"encargado", 1250.0f, 8.0f, 1760.0f},
                                         {"encargado", 1000.0f, 0.0f, 1600.0f},
                                         {"encargado", 999.99f, 3.0f, 1560.0f},
                                         {"encargado", 500.99f, 0.0f, 1500.0f},
                                         {"encargado", 0.99f, 8.0f, 1660.0f}
                                         });
}
```

Y, por último, implementamos una función genérica para comprobar los resultados y los parámetros:

```
@Test
public void testCalculaSalarioBruto() throws Exception{
    EmpleadoBR empleado = new EmpleadoBR(tipo, ventasMes, horasExtra);
    float resultado = empleado.calculaSalarioBruto(tipo, ventasMes, horasExtra);
    assertEquals(expected, resultado, 0.01);
}
```

Resultados:



En el caso del cálculo del salario neto, se deben usar parámetros que tengan las siguientes características: un float para el salario bruto, y un float para el resultado esperado. Por desgracia, desconozco como probar las excepciones con pruebas parametrizadas, así que se omitirán en las dos clases de tests parametrizados.

EmpleadoBR: Tests parametrizados del cálculo del salario bruto:

Empezamos con: `@RunWith(Parameterized.class)`

La clase necesita un constructor, como el anterior test:

```
private float salarioBruto, expected;

public EmpleadoBRParamSalNetoTest (float salarioBruto, float expected) {
    this.salarioBruto = salarioBruto;
    this.expected = expected;
}
```

