

DEPURACIÓN DEL CÓDIGO

1. Depuración del código.....	2
1.1. Descripción inicial del código.....	2
1.1.1. Función del código.....	2
1.1.2. Métodos más relevantes.....	3
1.1.3. Valor inicial del saldo (balance) antes de realizar cualquier operación.....	4
1.2. Puntos de control (Breakpoints).....	4
1.2.1. Breakpoints. Dónde y porqué.....	4
1.3. Variables y flujos de ejecución.....	5
1.4. Excepciones.....	5
2. Pruebas unitarias.....	6
2.1. Creación de AccountTest.....	6

1. Depuración del código

1.1. Descripción inicial del código

1.1.1. Función del código.

La función de este código es la de imitar un sistema bancario con sus funciones básicas, la de ingresar y retirar dinero de una cuenta bancaria. Cabe destacar que en este programa, se utilizan los bloques *try-catch* para manejar los posibles errores de este.

Primeramente, se activa el objeto de la clase *Account* con los datos del usuario de la cuenta bancaria. Con su nombre y apellidos, el número de cuenta y el saldo disponible.

```
myAccount = new Account("Flor Martinez", "1000-1234-56-123456789", 2500);
```

Se retira una cierta cantidad de dinero (2300€) de la cuenta de Flor y se ingresa otra cantidad de dinero (1695€). Si hay algún error con alguna de las dos funciones, nos dará un mensaje de error para cada caso.

<pre>try { myAccount.withdrawAmount(2300); } catch (Exception e){ System.err.println(e.getMessage()); System.out.println("Error al retirar"); }</pre>	<pre>try { System.out.println("Ingrés al compte"); myAccount.depositAmount(1695); } catch (Exception e){ System.err.println(e.getMessage()); System.out.println("Error en l'ingrés"); }</pre>
---	---

Para retirar

Para ingresar

Finalmente, se nos imprime un mensaje donde nos muestra el saldo actual del cliente.

```
System.out.println("El saldo actual es " + myAccount.getBalance());
```

1.1.2. Métodos más relevantes.

Podemos destacar algunos de los métodos más importantes de este programa.

1) *Account(String name, String account, double balance)*

Es un constructor para inicializar una cuenta.

```
public Account(String name, String account, double balance) {  
    super();  
    this.name = name;  
    this.account = account;  
    this.balance = balance;  
}
```

2) *getBalance()*

Te devuelve el saldo actual.

```
public double getBalance() {  
    return balance;  
}
```

3) *depositAmount(double amount)*

Nos permite hacer un ingreso y valida que la cantidad no sea negativa.

```
public void depositAmount(double amount) throws Exception  
{  
    if (amount<0)  
        throw new Exception("No es pot ingressar una quantitat negativa.");  
    balance += amount;  
}
```

4) *withdrawAmount(double amount)*

Nos permite realizar una retirada, nos valida que la cantidad deseada y que haya saldo suficiente.

```
public void withdrawAmount(double amount) throws Exception  
{  
    if (amount < 0)  
        throw new Exception ("No es pot retirar una quantitat negativa.");  
    if (getBalance()< amount)  
        throw new Exception ("No hi ha suficient saldo");  
    balance -= amount;  
}
```

1.1.3. Valor inicial del saldo (balance) antes de realizar cualquier operación.

Antes de realizar cualquier operación, el valor inicial de *balance* es de 2500, establecido en el archivo *Main.java* cuando se crea el objeto *myAccount* utilizando también el constructor de la clase *Account*.

```
myAccount = new Account("Flor Martinez", "1000-1234-56-123456789", 2500);
```

1.2. Puntos de control (Breakpoints)

1.2.1. Breakpoints. Dónde y porqué.

Línea 10

Este breakpoint permite comprobar si la creación de la instancia de la clase *Account* está funcionando correctamente, incluyendo la asignación de los valores iniciales (*nombre*, *cuenta*, *saldo*).

```
10 myAccount = new Account("Flor Martinez", "1000-1234-56-123456789", 2500);
```

Línea 13

Detener la ejecución aquí permite verificar si el saldo inicial es suficiente para realizar el retiro y asegurarse de que no se produzca ninguna excepción.

```
13 myAccount.withdrawAmount(2300);
```

Línea 15

Determina si el método *withdrawAmount()* genera una excepción y para verificar el mensaje de error que se produce.

```
15 System.err.println(e.getMessage());
```

Línea 21

Inspeccionar si el método *depositAmount()* actualiza correctamente el saldo del objeto *myAccount*.

```
21 myAccount.depositAmount(1695);
```

Línea 27

Verificar si el saldo se ha actualizado correctamente después de las operaciones de retiro e ingreso.

```
27 System.out.println("El saldo actual es " + myAccount.getBalance());
```

1.3. Variables y flujos de ejecución

Recorre el valor y al restar 2300 se habrá restado del valor inicial

args	String[0] (id=21)
myAccount	Account (id=20)
> account	"1000-1234-56-123456789" (id=
balance	200.0
> name	"Flor Martinez" (id=30)

Al recorrer un poco más se puede ver como finalmente pasa a la segunda parte que es sumar 1695.

no method return value	
> this	Account (id=20)
amount	1695.0
no method return value	
args	String[0] (id=21)
myAccount	Account (id=20)
> account	"1000-1234-56-123456789" (id=24)
balance	1895.0
> name	"Flor Martinez" (id=30)

Aquí se puede ver ya el valor después de retirar e ingresar

1.4. Excepciones

Aquí pondremos un valor negativo que lo que hará es que salga un mensaje de error

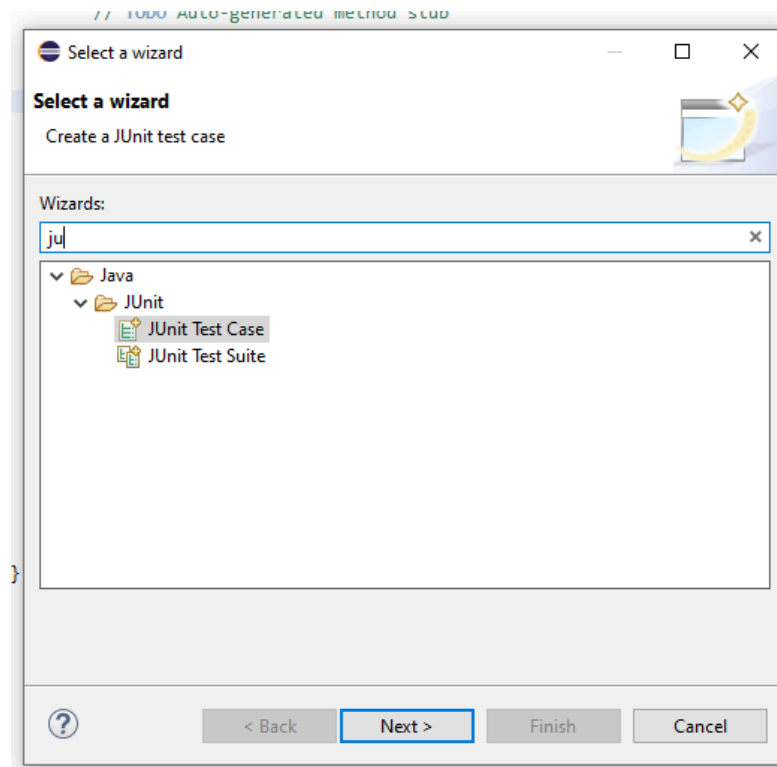
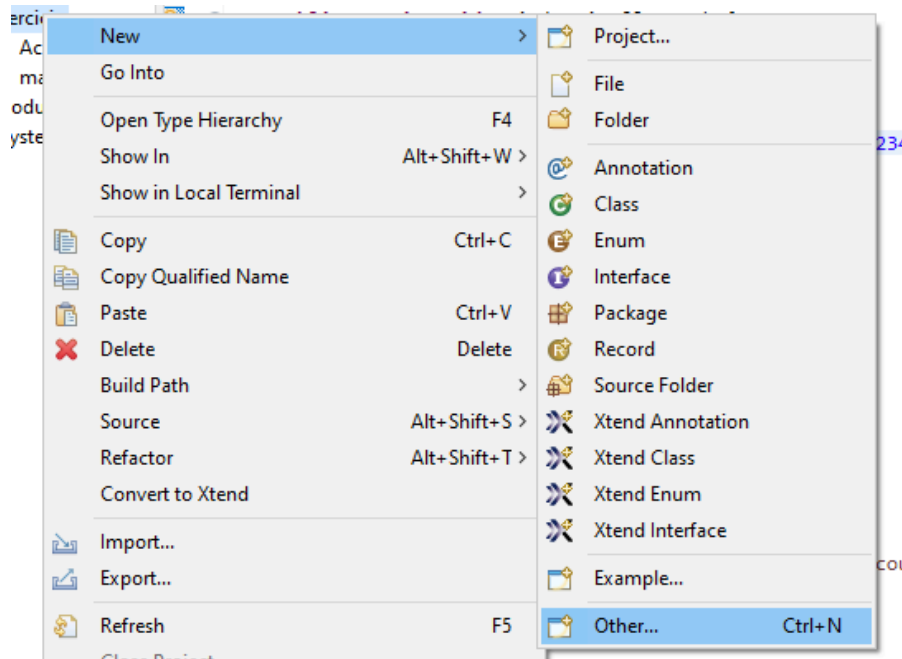
```
myAccount.depositAmount(-1695);
```

Aquí se puede ver el mensaje de error:

```
Ingrés al compte
No es pot ingressar una quantitat negativa.
Error en l'ingrés
El saldo actual es 200.0
```


2. Pruebas unitarias

2.1. Creación de *AccountTest*



New JUnit Test Case

JUnit Test Case

 This package name is discouraged. By convention, package names usually start with a lowercase letter

☐ New JUnit 3 test ☐ New JUnit 4 test ☒ New JUnit Jupiter test

Source folder:

Package:

Name:

Superclass:


Which method stubs would you like to create?

☐ @BeforeAll setUpBeforeClass() ☐ @AfterAll tearDownAfterClass()
☐ @BeforeEach setUp() ☐ @AfterEach tearDown()
☐ constructor

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

☐ Generate comments

Class under test:



New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.


Available methods:

Class	Method	Selected
Account	Account()	<input type="checkbox"/>
	Account(String, String, double)	<input type="checkbox"/>
	getName()	<input type="checkbox"/>
	setName(String)	<input type="checkbox"/>
	getAccount()	<input type="checkbox"/>
	getBalance()	<input type="checkbox"/>
	depositAmount(double)	<input checked="" type="checkbox"/>
	withdrawAmount(double)	<input checked="" type="checkbox"/>
Object	Object()	<input type="checkbox"/>
	getClass()	<input type="checkbox"/>
	hashCode()	<input type="checkbox"/>
	equals(Object)	<input type="checkbox"/>

2 methods selected.

☐ Create final method stubs

☐ Create tasks for generated test methods



```

1 package Ejercicio;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 class AccountTest {
6
7     @Test
8     void testDepositAmount() {
9         fail("Not yet implemented");
10    }
11
12    @Test
13    void testWithdrawAmount() {
14        fail("Not yet implemented");
15    }
16 }
17
18 package Ejercicio;
19
20 import static org.junit.jupiter.api.Assertions.*;
21
22 class AccountTest4 {
23
24     @Test
25     void testDepositAmount() {
26         Account myAccount1 = new Account("Alexis Buza", "1234-5678-90-123456789", 6000);
27         Account myAccount2 = new Account("Simeon Makenzo", "12345-67890", 5000);
28
29         try {
30             myAccount1.depositAmount(1460);
31             myAccount2.depositAmount(4630);
32         } catch (Exception e) {
33             fail("Error en el ingreso: " + e.getMessage());
34         }
35
36         assertEquals(7460.0, myAccount1.getBalance(), 0.001);
37         assertEquals(9630.0, myAccount2.getBalance(), 0.001);
38     }
39
40     @Test
41     void testWithdrawAmount() {
42         Account myAccount1 = new Account("Alexis Buza", "1234-5678-90-123456789", 6000);
43         Account myAccount2 = new Account("Simeon Makenzo", "12345-67890", 5000);
44
45         try {
46             myAccount1.withdrawAmount(1460);
47             myAccount2.withdrawAmount(4630);
48         } catch (Exception e) {
49             fail("Error en la retirada: " + e.getMessage());
50         }
51
52         assertEquals(4540.0, myAccount1.getBalance(), 0.001);
53         assertEquals(370.0, myAccount2.getBalance(), 0.001);
54     }
55 }

```