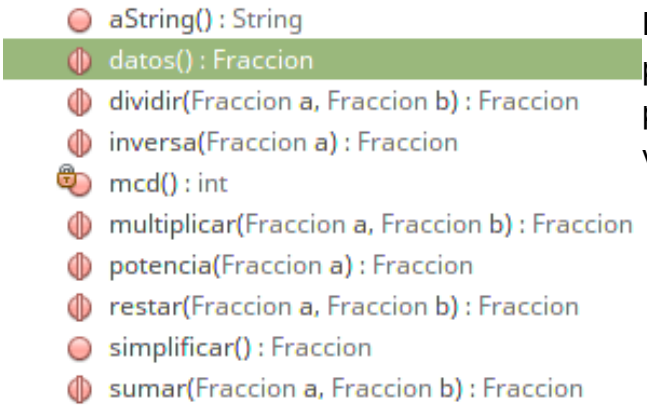
Javier González Rives

Tarea 3.2, Clases de prueba.

Para realizar las pruebas se creara un test Junit, pruebas unitarias, de los métodos para comprobar si los datos devueltos por estos son los correctos, para los métodos que conllevan introducción de datos se realizara una clase para comprobar la introducción de valores y si no ocurren errores.



En este caso todas las pruebas usaran pruebas unitarias excepto “datos()” que sirve para introducir valores y se hará a parte para ver si es correcta.

Datos():

Se ha creado una clase donde poder probar la introducción de datos a trabes de teclado, dos de los puntos mas conflictivos era la introducción de datos no numéricos (!int) y la división por 0, para tratar el primer error se ha introducido el lanzamiento de una excepción cuando no se cumpla la expresión regular ("^[0-9]+/[0-9]+$"), en el caso que el denominador sea 0 se lanza otra excepcion.

```
public class DatosTest {
    public static void main(String[] args) {
        try{
            Fraccion fraccionTest = Fraccion.datos();
            System.out.println(fraccionTest.aString());
        }
        catch(ArithmeticException arith){
            System.out.println(arith.getMessage());
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
    }
}
if(!fraccionDato.matches("^[0-9]+/[0-9]+$"))
    throw new ArithmeticException("hay elementos no numericos");
if(denom > 0)
      f = new Fraccion(num,denom);
else
      throw new ArithmeticException("error division por 0");
```

Javier González Rives

Pruebas unitarias.

Para las pruebas unitarias se usa en primer lugar @BeforeClass y @Afterclass, esto nos sirve para definir lo que se hará antes de las pruebas de los métodos y que se hará después, en este caso hemos creado cuatro instancias de la clase Fraccion llamadas fraccionn siendo la ultima ‘n’ el numero de la fraccion. En el interior de @BeforeClass se ha hecho la instanciación de las fracciones. En cambio en @AfterClass se ha puesto una salida por pantalla para indicar que la prueba ha llegado hasta el final con éxito.

```
 static Fraccion fraccion1,fraccion2,fraccion3,fraccion4;
    @BeforeClass
    public static void setUpClass(){
    fraccion1 = new Fraccion(5);
    fraccion2 = new Fraccion(7,4);
    fraccion3 = new Fraccion(-4,3);
    fraccion4 = new Fraccion(0);
    }
    @AfterClass
    public static void afterClass(){
        System.out.println("fin de la prueba");
    }
```

Suma():

En el caso de suma se han ido sumando las fracciones creadas para comparalas con el resultado que deberían de dar en este caso se ha introducido un cambio en el método original, al final de la suma se simplificaban los resultados.

```
@Test
    public void testSuma(){
        System.out.println("suma");
        //prueba suma1
        Fraccion resultado = new Fraccion(7,4);
        assertEquals(resultado.aString(),Fraccion.sumar(fraccion1,
fraccion2).aString());
        //prueba suma2
        resultado = new Fraccion(5,12);
        assertEquals(resultado.aString(),Fraccion.sumar(fraccion2,
fraccion3).aString());
        //prueba suma3
        assertEquals(fraccion1.aString(),Fraccion.sumar(fraccion1,
fraccion4).aString());
    }
```

Resta():

Para resta se ha llevado un sistema parecido al anterior ya que la operación de suma y resta son prácticamente los mismo.

```
@Test
    public void testResta(){
        System.out.println("resta");
        //prueba resta1
        Fraccion resultado = new Fraccion(13,4);
        assertEquals(resultado.aString(),Fraccion.restar(fraccion1
, fraccion2).aString());
        //prueba resta2
        resultado = new Fraccion(37,12);
        assertEquals(resultado.aString(),Fraccion.restar(fraccion2
, fraccion3).aString());
        //prueba resta3
        resultado = new Fraccion(5);
        assertEquals(resultado.aString(),Fraccion.restar(fraccion1
, fraccion4).aString());
    }
```

Multipicacion():

Es de los métodos menos problemáticos, el producto entre fracciones normalmente no tiene problema y que no necesita un denominador común para llevarse a cabo por ello la prueba consta de un único producto.

```
@Test
    public void testMultiplicar(){
        System.out.println("multiplicaccion");
        Fraccion resultado = new Fraccion(-28,12);
        assertEquals(resultado.aString(),Fraccion.sumar(fraccion2,
fraccion3).aString());
    }
```

Inversa():

```
@Test
    public void testInversa(){
        System.out.println("inversa");
        Fraccion resultado = new Fraccion(4,7);
        assertEquals(resultado.aString(),Fraccion.inversa(fraccion
2).aString());
    }
```

Division():

Este es uno de los puntos conflictivos, las divisiones se pueden hacer con normalidad hasta que aparece la división por 0, en este caso se ha añadido el lanzamiento de una excepción para los casos en los que se intente dividir entre 0.

```
 @Test
    public void testDivision(){
    Fraccion resultado = new Fraccion(20,7);
      assertEquals(resultado.aString(),Fraccion.dividir(fraccion1,
fraccion2).aString());
    try{
        fraccion1.dividir(fraccion2, fraccion4);
        fail("se ha intentado dividir por 0");
    }
    catch(ArithmeticException arith){
        System.out.println(arith.getMessage());
    }
    }
```

Los métodos simplificar, mcd y aString se han probado de forma indirecta ya que sin estos las pruebas anteriormente realizadas no podrían haberse llevado a cabo.

Ejercicio2.

@BeforeClass

Para crear los elementos de la tabla que mas tarde se inicializan en @BeforeClass es necesario crearlos como variables de clase fuera de ningún método y como estaticos.

```
static Integer tabla[] = new Integer[10];
    static TablaEnteros tablilla;
   @BeforeClass
   public static void beforeClass(){
       for(int i = 0; i<tabla.length ; i++ ){
           tabla[i]=i;
       }
       tablilla = new TablaEnteros(tabla);
   }
```

SumaTabla():

Para este método se usa la instancia de TablaEnteros anteriormente creada y se comprueba si el resultado es el adecuado.

```
@Test
    public void testSuma(){
        //prueba de la suma
        System.out.println("suma");
        assertEquals(45, tablilla.sumaTabla());
    }
```

MayorTabla():

Este método devuelve el elemento mayor de la tabla, como esta en orden del 0 al 9 sabemos que va salir el 9.

```
@Test
    public void testMayor(){
        // prueba mayor
        System.out.println("mayor");
        assertEquals(9,tablilla.mayorTabla());
    }
```

PosicionTabla(entero):

Este método nos devuelve en que posición esta cierto numero, comprobamos que el resultado coincide. En el caso que el numero no exista nos lanzara una excepción que debemos capturar, tenemos que comprobar con un try y un catch que efectivamente la excepción se produce.

```
@Test
    public void testPosicionTabla(){
        //test posicion
        System.out.println("posicion");
        assertEquals(9,tablilla.posicionTabla(9));
        try{
            tablilla.posicionTabla(33);
            fail("no se ha ejecutado la excepcion");
        }
        catch(NoSuchElementException nS){

        }
    }
```

TablaEnteros(Entero[]):

Para el constructor basta ver si los valores almacenados son correctos ademas hay que comprobar si saltan las excepciones para valores nulos y vacios.

```
@Test
    public void testTabla(){
        // prueba constructores
        Integer entero[] = new Integer[0];
        //caso 0
        try{
            new TablaEnteros(entero);
            fail("no ha saltado la ");
        }
        catch(IllegalArgumentException il){

        }
        //caso null
        entero = null;
        try{
            new TablaEnteros(entero);
            fail("no ha saltado la ");
        }
        catch(IllegalArgumentException il){

        }
    }
```

Todos los archivos estan en mi repositorio de github.

https://github.com/likendero/EntornosDesarrollo/tree/master/test