

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Tema 6 – Pruebas de software, técnicas de diagramación y documentación.

Pruebas de software

Tipos de pruebas

Encontrar un error es un éxito. Esta frase resume muy bien la ideología con la que hay que enfrentarse a grandes proyectos de software. La idea generalizada de algunos programadores, que piensan que su programa no tiene fallos, se acerca a la utopía. Por tanto, a medida que se desarrolla una aplicación, conviene que un analista se encargue de probarla "a fondo" para encontrar posibles problemas. De hecho, lo ideal es que sea una persona distinta la que programe y la que pruebe el software.

El mundo de las pruebas de software es un apartado de la programación que puede llevar tanto o más tiempo que la programación en sí misma. De hecho, quien se conoce como **Analista o Quality Assurance (QA)** es el encargado de realizar todas las tareas que envuelven al mundo de la programación que no son la propia escritura de código (además, es muy odiado por los programadores). Y dentro de estas tareas, quizá una de las más importantes es la realización de pruebas.

En este apartado, por falta de tiempo en la asignatura, nos limitaremos a trabajar con lo que se denomina **pruebas unitarias**, que son las pruebas de código más allá de la compilación.

Es decir, prueban la lógica del diseño del sistema y permiten detectar errores de diseño. En estas nos centraremos mediante la unidad **JUnit**, que nos permite crear código para realizar baterías de pruebas sobre módulos (clases, funciones, ...) que nos interese probar.

Por supuesto, existen otros tipos de prueba en los que no profundizaremos, como:

- **Pruebas de estrés:** se enfocan en sistemas de red (servidores sometidos a mucha carga).
- **Pruebas de integración:** se realizan cuando varios módulos, programados por distintas personas o equipos, se integran y deben seguir funcionando.
- **Pruebas funcionales:** verifican que se cumpla la especificación.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR	Francisco Bellas Aláez (Curro)						



Pruebas de aceptación: Es el conjunto final de pruebas realizado ya por el cliente o usuario que va a utilizar la aplicación. Básicamente es muy similar a las pruebas funcionales pero incluyen elementos subjetivos como la facilidad de uso, la estética, la usabilidad, etc... Además las pruebas no las hacen analistas si no los propios usuarios que van a utilizar la aplicación. En ocasiones se les denomina **pruebas de validación**.

Existen dos **tipos de pruebas de aceptación**:

- Pruebas **alfa**: Las realiza el cliente o usuario final pero siendo observado y supervisado por el desarrollador. Lo normal es realizarlo "en casa" del desarrollador. Es una especie de "prueba in vitro".

Observa el siguiente video:

<https://www.youtube.com/watch?v=QYBCLMiR9b0>

- Pruebas **beta**: La aplicación las realiza el usuario o cliente pero ya en su entorno final. Se informa a los desarrolladores de los problemas o posibles mejoras que se establecerán si se considera necesario en la distribución final de la aplicación.

Puedes ver más tipos de pruebas en el siguiente enlace:

<https://www.hiberus.com/crecemos-contigo/tipos-de-pruebas-de-software-segun-la-piramide-de-cohn/>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Pruebas unitarias y Casos de Prueba (test case)

En este tema nos centraremos, como se indicó previamente, en el desarrollo de las pruebas unitarias. Estas prueban la lógica de distintos módulos de un proyecto y pueden tener dos perspectivas: **Caja blanco o caja negra**.

Caja blanca

Este tipo de análisis consiste en ver y probar la totalidad del código. El objetivo es el hacer que en los test se pase por todas las líneas de código y se compruebe su correcto funcionamiento. Se trata de establecer los distintos caminos por los que circula el flujo de nuestra aplicación y probarlos. Lógicamente esto no es una tarea fácil y existen determinadas reglas a seguir. Existe el llamado cálculo de la **complejidad ciclomática** que nos da el número de caminos necesarios a seguir para realizar todas las pruebas. No lo veremos por falta de tiempo y por que no se usa demasiado hoy en día salvo para partes de software muy críticas.

También se analizan los bucles exhaustivamente de forma que se salte el bucle, que se pase una vez, que se pase varias veces pero un número menor al máximo de vueltas, etc...

Las pruebas de caja blanca también se pueden hacer mediante el debugger ejecutando paso a paso y anotando por qué ramas de código vamos pasando. Es tedioso pero sería lo más eficiente. Sólo se aplicaría a casos muy particulares donde el probabilidad de error tiene que ser mínima o nula (aplicaciones médicas, centrales nucleares, militares,...).

En cualquier caso y aunque se use más la caja negra siempre se pueden complementar los dos métodos.

De cara al funcionamiento del **JUnit** es la realización de pruebas de **caja negra** y por tanto la generación de casos de prueba para ese tipo de test por lo que no profundizaremos más en la caja blanca.

Caja negra:

En este tipo de pruebas se analiza el módulo (en nuestro caso el módulo será una clase con diversas funciones) desde un punto de vista externo. El analista no se preocupa del código si no de que para determinadas entradas las distintas funciones ofrezcan las salidas esperadas sin analizar el código fuente.

Vemos las especificaciones del mismo y se crea una batería de pruebas (**Casos de**

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

prueba) que barra todas las posibilidades de funcionamiento.

Tratamos de someterlo tanto a entradas válidas como no válidas. Son especialmente importantes también las entradas frontera entre válidas y no válidas.

En el siguiente ejemplo se plantean los casos de prueba para una situación dada:

Se desea realizar un módulo que trabaja con números comprendido entre 1 y 100 ambos incluidos. Debemos probar a introducir:

1. Número menor que 1
2. Número entre 1 y 100
3. Número mayor que 100
4. Casos extremos o frontera: 0, 1, 100 y 101
5. Otras situaciones: Número muy grande que sobrepase la capacidad del tipo (Overflow), algo que no sea un número (si se puede introducir), etc. Esto ya depende del uso que se le dé o de la especificación.

En el ejemplo anterior la primera, la tercera y la quinta son llamadas **clases de equivalencia inválidas** (son valores con los que el módulo no va a trabajar pero a la vez no debe fallar si le llega uno de ellos) y la segunda una **clase de equivalencia válida**.

Pero también es de gran importancia analizar los casos extremos, el punto 4, ya que es una fuente de errores habitual (incluye válidas e inválidas): límites en los índices de arrays o colecciones, de rangos de trabajo de un bucle o if, uso de cadenas vacías, objetos null, etc.

Si tras someterlo a todas esas pruebas los resultados son los esperados se dice que el test ha sido pasado con éxito.

Se ven por tanto tres grupos de pruebas: por un lado establecer **valores válidos**, por otro **valores inválidos** y por último comprobación de los **valores límite**.

Además si alguna de las pruebas falla y se cambia algo, **es necesario volver a realizar todas las pruebas**, y en un caso tan simple como el el primer ejemplo

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Entornos de desarrollo			CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)			

habría que hacer un mínimo de 9 pruebas cada vez que se cambie algo.