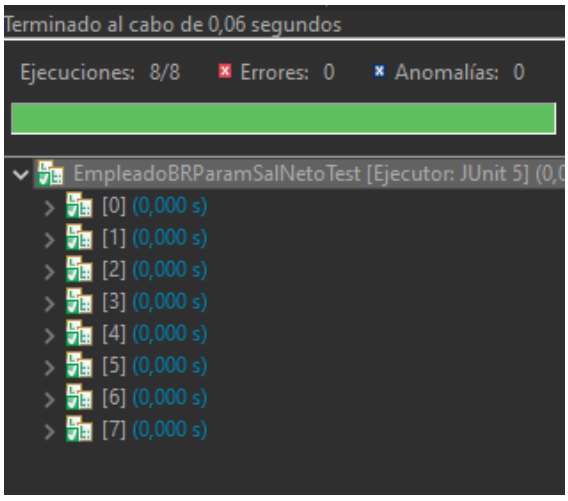
Inicialización del conjunto de parámetros:

```
@Parameters
public static Collection<Object[]> numeros() {
    return Arrays.asList(new Object[][] {{2000.0f, 1640.0f},
                                         {1500.0f,  1230.0f},
                                         {1499.99f,  1259.9916f},
                                         {1250.0f,  1050.0f},
                                         {1000.0f,  840.0f},
                                         {999.99f,  999.99f},
                                         {500.0f,  500.0f},
                                         {0.0f,  0.0f},
                                         });
}
```

Creamos una función test genérica para probar los parámetros:

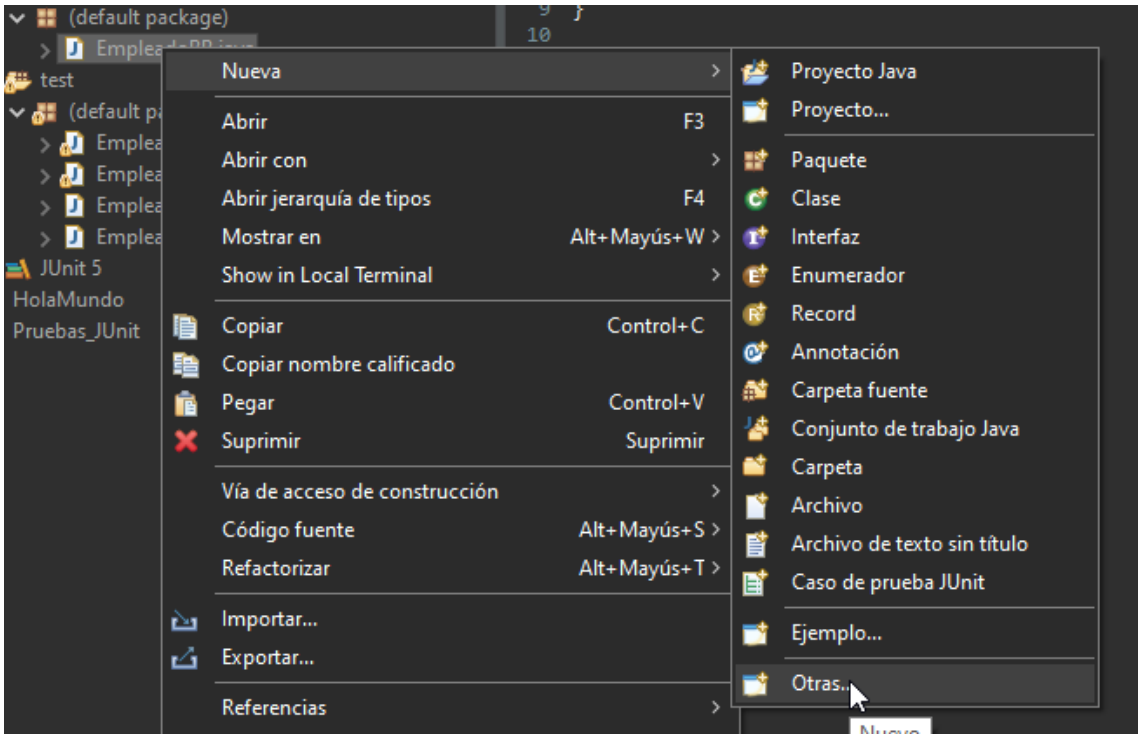
```
@Test
public void testCalculaSalarioNeto() throws Exception{
    EmpleadoBR empleado = new EmpleadoBR("vendedor", 0.0f, 0.0f);
    float resultado = empleado.calculaSalarioNeto(salarioBruto);
    assertEquals(expected, resultado, 0.01);
}
```

Resultados:



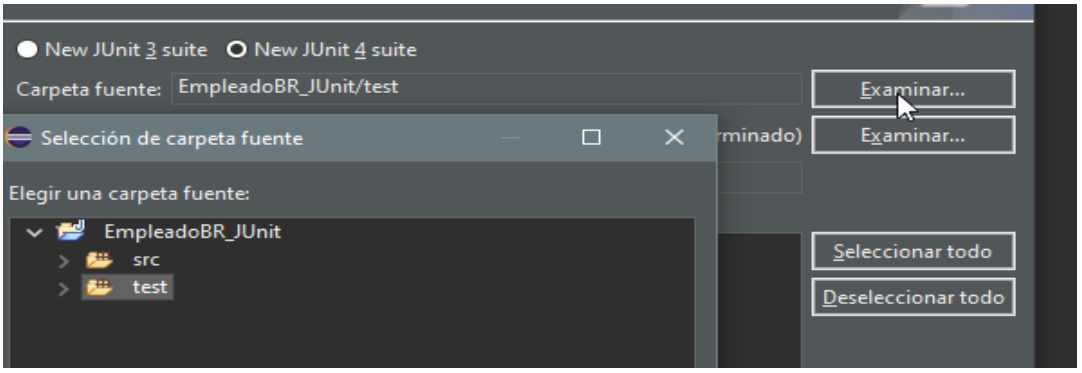
Ejercicio 3: Realiza con Eclipse una suite de pruebas.

Primero tenemos que crear la suite:



Clic derecho > Nueva > Otras.

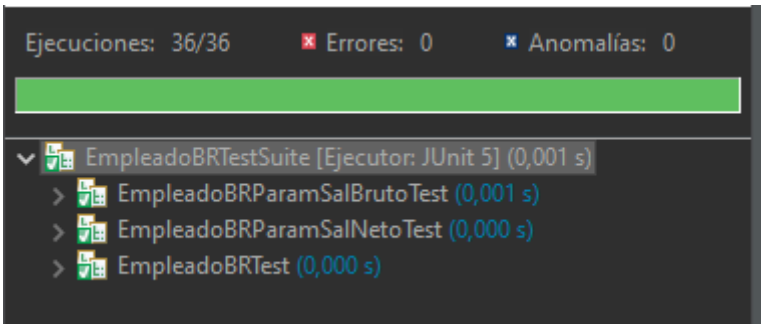
Buscamos suite de pruebas JUnit y seleccionamos siguiente.



Seleccionamos examinar y escogemos la carpeta en la que tenemos todos los tests.

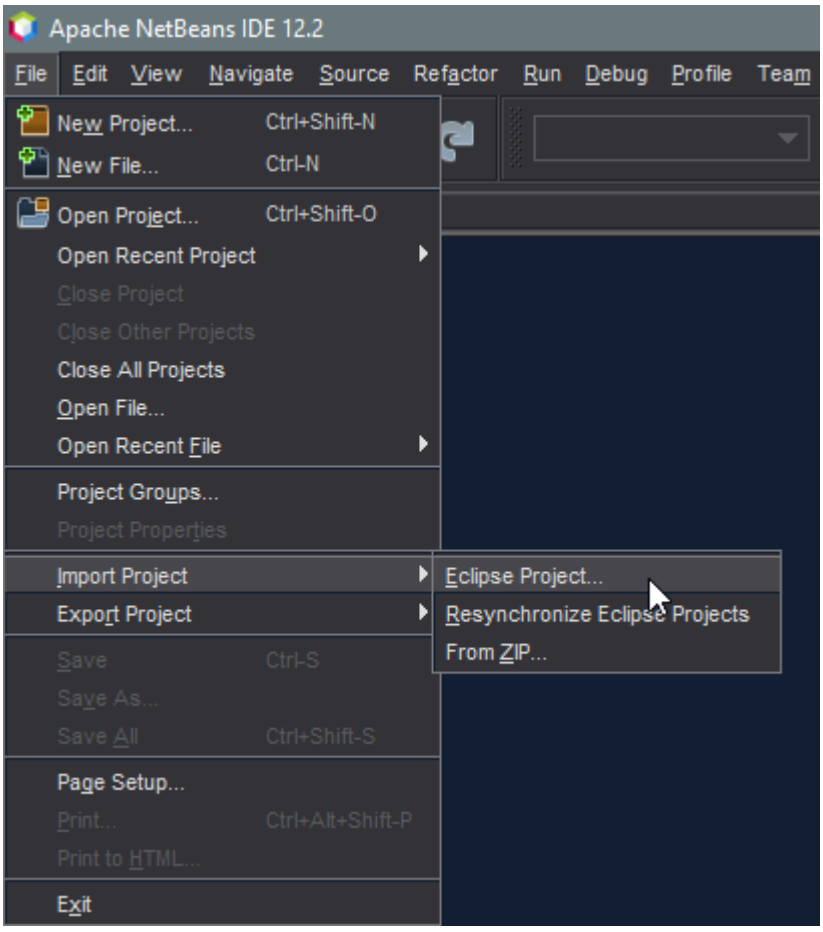
Seleccionamos las clases para incluirlas en la suite.

