Introducción a la programación práctica

Contenido

Programación con jC

Bienvenido a una serie de lecciones simples sobre la programación con jC. El objetivo de este tema es el de lograr que el lector sea capaz de arrancar con el lenguaje, sin necesidad de tener nociones previas de programación. En capítulos posteriores, se realizará una exposición más teórica a la materia.

La idea es ir descubriendo, poco a poco, pequeñas cosas sobre la programación con jC, a modo de pequeñas lecciones. jC es un lenguaje de programación estructurado, ideal para el principiante. Es capaz de soportar al principiante, así como al ya iniciado. Además, dispone de capacidades gráficas simples y de consola.

1 Instalación

La instalación de jC es muy sencilla, y aquí la veremos paso a paso, tal y como se describe en su página web[1].

1. jC necesita de Java para funcionar. Se trata de pulsar en *Descargar/Download* y ejecutar el instalador. Una vez hecho esto, sólo será necesario pulsar *Siguiente/Next* hasta terminar con la instalación.

   Es probable que ya se tenga Java instalado. En este caso, no será necesario hacer nada.

2. La instalación de **jC** también es muy sencilla. Se descarga el *instalador multiplataforma* (o el *instalador para Windows*, si se está usando dicho sistema operativo), y se ejecuta. Entonces comienza el proceso de instalación, cuyos pasos más importantes son la selección del idioma, el acuerdo de licencia, la selección del directorio de instalación, y la creación de accesos directos.

   1. Selección del idioma.

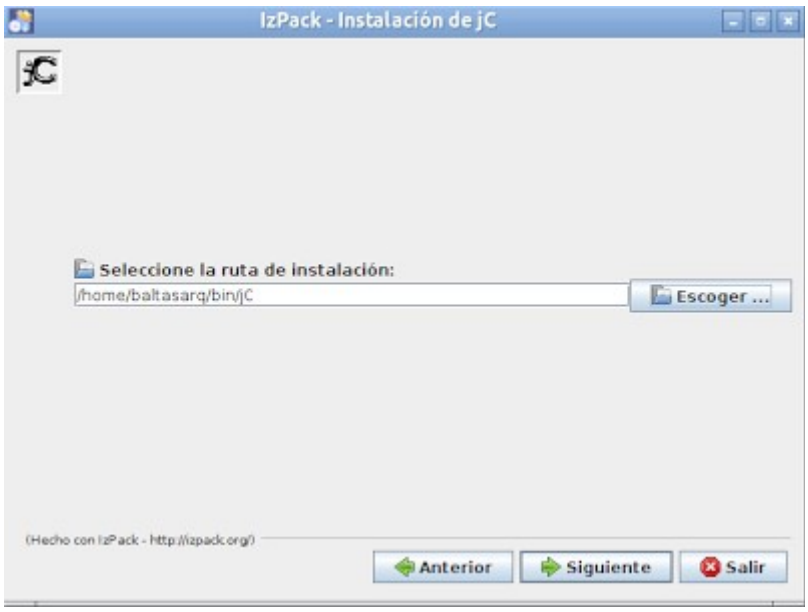
1 http://jbgarcia.webs.uvigo.es/

Aparecerá seleccionada la lengua que se está utilizando el sistema operativo. Si esto no es así, se despliega la lista y se selecciona *Español* o *Inglés*, los dos idiomas soportados.

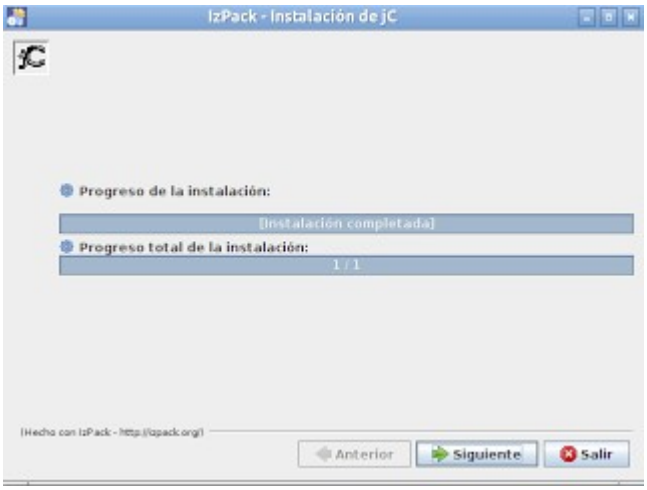
2. La pantalla de bienvenida aparece entonces. En esta pantalla no es necesario hacer nada, tan sólo pasar a la siguiente.

3. Lo mismo para la pantalla del acuerdo de licencia. Sólo es necesario seleccionar *Acepto el acuerdo*, y a continuación, pulsar en *Siguiente*.

4. A continuación, es necesario seleccionar el directorio donde se va a instalar **jC**. Es seguro dejar el directorio que sugiere la aplicación, pero no es obligatorio. Es perfectamente posible instalar **jC** en el escritorio, por poner un ejemplo extremo.



5. A continuación, se copian los archivos en el directorio de instalación.



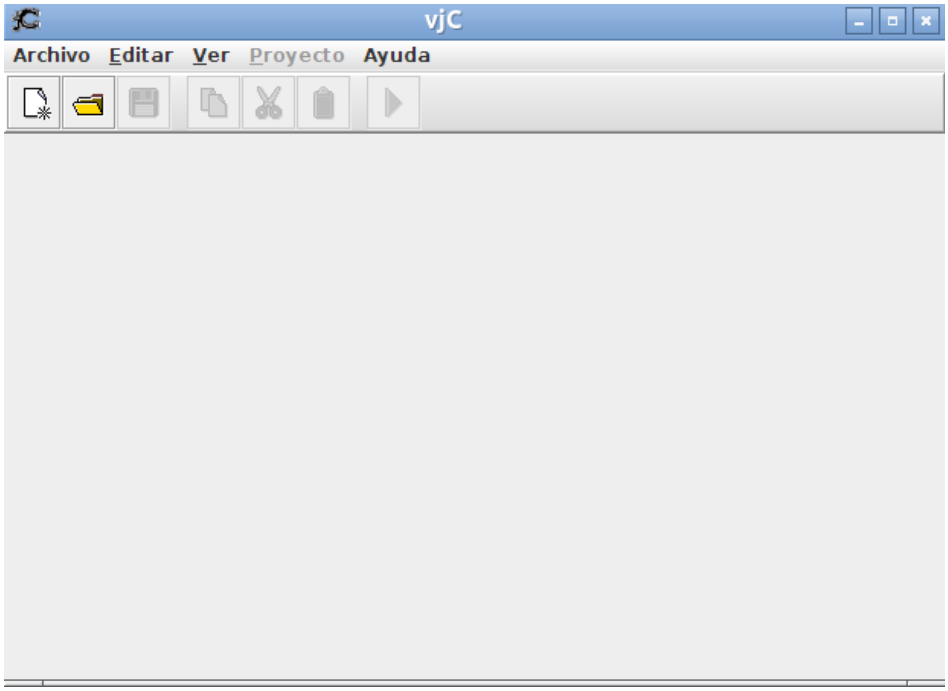
6. La instalación ya casi ha terminado. Simplemente quedan por elegir los accesos directos que se crearán. Al margen de los accesos directos que se creen, siempre será posible navegar hasta el directorio de la aplicación, y hacer doble click en *jcv.jar*.

Una vez seguidos todos estos pasos, tan solo resta ejecutar jC, bien a través del acceso directo en el menú de aplicaciones del sistema, bien navegando hasta el directorio de la aplicación, y haciendo doble click en *jcv.jar*.

Es este paradigma de programación el que se podría llamar "original", procedimental, o también conocido como "Divide y vencerás". Se trata de dividir el problema a resolver en tareas a realizar, y estas tareas en una serie de procedimientos, para finalmente encontrar el algoritmo que mejor se encuadre en ellos.

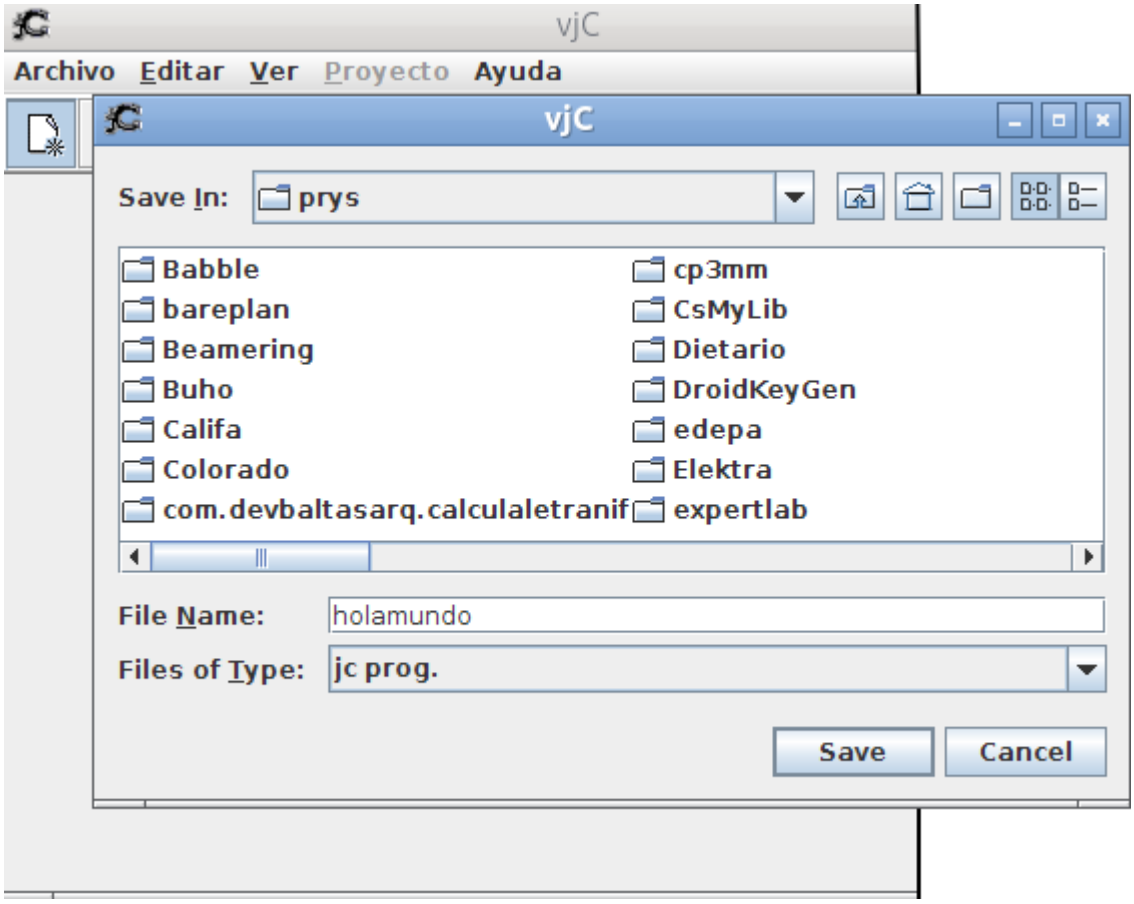
2 ¡HOLA, MUNDO!

Bueno, esto es un clásico. ¿Cómo podemos saber que tenemos bien instalado jC, y que podemos empezar a probar cosas?¿Cómo hacernos con lo más básico del lenguaje? La respuesta es un clásico: un programa que visualice por pantalla un saludo, como "¡Hola, mundo!", por ejemplo.



Al arrancar **jC**, la pantalla inicial no aparece demasiado cargada. La forma de empezar a trabajar con jC es escoger el primer icono de la barra de herramientas (el folio con un asterisco abajo a la derecha). Si se prefiere, también se puede escoger Archivo >> Nuevo >> Programa, o incluso pulsar Ctrl, y sin soltarla, pulsar la tecla 'N' (esto se suele abreviar como Ctrl+N).

Aparece entonces un cuadro de diálogo para que indiquemos un nombre de archivo. Este archivo es donde se va a guardar el sencillo programa que vamos a crear. Para comenzar, escogemos un directorio y tecleamos el nombre del programa, que será: "holamundo". Automáticamente se añadirá la extensión *.jc*. Una vez hecho esto, pulsamos en "Guardar" o "Save", y entonces podremos empezar a trabajar.



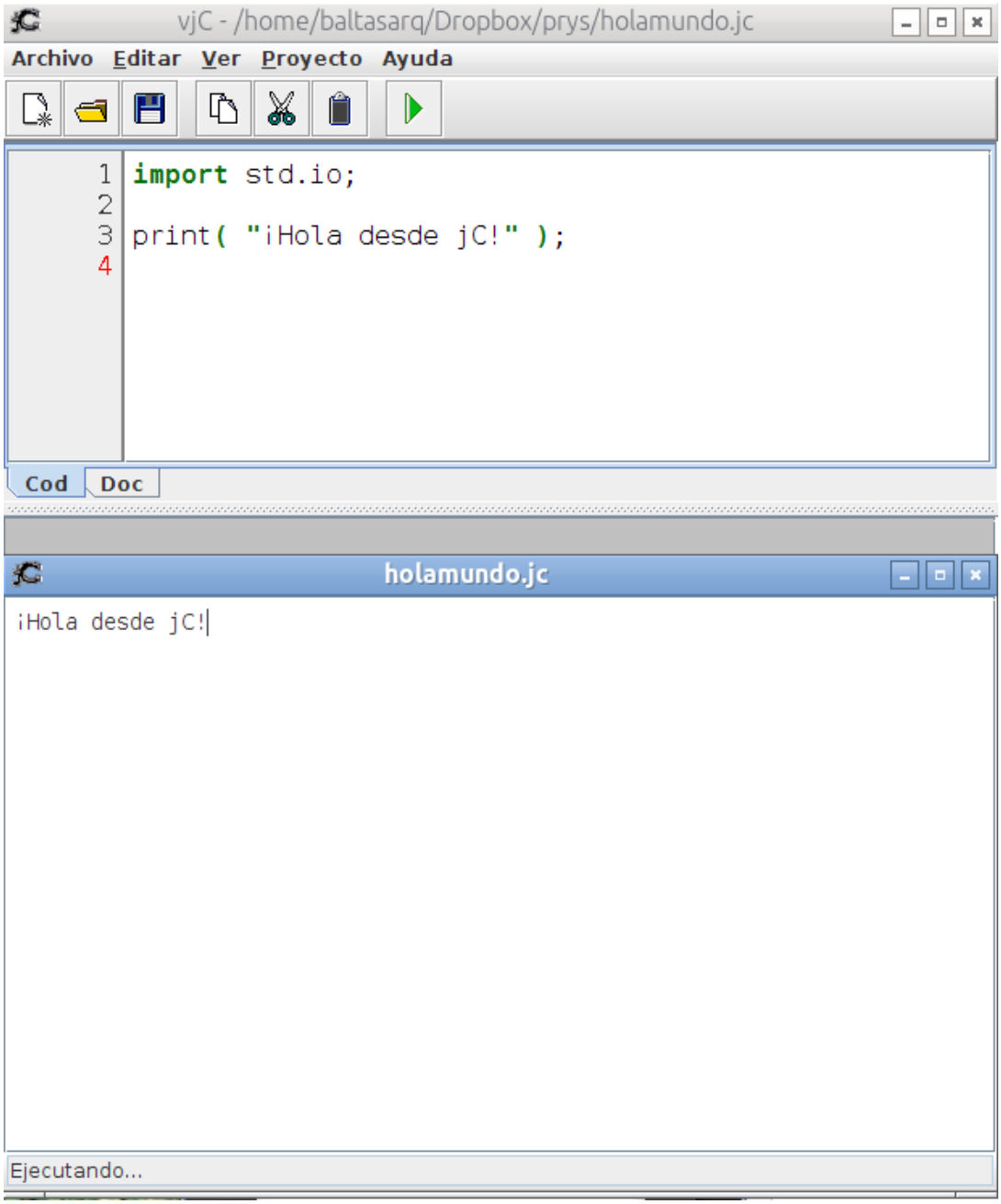
jC nos ofrece una plantilla para ir comenzando, pero lo cierto es que es muy pronto para saber qué es lo que hace todo esto. Por ahora, lo borramos todo, y dejamos sólo el primer *import*: *import std.io;*. La librería estándar **io** es necesaria para poder utilizar una función que nos permita visualizar algo por pantalla, así que con *import* se incluye.

Las funciones ofrecen funcionalidades (nunca mejor dicho) que necesitaremos frecuentemente. Alguien (en este caso, servidor), se ha preocupado de prepararlas para que estén disponibles para los programadores. Veremos en detalle más adelante qué es una función, pero por ahora dejémoslo estar. Debemos llamar a la función *print* de **io**, y para hacer eso ponemos el nombre de la función, y entre paréntesis, los datos que necesita para funcionar. En este caso, la función *print()* necesita saber qué debe

visualizar. Queremos que visualice texto, y el texto se introduce como tal, pero encerrado entre comillas. Así que finalmente, el programa queda como sigue:

```
import std.io;
print( "¡Hola desde jC!" );
```

El resultado es que esperaríamos: la frase que le hemos pasado a *print()* aparece en la pantalla. Bueno, ya sabemos cómo hacer algo tan sencillo como esto, ¡ahora hay que seguir avanzando por las lecciones!



3 SECUENCIA

Una de las primeras cuestiones a aprender, en cuanto a programación estructurada es la **secuencia**, o lo que es lo mismo, comprender que las instrucciones (sean las que sean) que se ponen unas detrás de las otras se ejecutan en el mismo orden en el que se escribieron. ¡Tan sencillo como esto! Para comenzar, utilizaremos el módulo turtle. Este módulo permite manejar una tortuga por la pantalla, de tal manera que podemos hacer dibujos sencillos, ya que la tortuga deja rastro a medida que avanza. ¿Cuál va a ser el objetivo? Nada más y nada menos que hacer un dibujo de una escalera. ¿Fácil, verdad? Para poder utilizar el módulo de la tortuga, **turtle**, debemos indicarle a **jC** que lo necesitaremos, y para eso se emplea la sentencia *import*. El módulo de la tortuga necesita del módulo de funciones gráficas (**gw**), así que será necesario importarlo también:

```
import gw;
import turtle;
```

De acuerdo, ahora sólo es necesario empezar a darle órdenes a la tortuga. Para comprobar qué funciones soporta, sólo es necesario pulsar F1 y hacer click en el módulo de la tortuga. Para hacer que la tortuga avance, es necesario emplear la función *forward()*, y tenemos que pasarle la distancia a avanzar.

```
import gw;
import turtle;
turtle.forward( 25 );
```

¡Fantástico! La tortuga ha avanzado 25 pixels, con lo cuál ha visualizado una pequeña línea recta. Quizás te preguntes dónde está la tortuga. La tortuga en sí no se ve, aunque estaría al final de lo último que has dibujado. Los gráficos de tortuga tienen su origen en el lenguaje de programación LOGO[2], que fue creado precisamente para enseñar a los principiantes a programar, con la ventaja de apreciar sus avances visualmente. Hay una línea separando los imports de la llamada a forward() porque así queda más claro. Las líneas en blanco no son significativas, podríamos dejar veinte si quisiéramos. O ninguna. Pero es importante que el programa sea fácil de leer, incluso por ti mismo, cuando lo retomes dentro de un tiempo. Así que cuanto más claro esté el código (el código o el código fuente son las instrucciones del programa), mejor. USamos la funciión *forward()*, pero... ¿qué es una función? Si en un programa logramos que se realice una tarea poniendo una instrucción después de otra, una función es un conjunto de instrucciones que ha preparado alguien, con un nombre determinado, para lograr una tarea determinado. En concreto, *forward()* es una secuencia de instrucciones que ha preparado servidor para que la gente no tenga que programar el comportamiento de la tortuga una y otra vez: sólo úsalo. Las veces que haga falta. La primera instrucción que hemos utilizado ha sido *turtle.forward(25)*. En realidad, habría sido igual de efectivo indicar exclusivamente *forward(25)*. Estrictamente, la parte *turtle.* sólo hace referencia al módulo donde está la función a continuación, *forward()*. Como no hay ambigüedad,

2 http://es.wikipedia.org/wiki/Logo_(lenguaje_de_programación)

(no estamos utilizando una función que esté creada en otro módulo con el mismo nombre), no sería necesario, como decía, añadir el prefijo **turtle**. Pero nos sirve como documentación, de forma que sabemos que es exactamente ese módulo el que queremos usar. Hagamos algo más. Si giramos la tortuga noventa grados, conseguiremos que "apunte" hacia abajo, y entonces podremos avanzar otro poco, haciendo un escalón. Para girar, tenemos las funciones *turnRight()* y *turnLeft()* del módulo de la tortuga.

```
import media.gw;
import media.turtle;

turtle.forward( 25 );
turtle.turnRight( 90 );
turtle.forward( 25 );
```

De acuerdo, las instrucciones se ejecutan una detrás de la otra: avanzamos hacia la derecha (al principio, la tortuga apunta hacia la derecha por defecto), giramos noventa grados, avanzamos de nuevo... y con eso se ha dibujado un escalón en la pantalla. Vamos por buen camino. Ahora sólo es necesario repetir lo de antes, para conseguir una escalera completa. De acuerdo, sólo será necesario copiar las instrucciones que ya tenemos ahora (menos los *imports*) y pegarlas tres veces para conseguir una escalera de tres peldaños. ¡Vaya, pues no funciona! Si lo has probado, comprobarás que no sale una escalera. Tiene sentido, una vez que completamos un peldaño, debemos hacer que la tortuga mire hacia la derecha de nuevo. Es mejor borrarlo todo y volver a empezar. Partir del listado de más arriba. Hemos aprendido que copiar y pegar, sin pensar en lo que estamos haciendo, no es una buena idea. Para colocar la tortuga apuntando hacia la derecha, debemos girarla noventa grados en sentido contrario hacia donde la giramos antes. Es decir, para tener dos escalones:

```
import media.gw;
import media.turtle;

turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );

turtle.turnLeft( 90 );
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );
```

No queda muy claro lo que estamos haciendo. Es fácil perderse viendo el código fuente. Para clarificarlo, incorporaremos comentarios dentro del código, que nos digan lo

que hace cada grupo de instrucciones. Pero claro, no podemos permitir que se compilen esos comentarios... ¡sería desastroso!. Si empezamos una línea con //, haremos que esa línea no se compile, pudiendo poner a continuación lo que queramos.

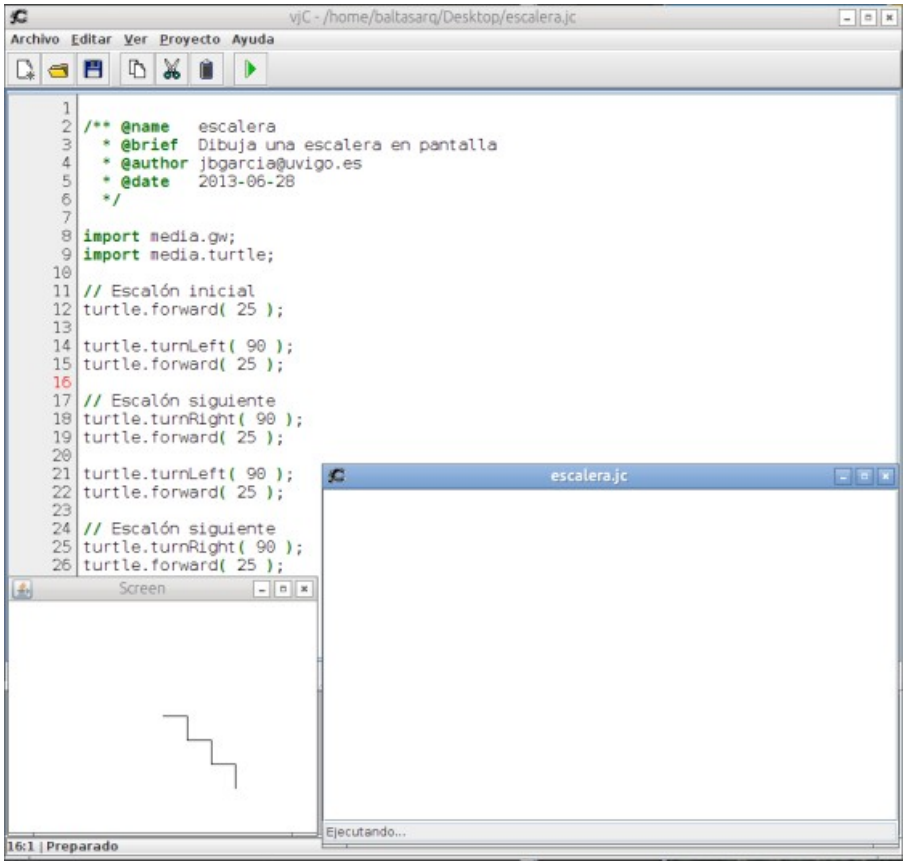
```
import media.gw;
import media.turtle;

// Escalón inicial
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );

// Escalón siguiente
turtle.turnLeft( 90 );
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );
```

De acuerdo, ahora sí. Si copiamos y pegamos las instrucciones desde el comentario de Escalón siguiente, hasta el final, podemos incorporar todos los peldaños que queramos, simplemente según el número de veces que peguemos. El código final es:

```
/** @name   escalera
  * @brief  Dibuja una escalera en pantalla
  * @author jbgarcia@uvigo.es
  * @date   2013-06-28
  */

import media.gw;
import media.turtle;

// Escalón inicial
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );
```

```
// Escalón siguiente
turtle.turnLeft( 90 );
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );

// Escalón siguiente
turtle.turnLeft( 90 );
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );
```

La primera parte es un comentario especial, es decir no es esencial para el programa. Cuando empezamos un párrafo por /*, ese párrafo no se compila. Si se empieza por /**, entonces es que se trata de un comentario de documentación. Si pulsas F4 (una vez introducido y ejecutado el programa), verás la documentación generada a partir de ese comentario.



4 Constantes

Continuamos con la escalera de la lección anterior. Finalmente, habíamos logrado crear una escalera de tres peldaños, usando el módulo de la tortuga. El código, al que llegamos sería similar al siguiente:

```
import media.gw;
import media.turtle;

// Escalón inicial
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );

// Escalón siguiente
turtle.turnLeft( 90 );
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );

// Escalón siguiente
turtle.turnLeft( 90 );
turtle.forward( 25 );

turtle.turnRight( 90 );
turtle.forward( 25 );
```

El código es simple, y está comentado. Gracias a estas dos características, podemos entender perfectamente lo que hacía el programa, pese a que han pasado ya unos días.

Si observamos el código con detenimiento, veremos que se repiten dos valores muy frecuentes: 25 y 90. El primer valor es el tamaño del peldaño (la distancia a recorrer para pintar un peldaño), y el segundo es el ángulo a girar. Este ángulo siempre es 90, gracias a las interesantes propiedades de las escaleras.

Podemos definir unas constantes para estos dos valores. Las constantes son solo nombres para ciertos valores. Esto hace que sea sencillo recordarlos, o al menos más sencillo que utilizar los valores en sí. En segundo lugar, es posible que queramos cambiar estos valores en el futuro, por ejemplo, porque queramos hacer la escalera más grande. En este momento, es complicado hacerlo, puesto que tendríamos que localizar todos los '25' en el código y cambiarlos manualmente.

Así, las constantes se pueden definir utilizando la siguiente sintaxis:

```
final def <nombre> = <valor>
```

Por ejemplo, podríamos definir el ángulo como la constante **Angulo**.

```
final def Angulo = 90;
```

Y también podríamos definir la distancia como la constante **Distancia**.

```
final def Distancia = 25;
```

Es importante acostumbrarse a poner los nombres de las constantes con la inicial en mayúscula. El objetivo es distinguirlo de otros nombres, hacer que destaque sobre el resto de alguna manera.

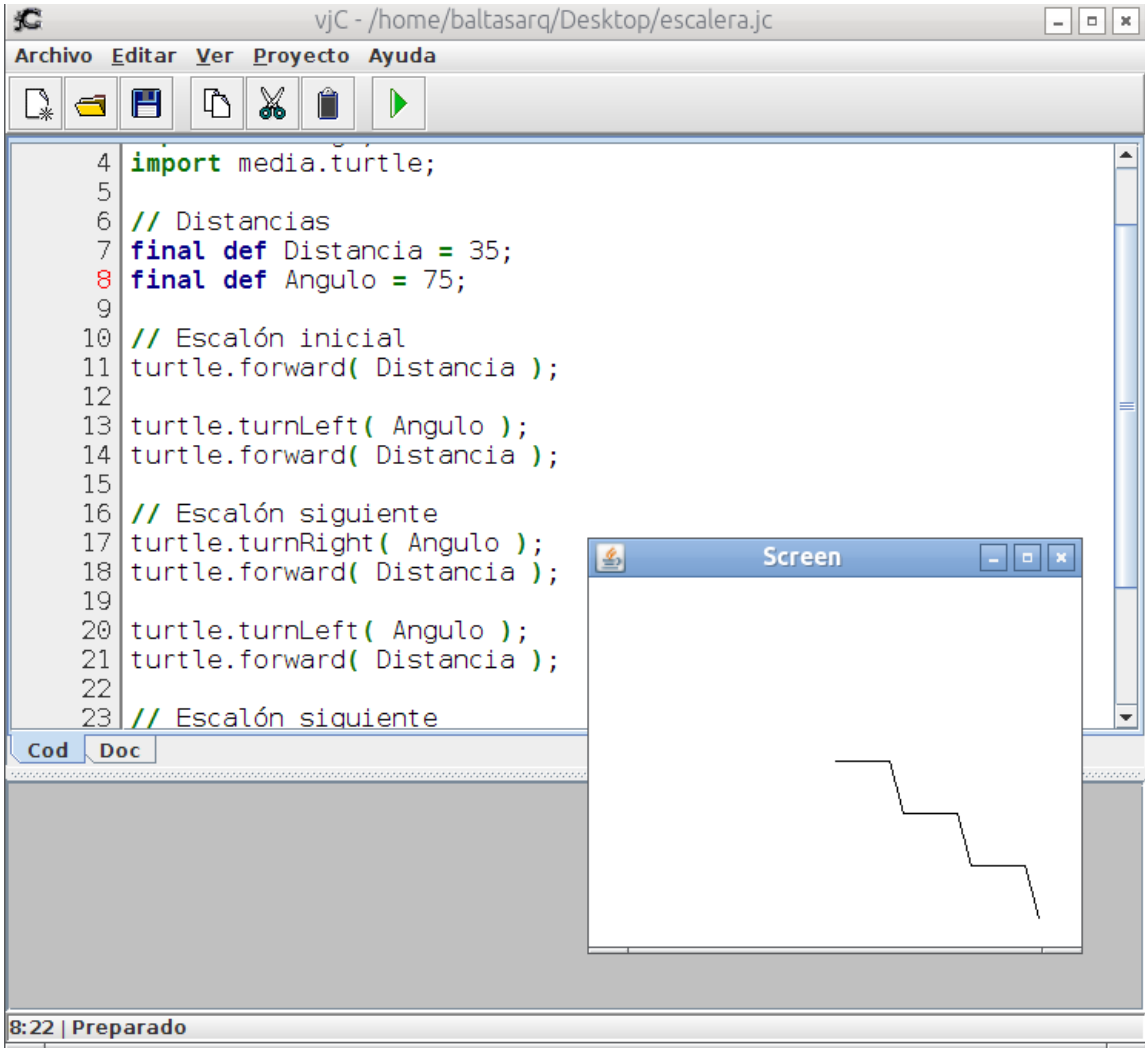
Si analizamos un poco ambas sentencias, veremos que hay dos partículas comunes: **final** y **def**. La primera partícula indica que el valor que le vamos a asignar al nombre es un valor que no va a cambiar en el futuro. La segunda partícula indica que estamos realizando una definición. Así, podríamos leer cualquiera de las dos, por ejemplo la primera, como "*defino la constante Angulo con un valor final de 90*".

Las constantes se ponen al comienzo del código, para que estén disponibles para el resto del programa. Eso sí, seguiremos manteniendo los **import**'s primero. Así, sólo resta sustituir las apariciones de los valores 25 por **Distancia**, y 90 por **Angulo**.

```
import media.gw;
import media.turtle;

// Distancias
final def Distancia = 35;
final def Angulo = 90;

// Escalón inicial
turtle.forward( Distancia );

turtle.turnRight( Angulo );
turtle.forward( Distancia );

// Escalón siguiente
turtle.turnLeft( Angulo );
turtle.forward( Distancia );

turtle.turnRight( Angulo );
turtle.forward( Distancia );
```

```
// Escalón siguiente
turtle.turnLeft( Angulo );
turtle.forward( Distancia );

turtle.turnRight( Angulo );
turtle.forward( Distancia );
```

Y efectivamente, ahora podemos modificar el valor asignado a **Distancia**, de tal manera que la escalera saldrá más grande o más pequeño. También podemos variar el ángulo, consiguiendo escaleras exóticas, como la que se muestra a continuación.



5 VARIABLES

Lo contrario de las constantes son las variables. Mientras una constante adopta un valor que se le indica en el momento de crearla, la variable (se le dé o no un valor al inicio) puede cambiar el valor contenido con el paso del tiempo.

Observemos el código del apartado anterior:

```
import media.gw;
import media.turtle;

// Distancias
final def Distancia = 35;
final def Angulo = 90;

// Escalón inicial
turtle.forward( Distancia );

turtle.turnRight( Angulo );
turtle.forward( Distancia );

// Escalón siguiente
turtle.turnLeft( Angulo );
turtle.forward( Distancia );

turtle.turnRight( Angulo );
turtle.forward( Distancia );

// Escalón siguiente
turtle.turnLeft( Angulo );
turtle.forward( Distancia );

turtle.turnRight( Angulo );
turtle.forward( Distancia );
```

Podemos modificar **Angulo** y **Distancia**, es decir, la caída desde el peldaño superior al inferior, y el tamaño del peldaño, respectivamente. Pero eso sí, hay que tener en cuenta que todos los peldaños son iguales.

El uso de variables nos permitirá, de hecho, conseguir que cada peldaño tenga un ángulo y tamaño diferente. La sintaxis para crear variables es muy parecida a la de creación de constantes, sólo tenemos que eliminar la palabra clave **final**.

```
def <variable> = <valor>;
```

Veamos el código anterior con una ligera modificación:

```
import media.gw;
import media.turtle;

// Distancias
def distancia = 35;
def angulo = 90;

// Escalón inicial
turtle.forward( distancia );

turtle.turnRight( angulo );
turtle.forward( distancia );

// Escalón siguiente
turtle.turnLeft( angulo );
turtle.forward( distancia );

turtle.turnRight( angulo );
turtle.forward( distancia );

// Escalón siguiente
turtle.turnLeft( angulo );
turtle.forward( distancia );

turtle.turnRight( angulo );
turtle.forward( distancia );
```

Las variables siempre se crean como las constantes, pero con la inicial en minúscula. Debemos intentar crear estos nombres tan pequeños como sea posible, pero manteniendo a la vez su significado. "angulo" y "distancia" son perfectamente manejables en cuanto a tamaño, y son muy significativos, es decir, describen con precisión cuál es su significado.

Cuando en un momento dado se quiera modificar una variable, sólo es necesario indicar el nombre de dicha variable, el símbolo '=' y a su derecha, el nuevo valor.

```
<variable> = <valor>;
```

Dado que los valores de las variables pueden ser modificadas en cualquier momento, es posible crear una pequeña modificación del programa anterior con valores distintos para cada peldaño de la escalera.

```
import media.gw;
import media.turtle;

// Distancias
def distancia = 35;
def angulo = 90;

// Escalón inicial
turtle.forward( distancia );

turtle.turnRight( angulo );
turtle.forward( distancia );

// Escalón siguiente
angulo = 70;
turtle.turnLeft( angulo );
turtle.forward( distancia );

turtle.turnRight( angulo );
turtle.forward( distancia );

// Escalón siguiente
angulo = 110;
turtle.turnLeft( angulo );
turtle.forward( distancia );

turtle.turnRight( angulo );
turtle.forward( distancia );
```

Gracias a esta modificación, los dos últimos peldaños son ahora un tanto estrambóticos, uno caído hacia abajo y el otro empinado hacia arriba. Es tan sólo una

muestra de lo que se puede hacer. ¡Las posibilidades son infinitas!¿Por qué no experimentar?

6 La tortuga

Hasta ahora, hemos estado utilizando la tortuga para mostrar algunos conceptos, pero en realidad no sabemos mucho de ella.

Como ya comentamos al comienzo, el módulo de la tortuga está basado en el lenguaje de programación Logo, y este lenguaje está especialmente diseñado para aprender programación. Utilizando la metáfora de la tortuga como un módulo más, podremos aprender a resolver problemas en **jC**, casi sin darnos cuenta.

Recordemos que la forma de trabajar con la tortuga es girar, avanzar, y, si se desea, retroceder. Para ello se utilizan las siguientes funciones:

1. **turn(x)**: Es necesario especificar un ángulo *x*, para que la tortuga apunte en ese ángulo de una circunferencia imaginaria a su alrededor.
2. **turnRight(x)**: Es necesario especificar un ángulo *x*, para que la tortuga gire el ángulo dado en el sentido de las agujas del reloj, con respecto al ángulo al que ya apuntaba.
3. **turnLeft(x)**: Es necesario especificar un ángulo *x*, para que la tortuga gire el ángulo dado en el sentido contrario al de las agujas del reloj, con respecto al ángulo al que ya apuntaba.
4. **forward(x)**: Avanza en el ángulo actual, una distancia *x* que es necesario especificarle.
5. **backward(x)**: Avanza en el sentido contrario al ángulo actual, una distancia *x* que es necesario especificarle.

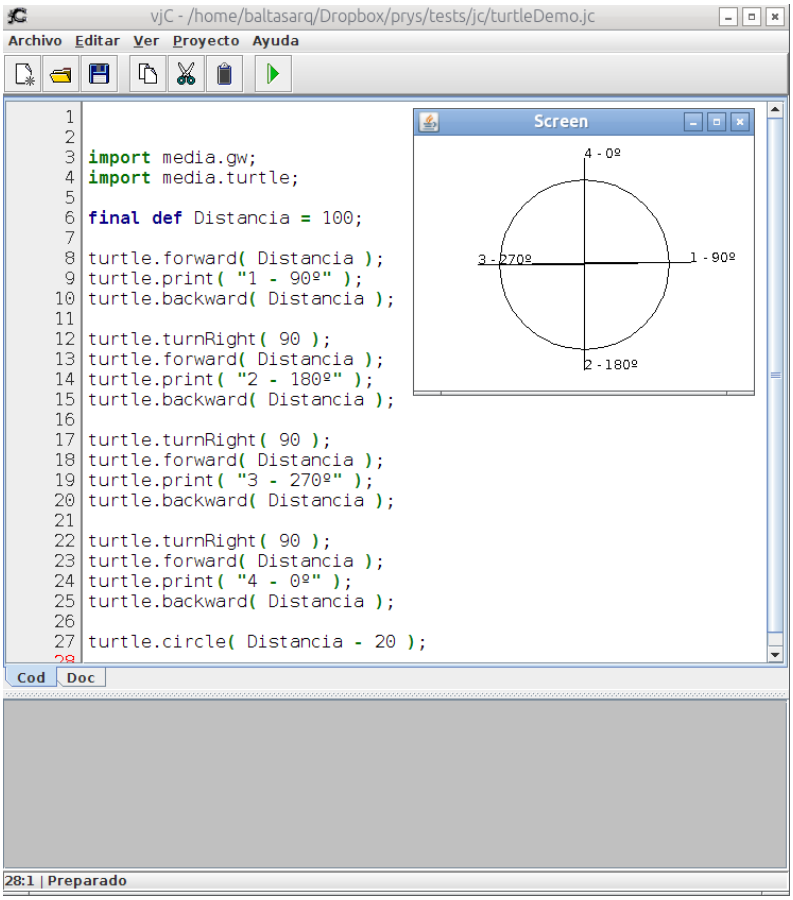
Para explorar en qué consiste esa circunferencia imaginaria, pongamos en práctica lo anterior con el siguiente programa:

```
import media.gw;
import media.turtle;

final def Distancia = 100;

turtle.forward( Distancia );
turtle.print( "1 - 90º" );
turtle.backward( Distancia );

turtle.turnRight( 90 );
turtle.forward( Distancia );
turtle.print( "2 - 180º" );
turtle.backward( Distancia );
```

```
turtle.turnRight( 90 );
turtle.forward( Distancia );
turtle.print( "3 - 270º" );
turtle.backward( Distancia );

turtle.turnRight( 90 );
turtle.forward( Distancia );
turtle.print( "4 - 0º" );
turtle.backward( Distancia );
turtle.circle( Distancia - 20 );
```

En el programa anterior, la tortuga avanza y retrocede una determinada distancia, gira 90º, vuelve a avanzar y retroceder, vuelve a girar 90º... y así hasta completar los cuatro giros y avances/retrocesos que permiten configurar la circunferencia imaginaria, pintada ahora de manera real.

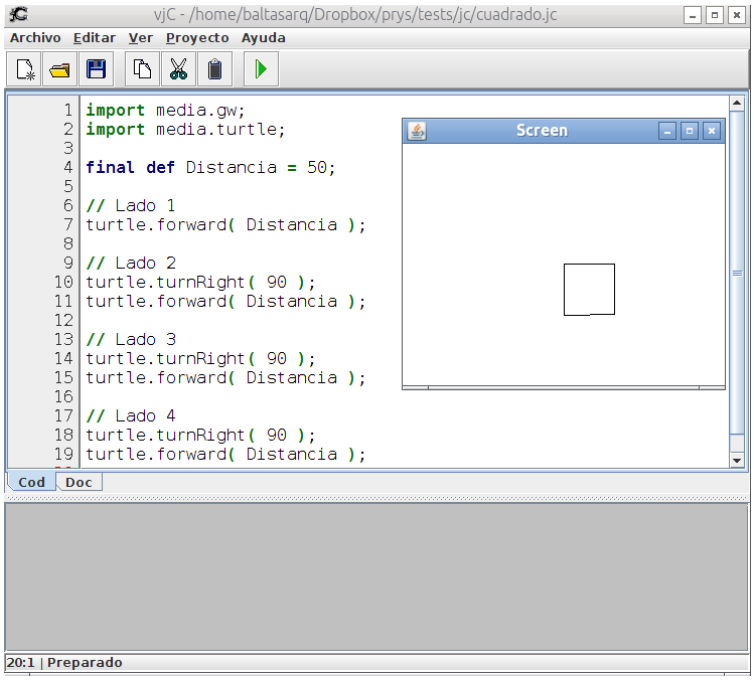
También se utiliza el método **print()** del módulo de la tortuga, que visualiza el mensaje de texto que se le pase, en la posición donde se encuentre la tortuga. La función **circle()** dibuja un círculo con el radio que se le pase, tomando como centro el punto donde se encuentre la tortuga. Es interesante notar que el valor que se le pasa a **circle()** no es un valor fijo, sino calculado a partir del valor de la **Distancia**, restándole 20.



Sabiendo esto... ¿cómo se podría dibujar un cuadrado? Se trataría de avanzar la distancia necesaria para completar un lado, girar 90º, volver a avanzar... y así hasta completar los cuatro lados. El código a continuación hace exactamente esto.

```
import media.gw;
import media.turtle;

final def Distancia = 50;

// Lado 1
turtle.forward( Distancia );

// Lado 2
turtle.turnRight( 90 );
turtle.forward( Distancia );

// Lado 3
turtle.turnRight( 90 );
turtle.forward( Distancia );

// Lado 4
turtle.turnRight( 90 );
turtle.forward( Distancia );
```

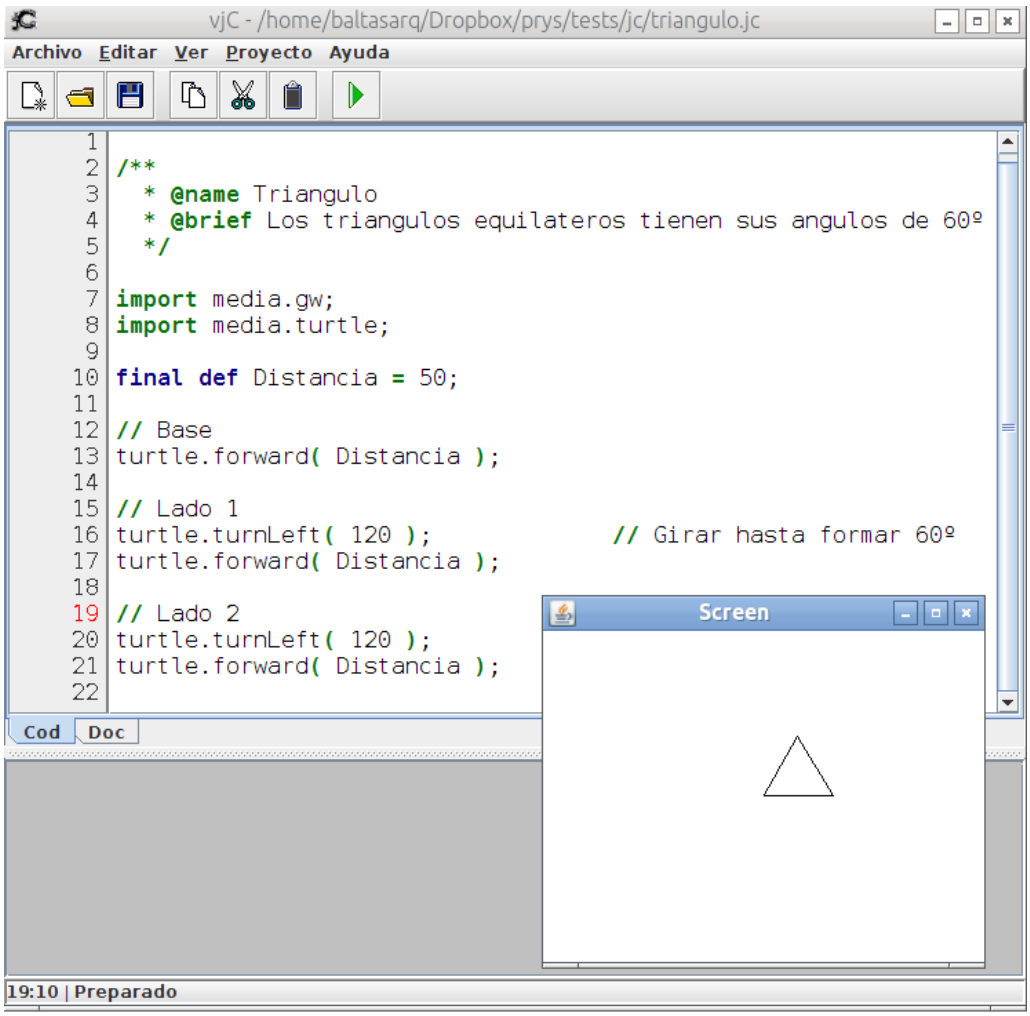
El código es muy sencillito de entender, hace exactamente lo comentado.



Si deseáramos hacer un triángulo equilátero, el programa no sería más complejo (sólo son tres lados), pero si hay que tener en cuenta que es necesario determinar el ángulo a girar, de forma que finalmente los ángulos de los lados de dicho triángulo sean 60º (un triángulo equilátero tiene todos sus ángulos de 60º. Para cualquier triángulo, la suma de los tres ángulos siempre es 180º).

Dado que la tortuga tiene que girar no 60º, sino lo suficiente para que el ángulo que quede sea de 60º, debemos pensar que si giramos, desde la base del triángulo, 60º, entonces la tortuga no estará mirando hacia dentro del triángulo, sino exactamente hacia fuera. Para alcanzar el ángulo deseado, debemos girar 60º más. En total, 120º.

```
/**
  * @name Triangulo
  * @brief Los triangulos equilateros tienen sus angulos de 60º
  */

import media.gw;
import media.turtle;

final def Distancia = 50;

// Base
turtle.forward( Distancia );

// Lado 1
turtle.turnLeft( 120 );  // Girar hasta formar 60º
turtle.forward( Distancia );

// Lado 2
turtle.turnLeft( 120 );
turtle.forward( Distancia );
```



Es posible hacer muchas más figuras geométricas, y, en realidad, dibujos de todo tipo... ¡sólo es necesario experimentar!

7 Cálculos

La forma de realizar cálculos en **jC** es muy sencilla. Las fórmulas se traducen casi tal cual, con la única salvedad de encerrar entre paréntesis las subexpresiones. Pero veámoslo.

Un ejemplo sería calcular la edad de un gato en términos humanos. Aunque no se corresponde demasiado con la realidad. se suele decir que un año la vida de un gato corresponde con siete años en la vida de un ser humano. Así, para calcular los años de un gato en términos de una vida humana, sólo sería necesario multiplicar la vida del gato, en años, por siete.

La única dificultad que nos puede surgir para traducir esta fórmula es saber cómo convertir una expresión matemática a **jC**. Para asignar un valor a una variable, no es necesario asignar ese valor ya calculado, podemos indicar al lenguaje de programación que lo compute. Las cuatro reglas básicas, matemáticamente hablando, se traducen de forma muy intuitiva al lenguaje. También debemos recordar que, cuando queremos indicar que una subexpresión se quiere calcular antes, esta se debe encerrar entre

paréntesis. A continuación, veremos una tabla resumen de operadores matemáticos disponibles.

Operador	Significado
+	Sumar: x + 10
-	Restar: x - 10
*	Multiplicar: x * 10
/	Dividir: x / 10

```
import std.io;

final def Multiplicador = 7;
def edad = 3;

def edadGatuna = edad * Multiplicador;

print( "Edad del gato: " );
println( edadGatuna );
```

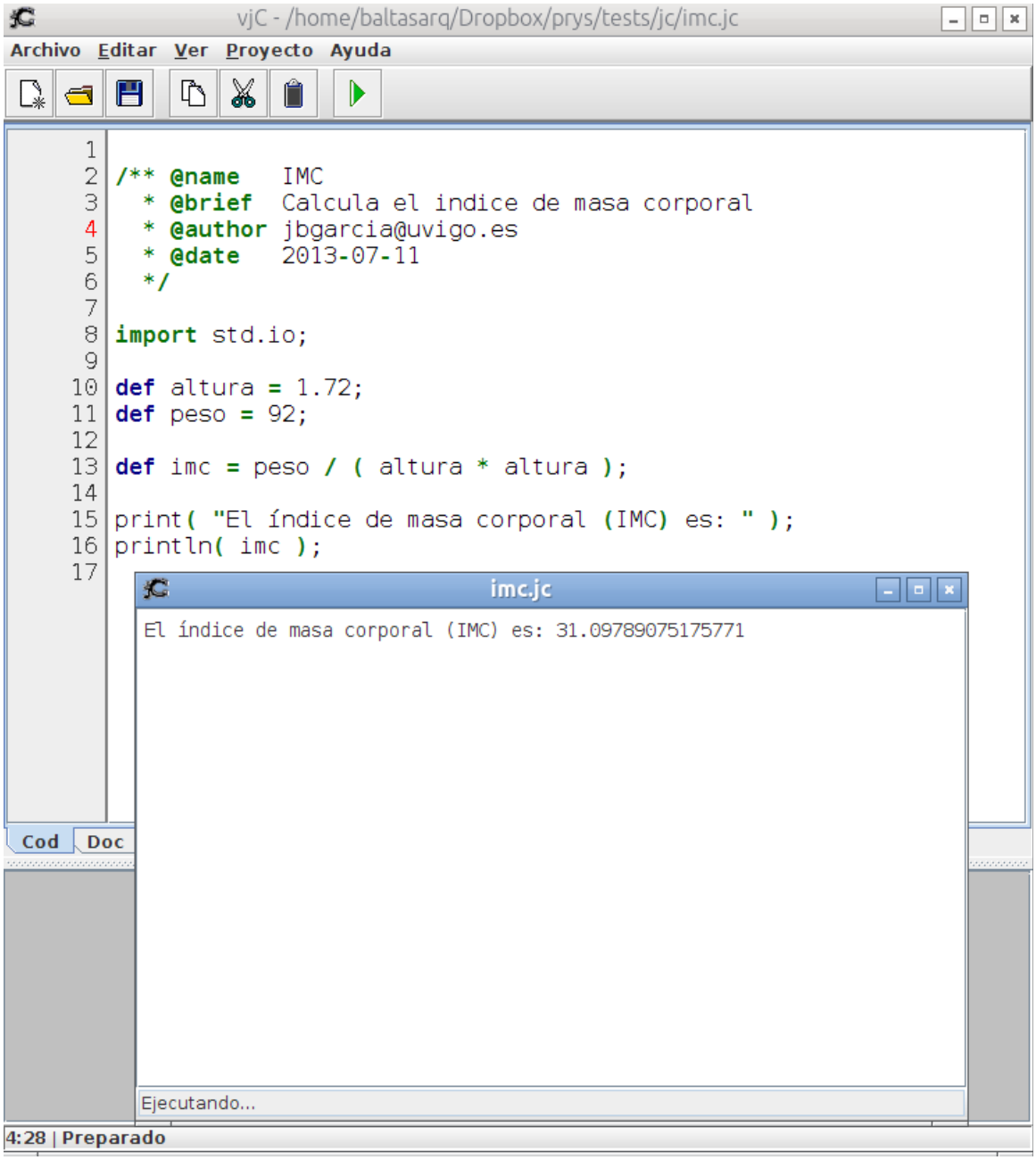
Para el siguiente ejemplo, poniendo en práctica lo que acabamos de descubrir, vamos a trabajar con una calculadora para el IMC (índice de masa corporal)[3]. Este es un valor que se calcula a partir de el peso y la altura (de hecho, es tan simple como: peso/altura^2). En cuanto a elevar al cuadrado un valor, sólo es necesario recordar que se obtiene el mismo efecto multiplicando el valor dado por sí mismo. Los números reales se expresan separando la parte entera de la decimal mediante un punto. Para lograr esto, el programa en **jC** es tan sencillo como:

```
import std.io;

final def Altura = 1.72;
final def Peso = 92;

def imc = Peso / ( Altura * Altura );

print( "IMC = " );
println( imc );
```

3 http://es.wikipedia.org/wiki/Índice_de_masa_corporal

Al finalizar el programa, la variable *imc* contiene un valor que, aproximadamente, si está por debajo de 20, indica desnutrición, entre 20 y 25 indica normalidad, entre 25 y 30 indica sobrepeso, entre 30 y 35 obsesidad leve, entre 35 y 40 obesidad moderada, y a partir de 40, obesidad mórbida.

En este ejemplo se está utilizando el módulo **io** (entrada/salida) de la librería estándar, que define las funciones, ya preparadas para nuestro uso: **print()**, que imprime cualquier valor, y **println()**, que hace lo mismo, además de un salto de línea a continuación. Aunque podríamos escribir, perfectamente, **io.print()** y similar, las funciones del módulo de entrada y salida son tan comunes que no resulta necesario. Recuérdese, además, que sólo es estrictamente necesario cuando se producen colisiones de nombres, es decir, una función existe con el mismo nombre en más de uno de los módulos que se están usando.



8 Decisión

Utilizando programación estructurada (la que soporta **jC** y estamos estudiando aquí), existen tres conceptos que es necesario conocer. **Secuencia**, **Decisión**, y **Repetición**.

La primera ya la hemos visto claramente, se trata tan solo de que cada instrucción se ejecuta una detrás de la otra, en el mismo orden en el que están escritas en el programa.

La estructura de decisión consiste en que, según se cumpla una condición o no, la ejecución continuará por una rama u otra. Para especificar la condición, y las ramas que se puedan ejecutar, **jC**, como muchos otros lenguajes de programación, utiliza la instrucción *if*.

```
if ( <condicion> )
{
    instruccion1;
    instruccion2;
    instruccion3;
    ...
    instruccionn;
} [ else {
    instruccion1;
    instruccion2;
    instruccion3;
    ...
    instruccionn;
} ]
```

La instrucción *if* puede llegar a definir dos bloques de instrucciones: el que se sitúa justo detrás de *if*, y el que aparece a continuación de éste, detrás de la instrucción *else*. Este segundo bloque de instrucciones es opcional, no es obligatorio (por eso aparece entre corchetes).