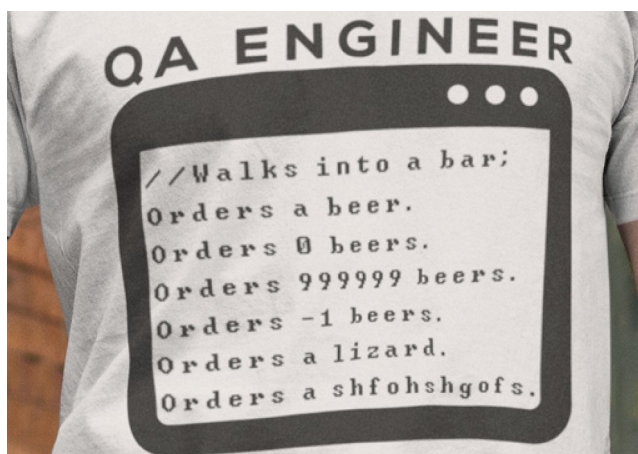
De lo que se trata es que para una entrada de datos determinada hay que establecer una o varias entradas válidas y una o varias inválidas y ver como se comporta en todos los casos.

Por ejemplo, si estuviéramos trabajando con un vector, la idea sería similar pero ajustada a ese tipo de dato, por ejemplo habría que probar: vector nulo, vector vacío, vector con un elemento, varias pruebas con dos o más elementos.

Además si hubiera funciones que acceden al vector habría que probar un acceso correcto a las fronteras (índices: -1,0, length-1 y length), valores válidos (entre 0 y length-1), y fuera de rango (negativos y mayores o iguales que length).

Es decir, hay que adaptar las pruebas a los tipos de datos.

También si una función tiene más de un parámetro habría que cursar **todas las combinaciones de pruebas con todos los parámetros**.



Evidentemente en aplicaciones grandes los casos de prueba pueden ser enormes y por ello conviene realizar un buen análisis de lo que realmente es necesario. Además la posibilidad de automatizar las pruebas es importante porque normalmente hay que hacerlas varias veces.

Es decir, yo acabo un programa (o un módulo) y lo pruebo. Si falla, lo arreglo y vuelvo a hacer **TODAS** las pruebas una y otra vez hasta que considero que no falla.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Si tengo que hacer unas 1000 pruebas y no lo tengo automatizado el análisis no es viable.

En lugar de hacer a mano las pruebas, lo ideal es programarlas y ejecutarlas de forma automática cada vez que cambio algo en el programa. Puede parecer una pérdida de tiempo estar programando pruebas en vez de código final pero cuando el volumen de pruebas crece tanto y es necesario repetirlo en cada cambio a la larga sale beneficioso.

Esta automatización de pruebas unitarias es lo que haremos mediante JUnit en Java.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

JUnit

Una vez comprendida grosso modo la metodología de pruebas de software vamos a llevar a la práctica parte de lo visto. Para ello usaremos el framework **JUnit** en su versión 5.

La versión actual es solo a partir de Java 8 por lo que si alguien necesitara realizar pruebas unitarias en versiones previas debería utilizar JUnit 4. Se puede ver la versión usada (y configurar) en el pom.xml o archivo equivalente según el gestor de proyectos usado.

JUnit es un framework para realizar pruebas de software. Por tanto consta de una serie de clases que permiten crear código para realizar las pruebas. Es código que ejecuta y prueba nuestro código. Existen otros muy similares como TestNG y su uso es muy similar.

En otros lenguajes existen frameworks similares. Por ejemplo en C# (y .Net en general) existe el denominado NUnit, o en PHP el PHPUnit. Las bases son las mismas.

Uso básico de JUnit

El uso de JUnit es sobre un proyecto Maven ya existente. Comienza entonces creando un nuevo proyecto y dentro una **clase** pública denominada **Operaciones** metiéndola ya en un **paquete** denominado **pruebas**.

Crea un nuevo proyecto Maven y escribe el siguiente código que debería entenderse a la perfección a estas alturas:

```
public class Operaciones {

    public static int factorial(int n) {
        int res = 0;

        for (int i=1;i<n;i++) {
            res*=i;
        }
        return res;
    }
}
```

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR	Francisco Bellas Aláez (Curro)						

```

public static boolean primo(int n) {
    boolean esPrimo = true;
    for (int i = 2; i < n; i++)
        if (n % i == 0) {
            esPrimo = false;
        } else {
            esPrimo = true;
        }
    return esPrimo;
}
}

```

Si lo revisas con detenimiento efectivamente vas a encontrar errores lógicos. Están puestos con el propósito de realizar las pruebas de software y detectarlos.

En Maven, dentro de **src/test/java** y en el package correspondiente está la clase **ApplicationTest**. Esta es el test que se encarga de ejecutar las pruebas JUnit que programemos.

```

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("Application")
public class ApplicationTest {

    @Test
    @DisplayName("Pointless test")
    void smokeTest() {
        assertThat(true).isEqualTo(true);
    }
}

```

Lo primero que llama la atención es una importación estática. Esto permite acceder a las funciones de la clase importada como si fueran propias (sin poner el nombre de la clase).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Prueba por ejemplo este código y verás que no te da ningún error:

```
import static java.lang.System.out;
import static java.lang.Math.*;

public class Application {
    public static void main(String[] args) {
        out.println("Raíz de 12: "+sqrt(12));
    }
}
```

Como ves se importan las funciones, no las clases. Ojo, no debes abusar de esto pues **no está considerado una buena práctica**.

Solo se usa en casos particulares como JUnit donde se está trabajando de forma casi exclusiva con métodos estáticos de la propia librería y así se evita estar continuamente repitiendo el nombre de las clases.

Si quieres leer más sobre usos de la directiva static revisa este artículo:

<https://www.adictosaltrabajo.com/2015/09/17/la-directiva-static-en-java/>

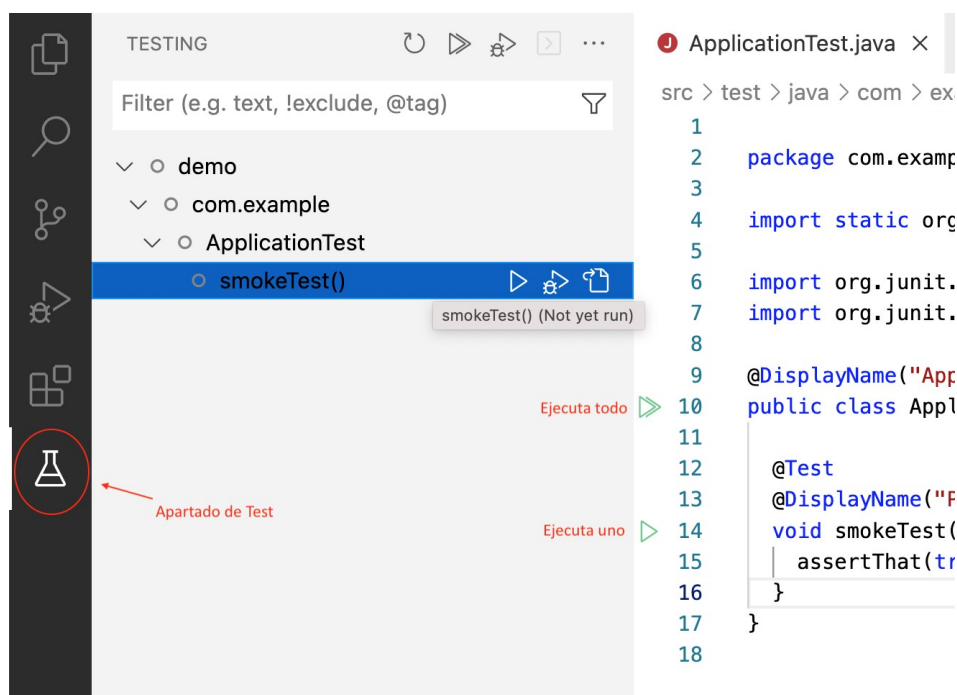
En la clase **ApplicationTest** creada automáticamente (puede tener otro nombre dependiendo del arquetipo usado), aparece la función `smokeTest` (prueba de humo) delante de la cual aparece una anotación en Java: **@Test**.

Estas anotaciones son similares a las de Javadoc o al `@Override` usado en herencia y sirven en este caso para asociar el código que viene a continuación con pruebas internas de JUnit. Por decirlo de forma simple, `@Test` indica que a continuación hay una función que debe ejecutarse para realizar una prueba. En general una anotación permite tener metadatos en un programa que se pueden usar en tiempo de compilación o de ejecución.

Vemos que aparecen dos anotaciones más que no afectan al funcionamiento de los test pero sirven para dar información sobre lo que se testea. Usan ambas la anotación **@DisplayName** una para la clase y otra para la función. Prueba a cambiar la cadena que tiene y ejecuta el test. De todas formas el nombre de la función debería ser claro.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Para ejecutar el test se puede ejecutar en el Activity Bar en el apartado de test (El icono del matraz) o directamente en el margen del editor pulsando en la clase (ejecutaría todos los test de la clase) o en la función (ejecutaría solo ese test).



Veremos otras anotaciones a medida que nos hagan falta.

Como curiosidad y aunque no suele ser muy usado (salvo en el desarrollo de librerías o Frameworks como JUnit), se pueden **crear anotaciones a medida**. Puedes ver ejemplos en : <https://www.javatpoint.com/custom-annotation>

Para realizar los test dentro de las funciones de prueba usaremos los **asserts** (afirmaciones) que son un conjunto de funciones de JUnit que prueban resultados.

Para este primer ejemplo probaremos los siguientes asserts:

assertEquals(resultadoReal, resultadoPrueba):

La prueba es correcta si resultadoReal==ResultadoPrueba.

assertTrue(condicionPrueba):

La prueba es correcta si condicionPrueba es true.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

assertFalse(condicionPrueba):

La prueba es correcta si condicionPrueba es false.

Veámoslo en funcionamiento para tenerlo más claro. Escribe el código en las funciones de prueba de forma que probamos varios factoriales y comprobamos varios números primos.

```
@Test
public void testFactorial() {
    assertEquals(120, Operaciones.factorial(5));
    assertEquals(1, Operaciones.factorial(1));
    assertEquals(1, Operaciones.factorial(0));
    assertEquals(6227020800L, Operaciones.factorial(13));
}
```

```
@Test
public void testPrimo() {
    int[] primos = {2, 3, 5, 7, 11, 13, 17, 19,
        23, 29, 31, 37, 41, 43, 47, 53, 59,
        61, 67, 71, 73, 79, 83, 89, 97};
    int[] noPrimos = {4, 6, 8, 9, 10, 12, 14,
        15, 16, 18, 20, 21, 22, 24, 25, 26,
        27, 28, 30};
    for (int numero : primos) {
        assertTrue(Operaciones.primo(numero), "Falla el " + numero);
    }
    for (int numero : noPrimos) {
        assertFalse(Operaciones.primo(numero), "Falla el " + numero);
    }
    assertFalse(Operaciones.primo(0));
    assertFalse(Operaciones.primo(-10)); //Conveniente probar más negativos
    assertTrue(Operaciones.primo(-7));
}
```

Fíjate que a las funciones de prueba se les denomina igual pero anteponiendo la palabra test delante y aplicando camelCase.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Entornos de desarrollo			CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)			

Es posible que falten imports, lo más cómodo es usar el asterisco:

```
import org.junit.jupiter.api.Assertions.*;
```

Para ejecutar este código no es necesario crear una función main pues JUnit la crea por nosotros y luego se encargan de ejecutar las funciones marcadas con la anotación @Test.

Ejecuta ahora el test completo. Tras esto verás el informe de las pruebas. En este caso las tres han fallado.

The screenshot shows an IDE with a 'TESTING' panel on the left and a code editor on the right. The 'TESTING' panel shows a list of tests: 'demo' (97ms), 'com.example' (97ms), and 'ApplicationTest' (Mi clase de...). Under 'ApplicationTest', three tests are listed: 'testFactorial()' (16ms), 'testPrimo()' (53ms), and 'testGradosARadianes()' (2...). All three tests are marked with a red 'X' icon, indicating they failed. The 'testFactorial()' test is selected. The code editor shows the source code for 'ApplicationTest.java'. It includes imports for JUnit and a class 'ApplicationTest' with a method 'testFactorial()' annotated with '@Test'. The method calls 'assertEquals(120, Operaciones.factorial(5))'. A red error message is displayed below the code: 'Expected [120] but was [0] testFactorial()'. Below this message is a table with two columns: 'Expected' and 'Actual'. The 'Expected' column shows '-120' and the 'Actual' column shows '-0'. To the right of the table, there is a list of test results, including 'Test run at 16/4/20...', 'testFactorial() Jav...', 'Expected [120] but ...', 'org.opentest4j.Ass...', and 'testPrimo() Java T...'.

Los fallos son precisamente las indicaciones de que no han pasado el test siempre que el error sea un `AssertException`. Por supuesto al probar puede saltar algún otro tipo de excepción que no se `assert` y que no deba saltar.

Empezando por **testFactorial** el informe indica que se esperaba 120 y se obtiene 0 por tanto es lógico que indique que el test

```
assertEquals(120, Operaciones.factorial(5));
```

ha fallado ¿Sabrías indicar por qué?

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

Una posible corrección sería inicializar **res a 1**. Pruébalo nuevamente y verás que sigue fallando. Ahora da 24 cuando debería dar 120.

En el bucle debes **sustituir el < por <=**.

Aún así sigue fallando pero ahora es otro el test que falla:

```
assertEquals(6227020800L, Operaciones.factorial(13));
```

Esto sucede porque al trabajar con int hay problemas con números grandes. Prueba a pasar a long el valor devuelto y res. Quedaría la función así:

```
public static long factorial(int n) {
    long res = 1;

    for (int i=1;i<=n;i++) {
        res*=i;
    }
    return res;
}
```

Prueba ahora el test, debería salir validado.

✓	✗	demo	104ms
✓	✗	com.example	104ms
✓	✗	ApplicationTest	Mi clase de Pruebas: 104ms
✓	✓	testFactorial()	35ms
	✗	testPrimo()	51ms
	✗	testGradosARadianes()	18ms

Ahora revisamos **primo**, nos indica que hay un error y hemos aprovechado la sobrecarga del *assert* con mensaje para que nos diga cuando falla. Y **falla con el 4**.

Fíjate que eso indica que los primos los detecta como tal pero los no primos también, por lo que evidentemente el algoritmo no funciona.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

Si mejoramos la función de primos de esta forma:

```
boolean esPrimo = true;
for (int i = 2; i < n; i++)
    if (n % i == 0) {
        esPrimo = false;
    }

return esPrimo;
```

Sigue sin pasar el test porque **falla el 0**. Además si lo corriéramos con un if también fallaría el test de números negativos.

Aquí entra la decisión si me interesa una función que contemple los números negativos o no. Esto va a depender de la especificación. En el segundo caso, directamente quito los asserts correspondientes.

En el primero puedo corregirlo así:

```
if (n==0) {
    return false;
}
n=Math.abs(n);
for (int i = 2; i < n; i++){
    if (n % i == 0) {
        return false;
    }
}
return true;
```

En este caso pasaría todos los test para primos.

Como puedes comprobar la ventaja de usar JUnit es que establezco unos test al principio que puedo ejecutar continuamente a medida que hago cambios en el programa sin necesidad de realizar una interfaz de usuario en la que tengo que meter y comprobar datos.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Otros asserts

`assertEquals(resultadoReal, resultadoPrueba, delta):`

La prueba es correcta si

$(\text{resultadoReal} - \text{delta} \leq \text{ResultadoPrueba} \leq \text{resultadoReal} + \text{delta})$

`assertNull(objeto)`

La prueba es correcta si el objeto es *null*

`assertNotNull(objeto)`

La prueba es correcta si el objeto no es *null*

`assertSame(objetoReal, objetoResultado)`

La prueba es correcta si ambos objetos son la misma referencia (Apuntan al mismo sitio)

`assertNotSame(objetoReal, objetoResultado)`

La prueba es correcta si no apuntan al mismo sitio

`assertArrayEquals(arrayReal, arrayResultado)`

La prueba es válida si ambos arrays son iguales (Contienen los mismos valores)

`fail()`

Provoca un fallo incondicional en la prueba.

Todos los métodos pueden llevar **como primer parámetro una cadena de información** en caso de fallo.

Para ver todos los métodos con sobrecargas accede a esta web:

<https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html>

Requisitos temporales

Se puede añadir a un test la anotación **@Timeout** para establecer también un tiempo para realizar la tarea por si consideraos que la rutina ha de estar más optimizada.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

Prueba a añadir el siguiente Test de primos:

```
@Test
@Timeout(value = 50, unit = TimeUnit.MILLISECONDS)
public void testPrimo2() {
    assertTrue(Operaciones.primo(1177027));
    assertTrue(Operaciones.primo(1287961));
    assertTrue(Operaciones.primo(1299827));
}
```

Nota: Esta prueba puede depender del equipo en el que se ejecute.

Prueba a cambiar el bucle y llévalo hasta $\leq n/2$ en lugar de n o $\leq \text{Math.sqrt}(n)$.

Como ves esto está más bien orientado a la optimización de rutinas.

Pruebas de excepciones

También en el caso de querer **probar una función que lanza una excepción** se puede realizar de varias maneras. La forma oficial en versiones actuales es mediante las denominadas **funciones lambda** que no veremos hasta el año que viene, por lo que lo haremos mediante una método más clásico como se ve en el siguiente ejemplo.

Primero añade al principio de la función **factorial** la siguiente sentencia

```
if (n < 0) {
    throw new IllegalArgumentException();
}
```

Y añade el siguiente Test a la parte de JUnit

```
@Test
public void testFactorialException() {
    try {
        Operaciones.factorial(-10);
        fail("No salta excepción");
    } catch (IllegalArgumentException e) {
        assertTrue(true);
    }
}
```


<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

En este caso se ejecuta un fail si no ha saltado la excepción pues es un requisito. Sin embargo si salta, y entra en el catch, indicamos un test positivo.

Nota: Aunque aún no hayamos visto las expresiones lambda, se pueden hacer pruebas de excepciones de forma muy sencilla. Sería algo así:

```
@Test
public void testFactorialExceptionLambda() {
    assertThrows(IllegalArgumentException.class,
        () -> Operaciones.factorial(-10));
}
```

El -> es el denominado **operador lambda** y se usa para crear y pasar como parámetro (entre otras cosas) funciones anónimas. **Por ahora no lo veremos y no debes usarlo.**

Más info en:

<https://billykorando.com/2019/03/04/handling-and-verifying-exceptions-in-junit-5/>

<https://www.adictosaltrabajo.com/2015/12/04/expresiones-lambda-con-java-8/>

Desactivación de pruebas

Si en un momento dado necesitas saltarte alguno de los tests no es necesario comentarlo, llega con poner la anotación **@Disable** antes de @Test y ya no hará dicha prueba. Recuerda hacer el import correspondiente.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Inicializaciones y finalizaciones (Ciclo de vida)

En ocasiones se requiere que antes de ejecutar los test se inicialicen un o más variables u objetos que se van a utilizar durante las pruebas. Para ello JUnit dispone de una serie de anotaciones para indicar cuales son esas funciones de inicialización. Puedes verlas resumidas en este ejemplo en el que habría que completar el código:

```

@BeforeAll
static void AntesDeTodos() {
    //Se ejecuta una sola vez antes de todos los test (ojo, es static).
}

@BeforeEach
void AntesDeCadaUno() {
    //Se ejecuta antes de cada test.
}

@Test
void Test1() {
    //Este sería uno de los múltiples test.
}

@Test
void Test2() {
    //Este sería otro de los múltiples test.
}

@AfterEach
void DespuesDeCadaUno() {
    //Se ejecuta después de cada test.
}

@AfterAll
static void DespuesDeTodos() {
    //Se ejecuta una sola vez después de todos los test (ojo, es static).
}

```

En el caso anterior si se manda ejecutar todos, JUnit lo ejecuta de esta manera:

```

AntesDeTodos()
AntesDeCadaUno()
Test1()
DespuesDeCadaUno()
AntesDeCadaUno()
Test2()
DespuesDeCadaUno()
DespuesDeTodos()

```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Mocking

En ocasiones cuando trabajamos con JUnit necesitamos acceder a ciertas clases o funciones externas a nuestro código que aún no están construidas, que no están probadas (podrían fallar) o a las que no tenemos acceso por cualquier motivo.

En situaciones así podemos "simular" esas clases o funciones de forma muy simple creando un módulo (Mock) que devuelva un único valor válido para unos parámetros fijos.

Por ejemplo puedes tener hacer la función:

```
public void suma(int a, int b){
    return 5;
}
```

Evidentemente la tienes que usar poniendo en a y b valores que devuelvan 5, pero de esta manera puedes probar tus funciones que usen la función suma sin que está esté totalmente programada.

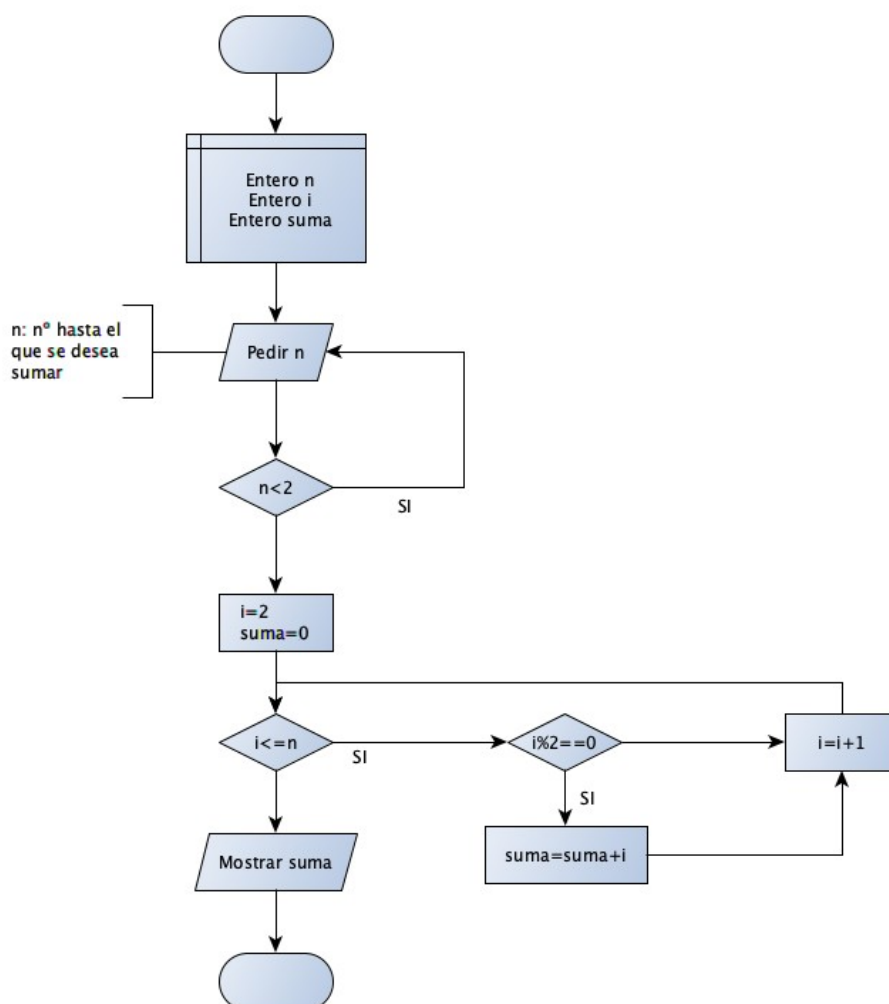
De todas formas para realizar estas tareas de mocking existen herramientas como Mockito que ayudan en gran medida. Por falta de tiempo no entraremos en ellas.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR		Francisco Bellas Aláez (Curro)					

Diagramas de flujo

En la fase de diseño, para planificar los algoritmos, se suelen usar los diagramas de flujo u ordinogramas. Un ordinograma representa paso a paso las instrucciones de un programa o un conjunto de ellas. Refleja la secuencia lógica para la resolución de un problema.

Veamos esto con un ejemplo de suma de números pares hasta uno dado:



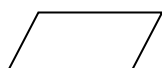
<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR		Francisco Bellas Aláez (Curro)					

Símbolos que utilizaremos en los diagramas de flujo

Existen diversos estándares de simbología para diagramas de flujo como Clásico, SDL, IBM, Gaddis y otros. En nuestro caso usaremos el esquema Clásico y solo algunos bloques esenciales para entender el funcionamiento básico.



Terminal: Inicio/fin de programa o algoritmo. Si es una función se suele indicar el nombre de la misma.

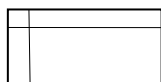


Entrada/salida: cualquier operación de introducción de datos (por ejemplo por teclado) y de salida de datos (por ejemplo por pantalla).

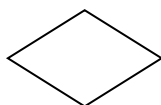
En la entrada se indica sobre qué variable recae, en la salida lo habitual es un mensaje (cadena).



Proceso: Asignación con operaciones aritméticas, lógicas, etc.



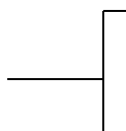
Declaración de variables: Se debe indicar tipo y nombre.



Decisión: Dependiendo de la condición que se establezca en su interior puede tomar como salida una de dos ramas (Sí o No).



Indicadores de dirección: Indican el orden en que se ejecutan los distintos pasos.

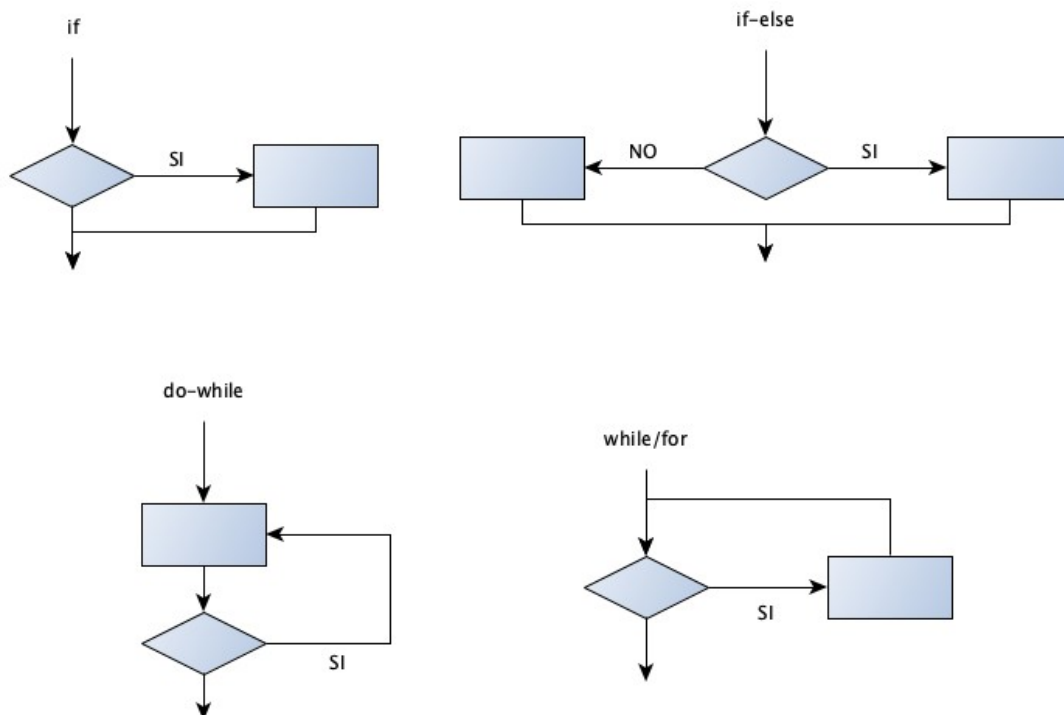


Comentario: Indicaciones que se quieran dar a mayores.

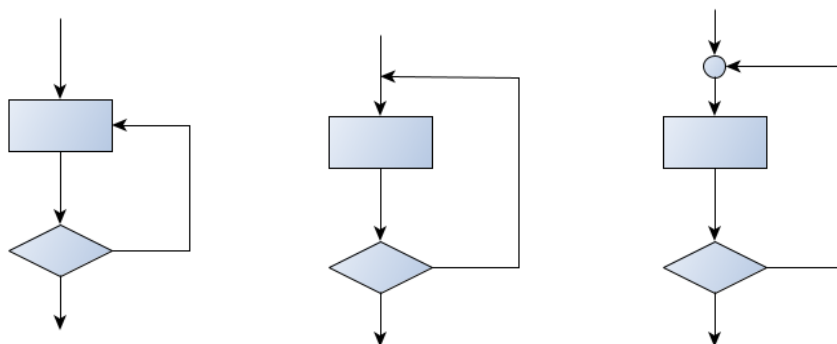
Existen otros pero no los veremos por el momento.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR		Francisco Bellas Aláez (Curro)					

Estructuras básicas vistas



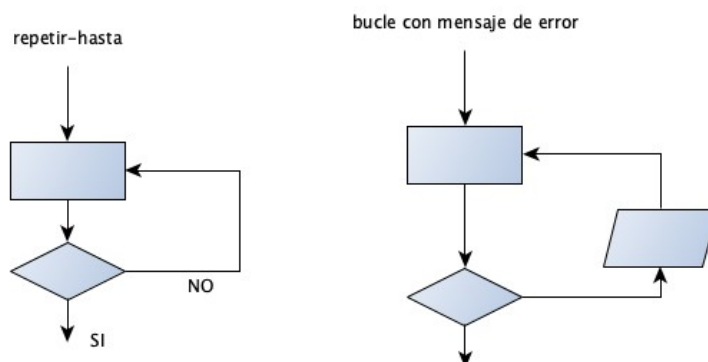
En los bucles las flechas de flujo pueden volver de distintas formas. Todas son válidas:



<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

Estructuras inexistentes en Java.

Hay que tener cuidado cuando se hacen diagramas de flujo pues normalmente son más flexibles que algunos lenguajes de programación. Los ejemplos siguientes son estructuras que no existen en Java (aunque sí se puedan usar en otros lenguajes).




El repetir-hasta existe en lenguajes como Pascal. Lo puedes ver también en el apéndice de pseudocódigo.

El bucle con mensaje de error implicaría que la condición del bucle no está ni al principio ni al final, si no en medio del cuerpo del bucle. En principio Java no dispone de él aunque se podría simular con un break, pero es una práctica no recomendada. En otros lenguajes se puede simular con sentencias como goto (típico en basic) o JMP en ensamblador.

Recomendaciones para el diseño de Diagramas de Flujo.

- Se deben usar solamente líneas de flujo horizontales y/o verticales.
- No deben quedar líneas de flujo sin conectar.
- Se deben trazar los símbolos de manera que se puedan leer de arriba hacia abajo y de izquierda a derecha en la medida de lo posible
- Todo texto escrito dentro de un símbolo deberá ser escrito claramente, evitando el uso de muchas palabras.

Para realizar los diagramas de flujo usaremos el programa **yEd**, pero existen otros como **flowgorithm** (<http://flowgorithm.org>) que facilitan la labor en gran medida,

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR		Francisco Bellas Aláez (Curro)					

permiten la ejecución de los diagramas e incluso traducirlos a otros lenguajes. Puedes usarlo para realizar comprobaciones y entender el funcionamiento básico.

Pseudocódigo

Una herramienta que se usa de forma habitual junto con los diagramas de flujo es el denominado **pseudocódigo** que consiste en escribir el programa como algoritmo pero sin fijarse excesivamente en la sintaxis de lenguaje. Puedes verlo en el [Apéndice I](#) y lo usarás en algún ejercicio.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

Diagrama de Clases UML

Asociado al diseño OO se puede usar la esquemática UML para representar las clases y las relaciones entre ellas. Iremos viendo a lo largo del tema distintos elementos usados en esta esquemática.

Como primer diagrama tenemos el recuadro para representar una clase tal cual se ve a continuación:

Nombre de la clase
+Atributos
+Métodos()

Tiene tres partes donde se coloca respectivamente el nombre, las propiedades y los métodos de dicha clase.

Un ejemplo de una clase en Java y UML:

```
class Perro {
    public String raza;
    public String nombre;
    private int edad;

    public int getEdad() {
        return edad;
    }

    public void setEdad(int a) {
        edad = a;
    }
}
```

El diagrama de Clases del caso anterior sería:

Perro
+raza: String +nombre: String -edad: int
+getEdad(): int +setEdad(a:int): void

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Encapsulado: Modificadores de acceso en UML

Mark	Visibility type
+	Public
#	Protected
-	Private
~	Package

Miembros estáticos

En UML un miembro estático aparece subrayado en el diagrama de clases.

Ejemplo sencillo (sin herencia)

A continuación se presentan unas clases en Java y su equivalente UML:

```
class Comida{
    public String nombre;
    public double precio;

    public Comida(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
}
```

```
class Perro {
    public static String definicion ="El mejor y más baboso amigo del hombre";

    public String raza;
    public String nombre;
    private int edad;

    public Perro() {
        setEdad(0);
        this.raza="";
        this.nombre="";
    }
}
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Entornos de desarrollo			CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)			

```

public Perro(int edad, String raza, String nombre){
    setEdad(edad);
    this.raza=raza;
    this.nombre=nombre;
}

public int getEdad() {
    return edad;
}

public void setEdad(int a){
    edad = a;
}

public void ladrar(){
    System.out.println("GUAU!!!");
}

public void ladrar(int n) {
    for (int i = 0; i < n; i++){
        ladrar();
    }
}

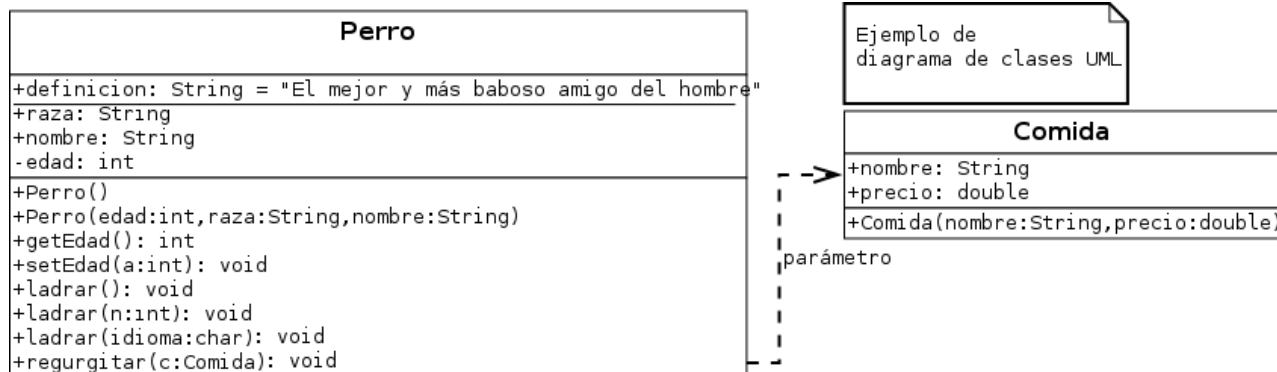
public void ladrar(char idioma) {
    switch (idioma) {
        case 'I': // Inglés
            System.out.println("BARK!!!");
            break;
        case 'F': // Francés
            System.out.println("WOF!!!");
            break;
        default:
            ladrar();
    }
}

public void regurgitar(Comida c){
    c.nombre="bolo alimenticio";
    c.precio =0;
}
}

```

Y a continuación el diagrama UML de clases:

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR		Francisco Bellas Aláez (Curro)					



Podemos ver la clase Perro completa. Cabe notar que el atributo estático *definicion* se escribe subrayado.

Para indicar que existe una dependencia entre la clases Perro y Comida se indica con una línea de trazos terminado en una flecha abierta. En este caso la dependencia es paramétrica ya que Perro depende de Comida pues lo usa como parámetro en uno de sus métodos.

Finalmente el recuadro con una esquina doblada se utiliza para realizar cualquier comentario del diagrama. Se le pueden añadir líneas con flechas para apuntar a una zona concreta del diagrama.

Herencia e Interfaces

En UML la **herencia** se identifica mediante una **flecha cerrada con línea continua**.

En UML un **Interface** se representa como una clase poniendo el **"estereotipo" «interface»** encima del nombre del a interface **y todo su contenido en cursiva**.

El estereotipo simplemente es un indicativo que se usa en clases UML para indicar el uso que se le da a esa clase. Otro uso típico es usar el estereotipo «utility» en clases con funciones estáticas como Math (son librerías de funciones).

En las clases que la implementan se escribe las funciones sobreescritas.

Se representa la **implementación** mediante una **flecha cerrada con línea discontinua**.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Veamos un ejemplo con estas características:

```

public class Animal {
    public String nombreCientifico;
    private int edad;

    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
}

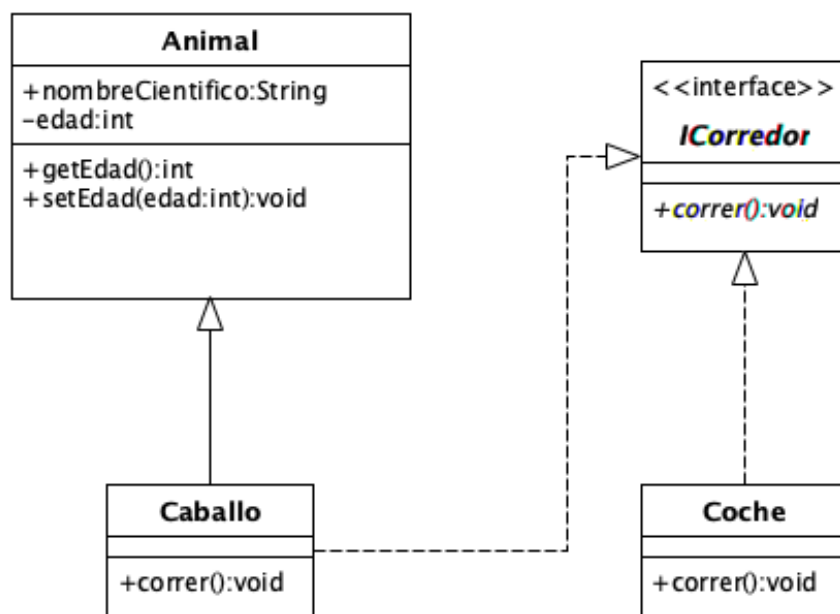
interface ICorredor {
    void correr();    //sólo la definición
}

class Coche implements ICorredor {
    public void correr(){ //código para mover las ruedas
    }
}

class Caballo extends Animal implements ICorredor {
    public void correr(){ //código para mover las patas
    }
}

```

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR	Francisco Bellas Aláez (Curro)						



Respecto a la sobreescritura llega con poner el nombre de la función en la clase hija. Se permite, aunque no es obligatorio, especificarlo de forma más clara con un estereotipo «override» antes o después del nombre de la función. P.ej:

```

+ funcion(): void «override»
ó
+ «override» funcion(): void

```

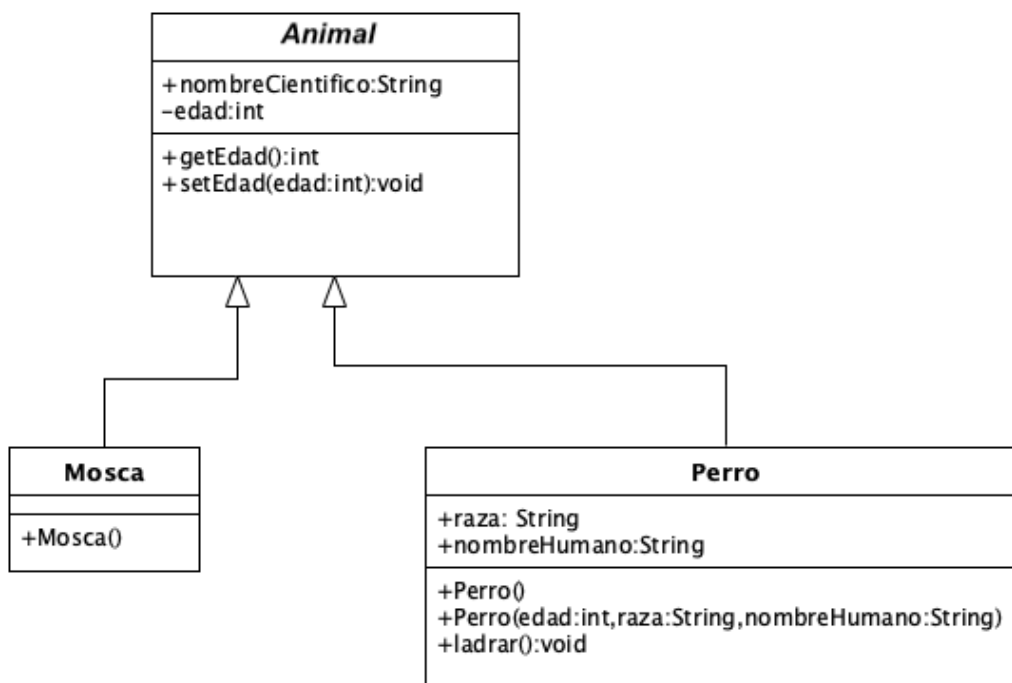
Clases Abstractas

En UML una clase abstracta se indica poniendo su **nombre en cursiva** de forma similar a cómo se hace con los interfaces (Itálica).

Si hubiera una o más funciones abstractas estas también irían en cursiva.

Vemos el ejemplo visto de la herencia de Perro y Mosca pero heredando de la clase abstracta Animal.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM
	MÓDULO	Entornos de desarrollo		
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:
	AUTOR	Francisco Bellas Aláez (Curro)		
		CURSO: 1º		



Más informaciónj sobre relaciones en diagramas de clases UML:

<https://www.gleek.io/blog/class-diagram-arrows>

COLEXIO VIVAS S.L.

RAMA:	Informática	CICLO:	DAM				
MÓDULO	Entornos de desarrollo					CURSO:	1º
PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
AUTOR	Francisco Bellas Aláez (Curro)						

Apéndice I: Pseudocódigo

Trata de expresar en lenguaje natural la lógica de un programa, es decir, su flujo de control.

La estructura que seguiremos de escritura de algoritmos será la siguiente:

Algoritmo <Nombre>

Entorno

<declaración de variables y constantes>

Inicio

<Programa>

Fin

Para escribir en pseudocódigo usaremos las expresiones matemáticas, lógicas, de relación y de asignación vistas. También las estructuras pero traducidas:

si condición entonces acciones fin si	si condición entonces acciones A sino acciones B fin si
según expresión hacer valor1: acción 1 valor2: acción 2 valor N: acción N otro: acción N+1 fin según	mientras condición hacer acciones fin mientras
hacer acciones mientras condición	para var=exp1 hasta exp2 [incremento exp3] hacer acciones fin para
repetir acciones hasta que condición	Nota: La estructura repetir no existe en Java pero sí en otros lenguajes como Basic o Pascal. Al contrario que hacer-mientras, la condición es de salida en vez de ser de repetición.

También usaremos los siguientes comandos:

Leer variable1, variable2, ... : Lee datos del usuario y los vuelca en sucesivas

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Entornos de desarrollo			CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)			

variables. También puede usarse **Pedir**.

Escribir Var_o_const1, Var_o_const1,... : Escribe en pantalla de forma sucesiva las constantes indicadas o el contenido de las variables indicadas. También puede usarse **Mostrar**.

Ejemplo:

```

Algoritmo Suma de pares
Entorno
    n, suma, i: entero

Inicio
    hacer
        Escribir "Introduce un número hasta el que desea sumar"
        Leer n
    mientras n<2
        i=2
        suma=0

        mientras i<=n
            si i%2==0 entonces
                suma=suma+i
            fin si
            i++
        fin mientras
        Escribir "El resultado es", suma
Fin

```

Existen programas como PSeInt para practicar el uso de pseudocódigo:

<http://pseint.sourceforge.net>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Apéndice II: Ejemplo documentación con tags ampliados.

Como ya se ha visto en anteriores temas, la forma genérica para crear páginas de documentación es usando el tipo de comentario para Javadoc:

```
/**
 * Comentarios Javadoc
 */
```

Este tipo de comentario puede preceder a la declaración de una Clase, un método, un constructor o un campo.

En este texto se presentan algunos tags más además de los ya vistos y un ejemplo completo de documentación. Luego como práctica un ejercicio en el que se comenta un proyecto de programación.

Como el código que genera Javadoc es HTML, precisamente se pueden usar tags de este lenguaje para cambiar el aspecto del documento generado por javadoc. Por ejemplo si escribimos:

```
/** Prueba de escritura en <b>negrita</b>.
 * Esto va en una línea <br> y esto en otra
 */
```

El resultado será algo así:

Prueba de escritura en **negrita**. Esto va en una línea
y esto en otra

Si no te funciona en el preview prueba a usar **<p>** en vez de **
**.

Como ya se explicó en un tema anterior, lo primero que se indica en el comentario es la descripción del elemento declarado. Esta descripción termina en cuanto se introduce una línea que comienza por @.

Tags más usados

@author (sólo en clases e interfaces): Autor de la clase.

@version (sólo en clases e interfaces): Versión de la clase. El formato habitual es nº de versión y fecha (p. ej. 1.26, 27/09/2008)

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

@param (sólo en métodos y constructores): Define los parámetros del método. Si hay varios se hará en el mismo orden que la declaración.

@return (sólo métodos): Indica el valor devuelto. Se debe escribir aunque sea redundante con la descripción del método. No se usa si el método devuelve void o es un constructor.

@exception (@throws es un sinónimo usado desde Javadoc 1.2): En los métodos que se lanza una excepción no controlada se explica el motivo de dicho suceso.

@see: "See Also" (Ver también): Enlaces con otras clases o elementos para los cuales existe documentación. Automáticamente aparecen como hiperenlaces.

@since: Indica desde que versión de la clase se incluye el elemento.

@deprecated: indica que el elemento ya no se usa. Se debe especificar desde que versión de la clase no está en uso y cual es el elemento substitutivo.

{@inheritDoc}: Sirve para funciones sobreescritas que hacen lo mismo que la clase padre. Simplemente se pone este tag y se aumenta la descripción. Se ve claro en este ejemplo:

```
/**
 * {@inheritDoc}
 * Añade una validación extra antes de ejecutar la lógica principal.
 */
@Override
public void procesarDatos() {
    // implementación...
}
```

{@link #Identificador}: Permite meter un hiperenlace en medio de una descripción hacia otro punto de la documentación. Por supuesto Identificador debe disponer de Javadoc si no simplemente aparece como código.

Mediante `{@link package.clase#identificador etiqueta}` se permite enlazar con un identificador que esté en otro paquete, en otra clase y sustituir el nombre completo por una etiqueta que nos interese.

Por supuesto se puede hacer por partes, por ejemplo se puede usar `clase#identificador` para referirse a otra clase del paquete actual.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

Existen más tags pero estos son los más extendidos. Puedes verlos en:
<https://www.baeldung.com/javadoc-multi-line-code>

Algunas recomendaciones de estilo a seguir

- Los métodos realizan una acción por lo que deben empezar por un verbo.
- Las palabras implicadas en el código deben ir entre tags `<code></code>`
- No abusar del tag `{@link}`. No es necesario repetir el mismo enlace ni poner todos los enlaces posibles, sólo aquellos que resulten interesantes.
- Cuando se nombra un método, salvo que se desee especificar los parámetros pasados, se deben obviar los paréntesis. Escribiríamos *visualizar* en lugar de *visualizar()*.
- No es necesario indicar lo que se está describiendo. Es decir, se deben evitar comienzos de descripción como: "Este método sirve para...", "Esta clase...", "Este campo es una matriz que..."
- Evita el mismo texto en una descripción y en los parámetros o valor devuelto. El texto debe aportar algo nuevo.

Otras normas de estilo en:

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Entornos de desarrollo					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR	Francisco Bellas Aláez (Curro)						

Ejemplo

A continuación un ejemplo de documentación de una clase y sus elementos. Se ha omitido la implementación de los métodos.

```
/**
 * Diferencia filas de columnas de una tabla. FILA=0. COLUMNA=1.
 * @author Curro
 * @version 1.0
 */
enum Tipo {
    FILA, COLUMNA
}
// Por si no lo viste en el apéndice correspondiente un enum es similar a una clase
```

```
/**
 * Gestión de matrices
 * @author Curro
 * @version 1.2
 */
public class Matriz {

    /**
     * Inicializa las propiedades ... a valores ...
     */
    public Matriz() {
        //Código del constructor
    }

    /**
     * Tabla que contiene los datos
     */
    private int[][] matriz;

    /**
     * Nombre de la tabla
     * <!-- Esto no aparece en el Javadoc -->
     * @see String
     */
    private String nombre;
```

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Entornos de desarrollo						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

```

/**
 * Establece el valor de la propiedad nombre.
 * @param nombre
 */
public void setNombre(String nombre) {
    //Código del método
}

```

```

/**
 * Devuelve el valor de la propiedad nombre.
 */
public String getNombre() {
    //Código del método
}

```

```

/**
 * Muestra el {@link #nombre} del objeto separando las
 * letras con espacios.
 * <p>Luego muestra los datos de {@link #matriz} en formato tabla
 */
public void visualizarMatriz() {
    //Código del método
}

```

```

/**
 * Muestra el {@link #nombre} del objeto separando las
 * letras con espacios.
 * <p>Luego muestra los datos de {@link #matriz} en formato tabla
 * @deprecated Desde la versión 0.8, sustituida por {@link #visualizarMatriz}
 */
public void muestraMatriz() {
    //Código del método
}

```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Entornos de desarrollo				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

/**
 * Calcula y devuelve la suma de valores de una fila o de una
 * columna de {@link #matriz la propiedad matriz}
 * @param t Indica si se sumarán los valores de una fila
 * (<code>Tipo.FILA</code>) o de una columna (<code>Tipo.COLUMNNA</code>)
 * @param indice Indica índice de fila o columna
 * @return La suma realizada
 * @throws ArrayIndexOutOfBoundsException si el parámetro
 * indice se sale de los límites de <code>matriz</code>
 * @see Tipo
 */
public int suma(Tipo t, int indice) throws ArrayIndexOutOfBoundsException
{
    //Código del método
}
}

```

Más ejemplos en:

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#examples>