Ejecutemos la suite para comprobar que todo funciona:

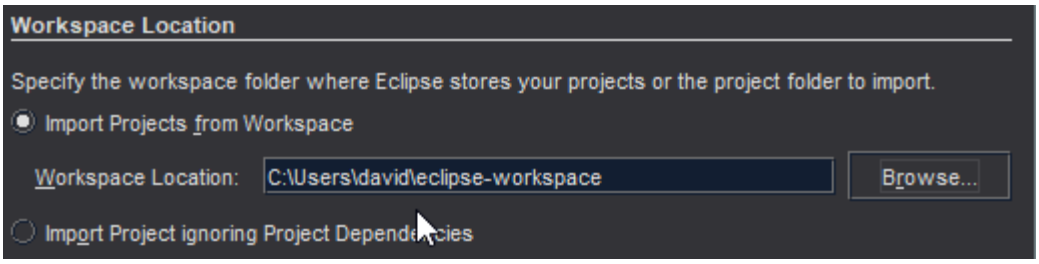


Ejercicio 4: Realiza lo mismo para el IDE Netbeans.

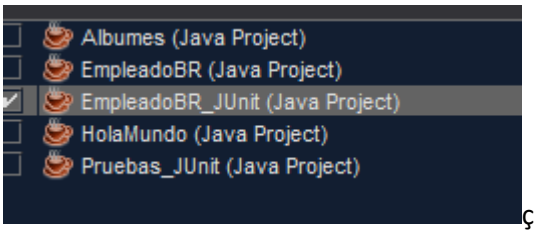
Desde Netbeans se puede importar un proyecto de Eclipse, así que importamos el proyecto que hemos realizado en Eclipse a Netbeans.



Tendremos que dar la carpeta del espacio de trabajo de Eclipse.

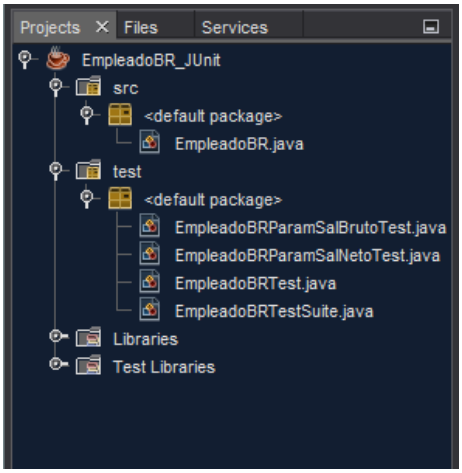


Seleccionamos siguiente, y seleccionamos los proyectos a importar.

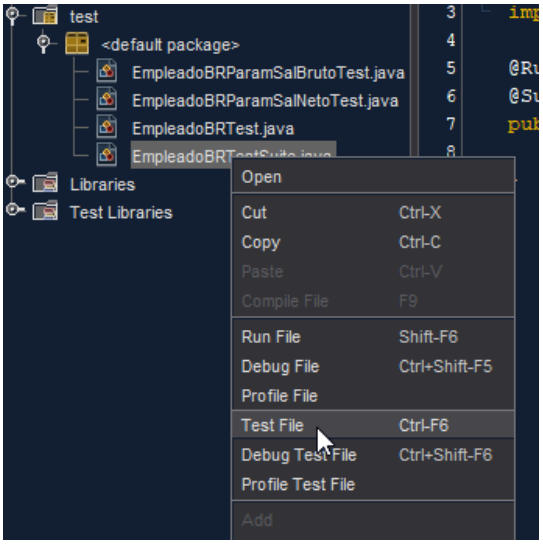


ç

Una vez importado ya tenemos nuestras clases.



En Netbeans, para ejecutar los clasos de prueba, debemos ir a los tests, seleccionarlos con clic derecho, y seleccionar *Test File*.



Obtendremos los mismos resultados: