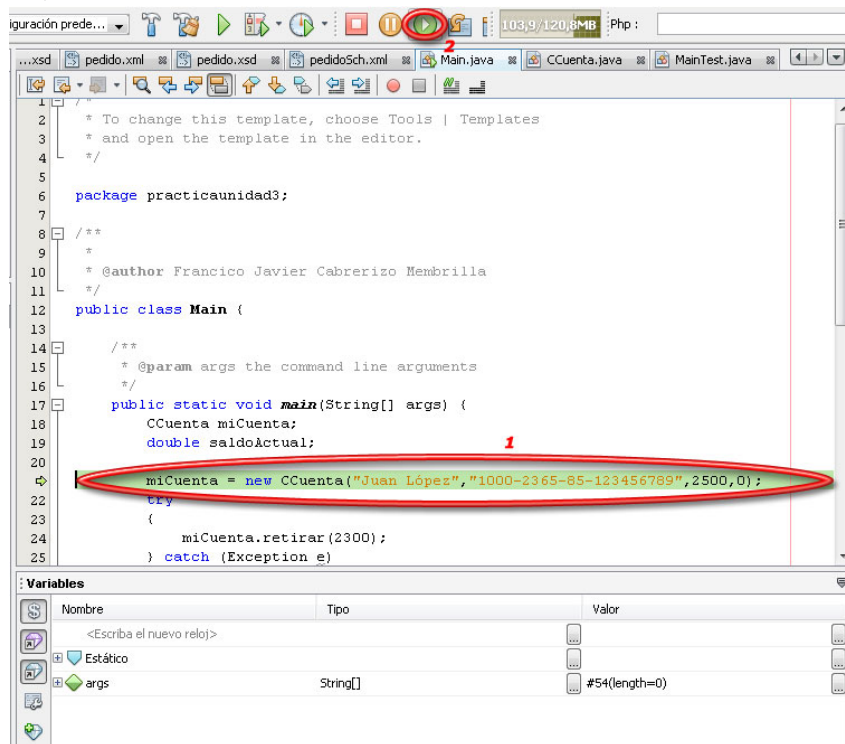


Ejercicio 1

Para ejecutarlo paso a paso se ha de pulsar “F7” o en la opción “Depurar” seleccionamos “Paso a paso”, tras lo cual nos aparecerá nuestra aplicación detenida en la primera orden ejecutable que encuentre, es decir, veremos:



Ahora, de nuevo pulsamos **F7** o volvemos a darle sobre **Depurar/Paso** a paso y nos traslada al método constructor que llamaba la anterior orden pasándole los valores indicados. Como estamos creando un objeto, veremos que cuando se está creando **CCuenta** se llama al método **Object**.

Dentro del constructor invocado se le asignan los correspondientes valores a sus atributos y se le devuelve el control a **Main**. Tras la creación del objeto **miCuenta** del tipo **CCuenta**, y si pulsamos de nuevo **F7** pasamos el control a la siguiente instrucción que hace que entremos en un bloque **try-catch** donde se llama al método **retirar** para el objeto **miCuenta** pasándole un valor numérico entero de **2300** (la conversión de entero a doble es intrínseca, pero no al revés).

Cuando entramos en dicho método, las variables utilizadas poseen los valores:

Variables			
	Nombre	Tipo	Valor
	<Escriba el nuevo reloj>		
	this	CCuenta	practicaunidad3.CCuenta@1...
	nombre	String	"Juan López"
	cuenta	String	"1000-2365-85-123456789"
	saldo	double	2500.0
	tipoInterés	double	0.0
	cantidad	double	2300.0

Dentro del método se comprueba que no se hayan introducido valores numéricos negativos ≤ 0 y que exista suficiente saldo en cuenta (primero llama al método estado para que nos indique cuál es el saldo disponible y después realiza la comprobación) **estado()**<cantidad, para al final restarle la cantidad introducida al saldo disponible (todo ello pulsando **F7** para que vaya pasando de instrucción en instrucción) y nos devuelve el control al Main.

El cursor de ejecución paso a paso lo tenemos ahora sobre el final del bloque try-catch (no ha dado errores) y la variable saldo se ha actualizado teniendo un valor de 200:

```

17 public static void main(String[] args) {
18     CCuenta miCuenta;
19     double saldoActual;
20
21     miCuenta = new CCuenta("Juan López", "1000-2365-85-123456789", 200.0, 0.0);
22     try
23     {
24         miCuenta.retirar(2300);
25     } catch (Exception e)
26     {
27         System.out.print("Fallo al retirar");
28     }
29
30     try
31     {
32         System.out.println("Ingreso en cuenta");

```

Nombre	Tipo	Valor
<Escriba el nuevo reloj>		
Estático		
args	String[]	#54(length=0)
miCuenta	CCuenta	practicaunidad3.CCuenta@1...
nombre	String	"Juan López"
cuenta	String	"1000-2365-85-123456789"
saldo	double	200.0
tipoInterés	double	0.0

Si seguimos pulsando **F7** nos metemos en el siguiente bloque **try-catch** donde nos muestra por pantalla el mensaje de **Ingreso en cuenta** (para realizar todo este proceso de poner en pantalla un texto se necesita hacer multitud de llamadas a *PrintStream*, *Writer*, *BufferedWriter*, *OutputStreamWriter*, *Buffer*, *ByteBuffer*, *BufferedReader*, *BufferedOutputStream*, *CharBuffer*, etc que son métodos incorporados automáticamente por Java y que no tenemos por qué verificar su funcionamiento, por lo que podríamos poner el cursor justo sobre la línea de la llamada al método y pulsar **F4** para que ejecute todo el proceso seguido hasta llegar a donde tenemos situado el cursor) y realiza una llamada al método **ingresar** del objeto **miCuenta** pasándole un valor numérico entero de **695**. Si volvemos a pulsar **F7** accede la ejecución a dicho método, donde se comprobará si se le ha pasado una cantidad positiva, ya que de no ser así, lanzará una excepción general con el mensaje "No se puede ingresar una cantidad negativa". Como le hemos pasado el valor **695**, al pulsar **F7** pasa a la instrucción siguiente al **if** comprobado como falso, y le añade el valor de la variable **cantidad** al contenido actual de **saldo**.

Nombre	Tipo	Valor
<Escriba el nuevo reloj>		
this	CCuenta	practicaunidad3.CCuenta@1aaf...
nombre	String	"Juan López"
cuenta	String	"1000-2365-85-123456789"
saldo	double	200.0
tipoInterés	double	0.0
cantidad	double	695.0

Antes que "saldo" reciba el valor de "cantidad"

Nombre	Tipo	Valor
<Escriba el nuevo reloj>		
this	CCuenta	practicaunidad3.CCuenta@1aaf...
nombre	String	"Juan López"
cuenta	String	"1000-2365-85-123456789"
saldo	double	895.0
tipoInterés	double	0.0
cantidad	double	695.0

Después que "saldo" reciba el valor de "cantidad"

Ahora nos encontramos en el método principal **Main** sobre la línea en que la variable **saldoActual** recibe el valor de lo que devuelva el método **estado** de nuestro objeto **miCuenta**, por lo que para comprobarlo pulsamos **F7** y entramos en la clase **CCuenta** sobre el método **estado** que, cuando pulsemos nuevamente **F7**, nos devolverá el valor actual de **saldo**, y volveremos al método principal.

Ahora estamos sobre la instrucción que nos muestra el mensaje “El saldo actual es” y el valor de la variable **saldoActual**, el cual es de **895,0**. Para ejecutarla y pasar a la siguiente línea, sin necesidad de pasar por todos los métodos que precisa dicha orden, antes pusimos el cursor en la siguiente instrucción y pulsamos sobre **F4**, y ahora vamos a hacerlo pulsando **Ctrl+F7** que se encarga de realizar todo el proceso que se necesite en la línea donde nos encontremos y salga del método para continuar con la siguiente instrucción, por lo que pulsamos **F7** para entrar en el primer método llamado por **System.out.println** que es **PrintStream** y es entonces cuando pulsamos **Ctrl+F7** para ejecutar todo este conjunto de llamadas y devolver el control a la instrucción del método llamante, que en este caso es el **Main**. Ya al final si pulsamos de nuevo **F7** cuando nos encontramos sobre la penúltima llave de cierre, entramos en los métodos de comprobación de errores **Thread** los cuales podemos ejecutar y salir al método principal de nuevo pulsando nuevamente **Ctrl+F7**, y finalizaremos la ejecución de la aplicación.

Ejercicio 2

Para comprobar el método ingresar con JUnit lo primero que hemos de hacer es seleccionar la clase que lo contiene y con el botón derecho pulsamos sobre **Herramientas / Crear pruebas JUnit**, tras lo cual nos aparece una pantalla para solicitar el nombre que le deseamos poner a la clase de prueba, si deseamos crear un método inicializador (**setUp**) y finalizador (**tearDown**), etc.

Dejamos las opciones por defecto y pulsamos sobre aceptar.

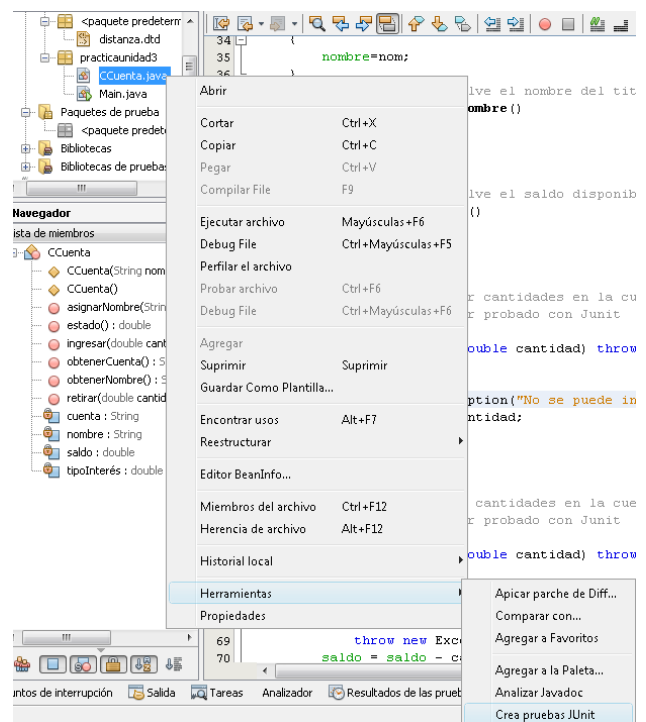
En la nueva clase que nos crea JUnit comentamos o eliminamos todos los métodos que no necesitamos comprobar, quedándonos únicamente con el método constructor **CCuentaTest**, el inicializador **setUp**, el finalizador **tearDown**, y el del que deseamos probar **testIngresar**, en el que pondremos barras de comentario sobre la línea **fail**, ya que en caso contrario, durante la prueba, nos saltará un error en dicha línea.

Cuando hagamos lo indicado, en la clase **CCuentaTest** tendremos el siguiente código:

```
package practicaunidad3;

import junit.framework.TestCase;

/**
 *
 * @author José Luis
 */
```



```

public class CCuentaTest extends TestCase {
    CCuenta instance = new CCuenta();

    public CCuentaTest(String testName) {
        super(testName);
    }

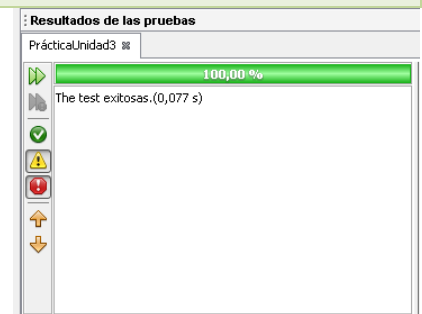
    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testIngresar() throws Exception {
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }
}

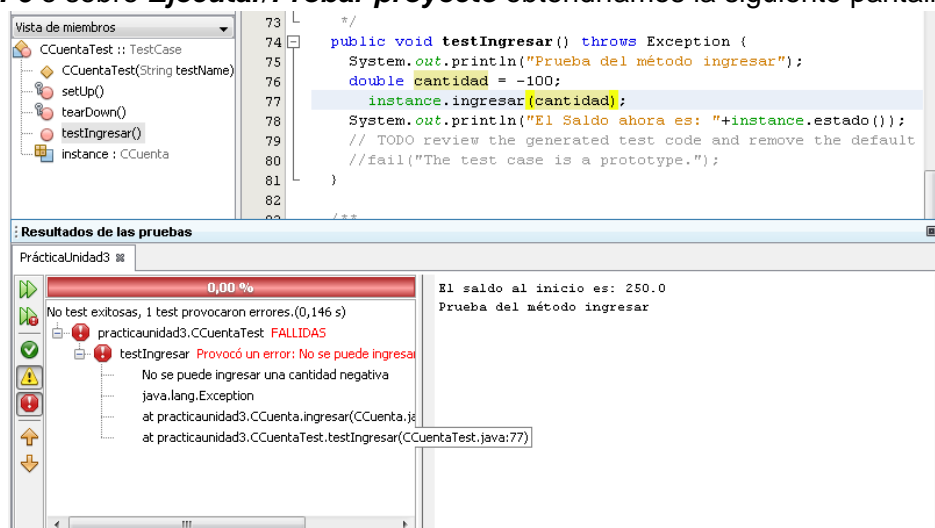
```

Y lo primero que hacemos es probarlo para ver que no nos da ningún error (*aún no hemos diseñado ninguna prueba, por lo que no puede fallar*) obteniendo la siguiente pantalla:



Aprovechamos para poner unos inicializadores de los valores que le pasaremos a nuestro método de prueba, dentro del método **setUp** para que se tomen dichos valores antes de comenzar la ejecución de la clase, por ejemplo, inicializamos una variable **double** a **250**, usamos el objeto creado en la clase de prueba para acceder al método **ingresar** y comprobamos antes de realizar las pruebas de dicho método que su valor se introduce de forma adecuada en **saldo** mediante la llamada al método **ingresar**.

Ponemos las instrucciones que nos servirán para probar nuestro método dentro de **testIngresar**, y en nuestro caso, indicaremos que se compruebe si admite números negativos, obteniendo un aviso de error en el método **ingresar**, aunque en esta ocasión no es JUnit quien controla o descubre el error sino que ya poseemos una instrucción en dicho método que se encarga de ello. Tras pulsar **Alt+F6** o sobre **Ejecutar/Probar proyecto** obtendríamos la siguiente pantalla:



Con esta información podemos comprobar que el método falló dando el error: *No se puede ingresar una cantidad negativa*, y dentro de nuestra prueba falló la instrucción de la **línea 77**, es decir, la llamada al método comprobado.

Ahora vamos a probar a poner un bucle en el que se obtengan valores muy elevados con el fin de comprobar si se recorre de forma adecuada o falla antes de su finalización. El código quedaría:

```
package practicaunidad3;

import junit.framework.TestCase;

/**
 *
 * @author José Luis
 */
public class CCuentaTest extends TestCase {
    CCuenta instance = new CCuenta();

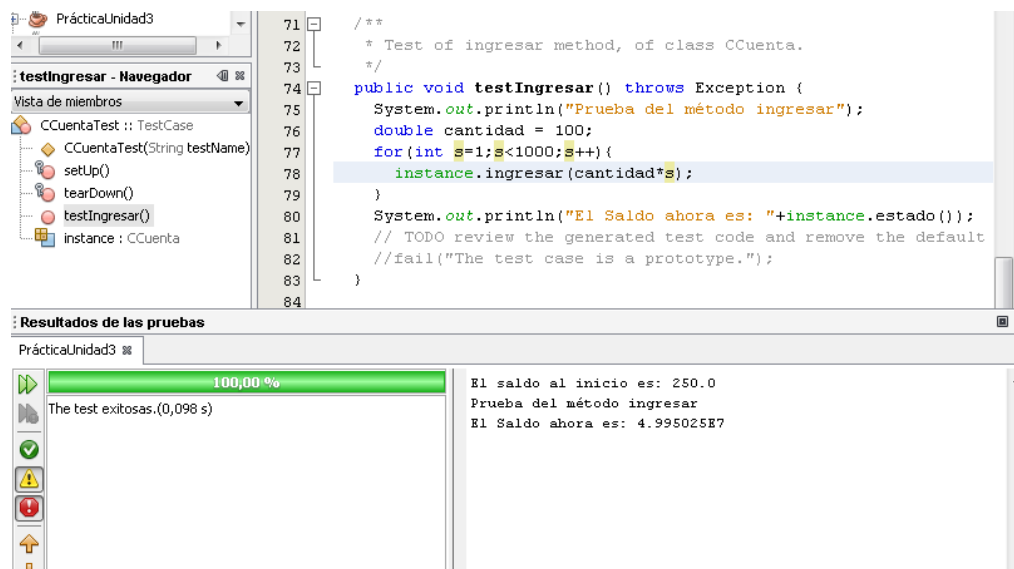
    public CCuentaTest(String testName) {
        super(testName);
    }

    @Override
    protected void setUp() throws Exception {
        double prueba=250;
        instance.ingresar(prueba);
        System.out.println("El saldo al inicio es: "+prueba);
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

    /**
     * Test of ingresar method, of class CCuenta.
     */
    public void testIngresar() throws Exception {
        System.out.println("Prueba del método ingresar");
        double cantidad = 100;
        for(int s=1;s<1000;s++){
            instance.ingresar(cantidad*s);
        }
        System.out.println("El Saldo ahora es: "+instance.estado());
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }
}
```

Y la prueba se realiza sin problemas:



Ejercicio 3

Para este ejercicio actuaremos igual que en el caso anterior, pero no usaremos los métodos **setUp** y **tearDown** ya que no vamos a inicializar valores de otras variables antes de la ejecución real del método que probaremos, ni necesitamos vaciar la caché de datos cuando finalice la comprobación.

Cuando pulsemos el botón derecho sobre la clase que contiene el método a comprobar, y seleccionemos **Herramientas / Crear pruebas JUnit**, y en la pantalla que nos aparece desmarcamos las casillas de **probar inicializador** y **probar finalizador**, con lo cual nos creará una clase de prueba sin los métodos **setUp** y **tearDown**, aunque en realidad no es una clase de prueba nueva, sino la anterior con todos los métodos probatorios nuevos excepto los anteriormente nombrados que se mantienen iguales.

En esta clase de prueba nos quedaremos únicamente con el método que comprueba la función **retirar**, por lo que tendremos:

```
package practicaunidad3;

import junit.framework.TestCase;

/**
 *
 * @author José Luis
 */
public class CCuentaTest extends TestCase {
    CCuenta instance = new CCuenta();

    public CCuentaTest(String testName) {
        super(testName);
    }

    /**
     * Test of retirar method, of class CCuenta.
     */
    public void testRetirar() throws Exception {
        System.out.println("Prueba del método retirar");
        double cantidad = 0.0;
        instance.retirar(cantidad);
        // TODO review the generated test code and remove the default call to fail.
        // fail("The test case is a prototype.");
    }
}
```

Al ejecutarlo por primera vez (**Alt+F6**) comprobamos que nos da un error (*controlado en el método probado*) y se debe a que nos toma por negativa la cantidad introducida para retirar, siendo dicha cantidad de **0** por lo que descubrimos que se trata de un error en el código del programa, el cual modificaremos en su línea:

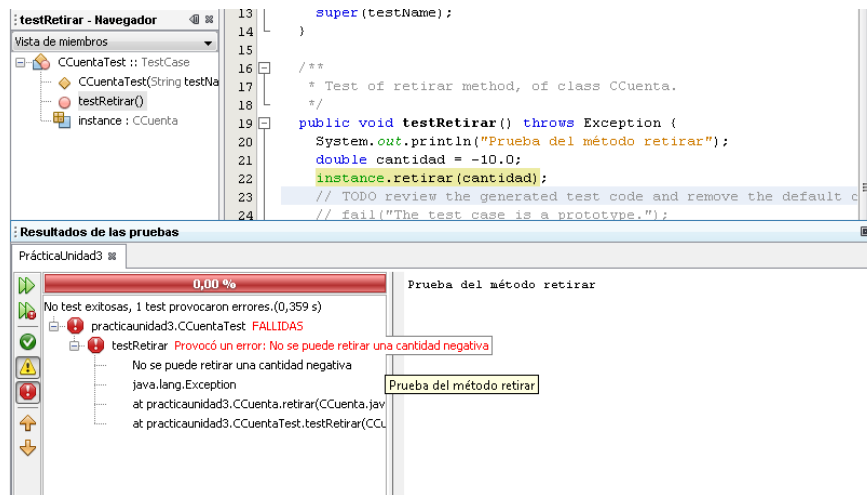
```
if (cantidad <= 0)
    throw new Exception ("No se puede retirar una cantidad negativa");
```

cambiando su comprobación **cantidad <= 0** por **cantidad < 0**, y volvemos a comprobar con JUnit obteniendo, en esta ocasión, un resultado de ejecución correcta.

Probaremos ahora a enviar un valor negativo modificando el código y tecleando:

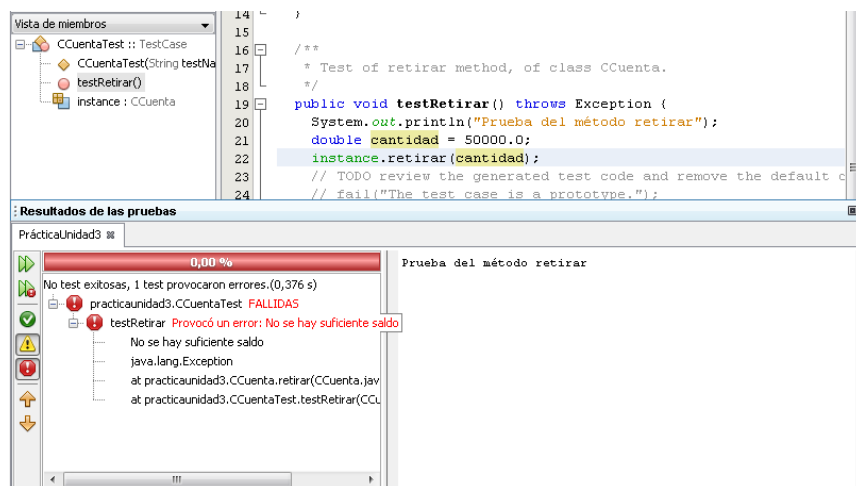
```
public void testRetirar() throws Exception {
    System.out.println("Prueba del método retirar");
    double cantidad = -10.0;
    instance.retirar(cantidad);
    // TODO review the generated test code and remove the default call to fail.
    // fail("The test case is a prototype.");
}
```

Con lo que nos aparecerá:



Vemos que sí recoge de forma correcta el error de introducción de valores negativos, por lo que descubrimos que el método funciona bien (lanza el error: *No se puede retirar una cantidad negativa*)

Ahora probaremos intentando descontar más dinero del existente en el saldo actual, por lo que volvemos a cambiar el código, asignándole a la variable **cantidad** el valor de 50000 y volviendo a comprobar su ejecución, obteniéndose:



Con ello comprobamos que también está controlada la posible introducción de valores que superen el saldo existente, pero falta por comprobar si se intenta descontar una cantidad que sí exista en el saldo, por lo que ponemos:

```
public void testRetirar() throws Exception {
    System.out.println("Prueba del método retirar");
    double cantidad = 1000.0;
    System.out.println("Saldo: " + instance.estado());
    instance.retirar(cantidad);
}
```

En esta comprobación nos dará error, pero descubrimos que el saldo que nos muestra es igual a 0.0 con lo que es normal que nos dé error por no poder descontar nada a dicho valor.

Para corregirlo, sólo hemos de poner la declaración del objeto con sus parámetros para darle unos valores iniciales, es decir, teclearemos al principio de la clase de prueba, en sustitución de la anterior declaración:

```
CCuenta instance = new CCuenta("Juan López", "1000-2365-85-123456789", 2500, 0);
```

Tras esta modificación, probamos la clase de prueba y nos dará todo correcto.

Ejercicio 4

En esta ocasión podríamos proceder del mismo modo que en los dos puntos anteriores, pero manteniendo los métodos, aunque considero más adecuado modificar la clase de prueba que poseemos en estos momentos, pudiendo teclear:

```
package practicaunidad3;

import junit.framework.TestCase;

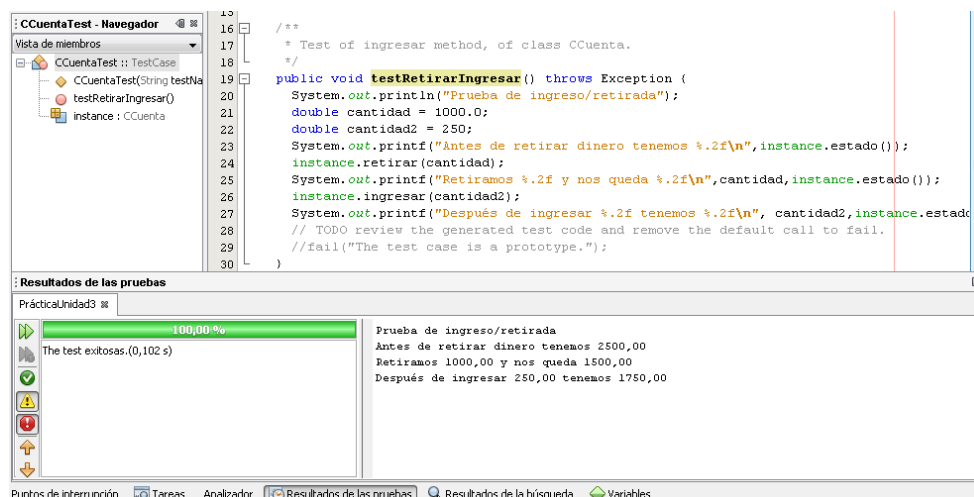
/**
 *
 * @author José Luis
 */
public class CCuentaTest extends TestCase {
    CCuenta instance = new CCuenta("Juan López", "1000-2365-85-123456789", 2500, 0);

    public CCuentaTest(String testName) {
        super(testName);
    }

    /**
     * Test of ingresar method, of class CCuenta.
     */
    public void testRetirarIngresar() throws Exception {
        System.out.println("Prueba de ingreso/retirada");
        double cantidad = 1000.0;
        double cantidad2 = 250;
        System.out.printf("Antes de retirar dinero tenemos %.2f\n", instance.estado());
        instance.retirar(cantidad);
        System.out.printf("Retiramos %.2f y nos queda %.2f\n", cantidad, instance.estado());
        instance.ingresar(cantidad2);
        System.out.printf("Después de ingresar %.2f tenemos %.2f\n", cantidad2, instance.estado());
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }
}
```

con ello, damos unos valores iniciales al objeto **instance** y creamos un método **testRetirarIngresar** en el que inicializamos dos variables dobles a fin de que nos sirvan para poder realizar las llamadas a los métodos **retirar** e **ingresar**. Y en este caso, como partimos de un **saldo** de **2500** dada la inicialización del objeto, vamos a retirar **1000** y a ingresar **250**, todo ello con salida en pantalla de los valores que van obteniéndose en **saldo**.

Al pulsar **Alt+F6** obtenemos la siguiente salida:



Que nos informa que la ejecución se ha realizado de forma correcta.

Como ya realizamos otras pruebas en los puntos anteriores, ahora mostraré otra forma de poder hacer comprobaciones utilizando varios métodos distintos dentro de la clase test, en lugar de unir todas las comprobaciones en el mismo método como acabamos de realizar.

La forma más ortodoxa de realizar pruebas sobre cualquier método de una clase es poner un método por cada uno a comprobar, es decir, que el método **testRetirarIngresar** que creamos antes podría haberse realizado poniendo el siguiente código:

```
package practicaunidad3;

import junit.framework.TestCase;

/**
 *
 * @author José Luis
 */
public class CCuentaTest extends TestCase {
    CCuenta instance = new CCuenta("Juan López","1000-2365-85-123456789",2500,0);

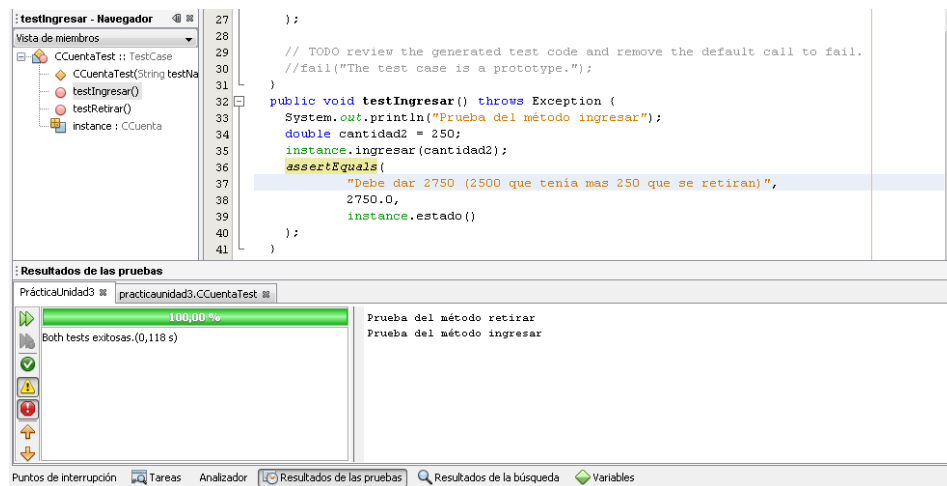
    public CCuentaTest(String testName) {
        super(testName);
    }

    /**
     * Test of ingresar method, of class CCuenta.
     */
    public void testRetirar() throws Exception {
        System.out.println("Prueba del método retirar");
        double cantidad = 1000.0;
        instance.retirar(cantidad);
        assertEquals(
            "Debe dar 1500 (2500 que tenía menos 1000 que se retiran)",
            1500.0,
            instance.estado()
        );
    }

    /**
     * Test of retirar method, of class CCuenta.
     */
    public void testIngresar() throws Exception {
        System.out.println("Prueba del método ingresar");
        double cantidad2 = 250;
        instance.ingresar(cantidad2);
        assertEquals(
            "Debe dar 2750 (2500 que tenía mas 250 que se ingresan)",
            2750.0,
            instance.estado()
        );
    }
}
```

Lo cual nos dará el mismo resultado que la prueba anterior (es *todo correcto*) pero nos ofrece la libertad de poder probar cada método por separado con lo que ganamos en claridad ya que si nos da error en algún método rápidamente sabríamos qué había fallado, pero al situarlos todos dentro de uno mismo, tendríamos que estar comprobando en qué línea, si ha fallado por algo puesto dentro del mismo método, etc.

También he optado en este ejercicio por usar el método **assert** que tiene bastantes posibilidades, como en el ejemplo que se encargará de darnos un error sólo si no se cumple la condición de que el **saldo** haya alcanzado el valor que debería. Si se cumple la condición el método continúa sin poner ningún mensaje, por ello, tras su ejecución obtendremos:



Como vemos, el método **`assertEquals`** comprobará si son iguales los valores alcanzados por el saldo y los que hemos dicho nosotros que se deben alcanzar.

Esta última forma de poner las pruebas es la más utilizada, aunque hay que tener en cuenta que si las comprobaciones se ponen todas en el mismo método se corre el riesgo de ir arrastrando los resultados de una operación a la siguiente, en cambio, al utilizar dos métodos independientes, cada uno comenzará a utilizar los valores como si fuese único. Esto se puede comprobar con esta última forma propuesta, en la que el saldo alcanzado después de retirar (*a la finalización del primer método*) no es el mismo con el que se comienza el segundo método (*ingresar*), por lo que la comprobación con el método **`assertEquals`** no debe controlar el valor obtenido teniendo en cuenta los anteriores métodos ejecutados, sino los inicializados en la clase de prueba en el método inicializador o al principio de la clase test.

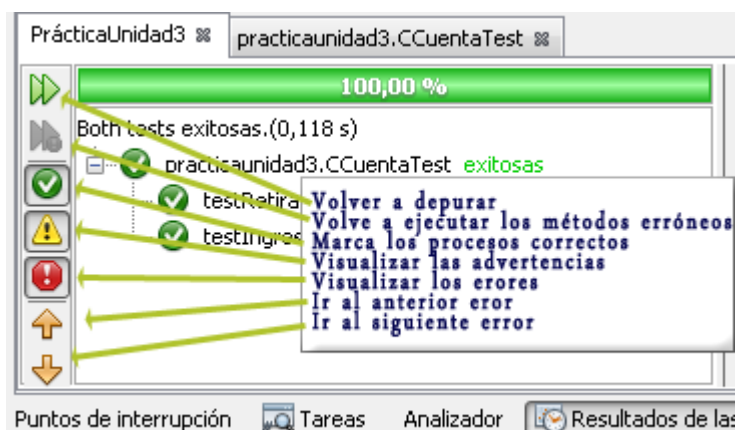
Ejercicio 5

Todas las pruebas realizadas con ***JUnit*** nos hace ver la utilidad de este depurador/verificador de código abierto que viene instalado (o se le puede instalar) en la mayoría de los IDE java existentes en el mercado.

En los ejercicios 2 y 3 utilicé ***JUnit*** con instrucciones java comunes, sin hacer uso de los métodos **`assert`** que nos ofrece para la realización de pruebas y comprobaciones, pero intentaba centrarme en la utilidad de la herramienta ***JUnit*** sin desviarme con otros métodos que nos hiciesen centrarnos en dichas funciones, sin dejarnos ver la amplitud de este framework.

Quando deseamos dar unos valores iniciales usaremos el método **`setUp`** para incluirlos, y si usamos listas, vectores, ficheros o cualquier otro método de acceso a datos que requieran de apertura de conexión y cierre de la misma, podría utilizar el método mencionado anteriormente para iniciar la conexión y el método **`tearDown`** para finalizarla una vez termine la comprobación.

Quando ejecutamos el test nos aparece una ventana de resultados que podremos utilizar para volver a comprobar el código entero o sólo los módulos que han dado error, ver cuáles son los métodos correctos, aquellos que nos dan una advertencia o los que producen un error y detienen la ejecución. Así mismo, disponemos de dos botones en forma de flecha



arriba y abajo que nos permiten ir al anterior o al siguiente error.

Volviendo a los ejercicios anteriores, en la comprobación para el método **ingresar** podríamos haber tecleado lo mismo que se puso para su método en el ejercicio 4, y además podríamos haber utilizado también, antes del anterior **assert**:

```
assertNotNull("Debería tener valor",instance.estado());
```

para saber si el saldo tiene valor, ya que dicha orden comprueba que lo especificado en segundo término contenga un valor no nulo, y si se confirma que no es nulo continúa la ejecución y en caso contrario nos mostrará el texto tecleado en primer término. Si utilizásemos la orden **assertNull** sería justamente lo contrario, comprueba si el valor del segundo término es nulo, y sería en este caso cuando continuaría sin problemas y en caso contrario saltaría el error.

Realmente considero que este bloque de órdenes que nos ofrecen están bastante bien ya que su misión es comprobar una afirmación (**assert**) para que, en caso que no se cumpla, nos salte el mensaje puesto en primer término y nos muestre el error, y de lo contrario nos permita continuar como si nada.

Todas estas pruebas que nos permite realizar **JUnit** con nuestros proyectos (ya sea con una clase únicamente, un método en particular o con todo el proyecto en conjunto) conseguirán que desarrollemos aplicaciones libres de fallos y por consiguiente, que demuestre la profesionalidad de un programador.