De acuerdo, pero... ¿cómo se especifica la condición? En realidad, es muy sencillo: se utilizan los operadores < (menor que), > (mayor que), == (igual a), != (diferente de), <= (menor o igual que) y >= (mayor o igual que). Estos operadores son binarios, es decir, a la izquierda y a la derecha de ellos se acompañan sendos valores. Dado que no tiene mucho sentido comparar dos literales (por ejemplo, 2 < 5 siempre supondrá el mismo resultado), uno de los dos valores debe ser una variable, o al menos, una constante. Todo esto se resume en la siguiente tabla, utilizando una hipotética variable *x*:

Operador	**Significado**
<	Menor que: x < 10
>	Mayor que: x > 10
==	Igual a: x == 10
!=	Diferente de: x != 10
<=	Menor o igual que: x <= 10
>=	Mayor o igual que: x >= 10

Retomando el ejemplo anterior sobre el IMC, sabemos que un índice de masa corporal menor que 20 implica desnutrición, 20 a 25 es normal, 25 a 30 es sobrepreso, 30 a 35 es obesidad leve, 30 a 35 obesidad moderada, y 35 a 40 obesidad mórbida. Así, resumiendo en una tabla:

IMC	**Significado**
imc < 20	Desnutrición
20 <= imc <= 25	Normal
25 < imc <= 30	Sobrepeso
30 < imc <= 35	Obesidad leve
35 < imc <= 40	Obesidad moderada
imc > 40	Obesidad mórbida

El código anterior queda como sigue:

```
import std.io;

final def Altura = 1.72;
final def Peso = 92;

def imc = Peso / ( Altura * Altura );

print( "IMC = " );
println( imc );
```

Podemos pensar fácilmente cómo complementar la información del índice desnudo, con la información de la tabla anterior. La primera es muy sencilla.

```
if ( imc < 20 ) {
    println( "Desnutrición." );
}
```

Así mismo, también la última es muy sencilla.

```
if ( imc > 40 ) {
    println( "Obesidad mórbida." );
}
```

Y así sucesivamente. El código completo es el siguiente:

```
/** @name   IMC
  * @brief  Calcula el indice de masa corporal
  * @author jbgarcia@uvigo.es
  * @date   2013-07-11
  */

import std.io;

final def Altura = 1.72;
final def Peso = 92;

def imc = Peso / ( Altura * Altura );

print( "El índice de masa corporal (IMC) es: " );
println( imc );

if ( imc < 20 )
{
 print( "Estás desnutrido!" );
}

if ( imc > 25 ) {
 print( "Tienes sobrepreso." );
}

if ( imc > 30 ) {
 print( "Tienes obesidad leve." );
}

if ( imc > 35 ) {
 print( "Tienes obesidad moderada." );
}

if ( imc > 40 ) {
 print( "Tienes obesidad mórbida." );
}
```

El problema es que para los valores que se le han dado al programa, sucede que se producen dos salidas:

```
El índice de masa corporal (IMC) es: 31.09789075175771
Tienes sobrepreso. Tienes obesidad leve.
```

Efectivamente, no se pueden colocar los **if**'s de manera independiente. Además, estamos comprobando sólo una condición en cada momento. ¡Y ni siquiera podemos indicar que la persona está normal!

Así, al menos parte del problema consiste en las tres condiciones complejas de antes... no podemos poner *20 < imc < 25* dentro de la condición de un **if**. La única forma de conseguir algo parecido, es subdividir la condición en dos subcondiciones más pequeñas: *imc > 20* **y** *imc < 25*. La cuestión es: ¿cómo representar esa 'y' que intuitivamente hemos colocado entre ambas condiciones? Pues podemos utilizar juntores lógicos, representados por &&, || y !.

Juntor lógico	**Significado**
&&	AND (y). **Ambas** condiciones deben cumplirse. Normal: `imc >= 20 && imc <= 25`
||	OR (o). **Alguna** de las dos condiciones debe cumplirse. Necesita reajuste: `imc < 20 || imc > 25`
!	NOT (no): La condición **no** debe cumplirse. Desnutrición: `!( imc > 20 )`

A la vista de estos descubrimientos, podemos decir, para una persona normal:

```
if ( imc >= 20 && imc <= 25 ) {
    println( "Es un IMC normal." );
}
```

Para una persona con sobrepreso:

```
if ( imc > 25 && imc <= 30 ) {
    println( "El IMC indica sobrepreso." );
}
```

Las condiciones complejas es mucho más conveniente colocarlas de tal manera que cada subcondición ocupe una sola línea. De esta manera, es mucho más sencillo leerlo.

```
if ( imc >= 20
  && imc <= 25 )
{
    println( "Es un IMC normal." );
}

if ( imc > 25
  && imc <= 30 )
{
    println( "El IMC indica sobrepreso." );
}
```

Así, el código completo queda como sigue:

```
/** @name   IMC
  * @brief  Calcula el indice de masa corporal
  * @author jbgarcia@uvigo.es
  * @date   2013-07-11
  */

import std.io;

def altura = 1.72;
def peso = 92;

def imc = peso / ( altura * altura );

print( "El índice de masa corporal (IMC) es: " );
println( imc );

if ( imc < 20 )
{
 println( "Estás desnutrido!" );
}

if ( imc >= 20
  && imc <= 25)
{
 println( "Normal." );
```

```
}

if ( imc > 25
  && imc <= 30 )
{
 println( "Tienes sobrepeso." );
}

if ( imc > 30
  && imc <= 35 )
{
 println( "Tienes obesidad leve." );
}

if ( imc > 35
  && imc <= 40 )
{
 println( "Tienes obesidad moderada." );
}

if ( imc > 40 ) {
 println( "Tienes obesidad mórbida." );
}
```

Ahora todo funciona bien. Sólo queda una cuestión, y es que todas las condiciones se están comprobando en todos los casos. Por ejemplo, si la persona tiene un IMC menor que 20, ya no tiene sentido comprobar si está entre 20 y 25, o 25 y 30, etc.

Para solucionar esto, se puede emplear la posibilidad **else** de una estructura **if**.

```
if ( imc < 20 ) {
 println( "Estás desnutrido!" );
} else {

 if ( imc >= 20
   && imc <= 25)
 {
  println( "Normal." );
 } else {

  if ( imc > 25
    && imc <= 30 )
  {
   println( "Tienes sobrepeso." );
  } else {

   if ( imc > 30
     && imc <= 35 )
   {
    println( "Tienes obesidad leve." );
   } else {

    if ( imc > 35
      && imc <= 40 )
    {
     println( "Tienes obesidad moderada." );
    } else {
     println( "Tienes obesidad mórbida." );
    }
   }
  }
 }
}
```

De esta manera, en cuanto se opta por una rama de la decisión, se descartan todas las demás, puesto que en caso de entrar en el primer bloque de código (cuando la condición es verdadera), nunca se entra en el segundo bloque de código (y viceversa). Otra consecuencia es que la última condición ya no es necesario comprobarla, puesto que, cuando se llega al final de la secuencia de decisiones, es imposible que *imc* no sea mayor que 40.

El código finalmente queda como sigue:

```
/** @name   IMC
  * @brief  Calcula el indice de masa corporal
  * @author jbgarcia@uvigo.es
  * @date   2013-07-11
  */

import std.io;

def altura = 1.72;
def peso = 92;

def imc = peso / ( altura * altura );

print( "El índice de masa corporal (IMC) es: " );
println( imc );

if ( imc < 20 )
{
 println( "Estás desnutrido!" );
} else {

 if ( imc >= 20
   && imc <= 25)
 {
  println( "Normal." );
 } else {

  if ( imc > 25
    && imc <= 30 )
  {
   println( "Tienes sobrepeso." );
```

```
  } else {

   if ( imc > 30
     && imc <= 35 )
   {
    println( "Tienes obesidad leve." );
   } else {

    if ( imc > 35
      && imc <= 40 )
    {
     println( "Tienes obesidad moderada." );
    } else {
     println( "Tienes obesidad mórbida." );
    }
   }
  }
 }
}
```

9 Repetición

La tercera estructura que veremos en cuanto a los lenguajes procedimentales (como **jC**), después de la secuencia (las instrucciones se ejecutan una detrás de otra, en el orden en el que fueron escritas), es la repetición (también llamada iteración).

La estructura de repetición simplemente significa que un conjunto de instrucciones se repite un número determinado de veces. Para ello, en **jC** una de las posibles instrucciones a utilizar es *while*.

```
while( <condición> ) {
    instruccion1
    instruccion2
    instruccion3
    ...
    instruccionn
}
```

La estructura más arriba se denomina bucle, porque "da vueltas" (es decir, itera o repite), mientras se cumpla una condición (la que se especifica en el paréntesis del *while*). Cuando deja de cumplirse esa condición, termina.

La secuencia de ejecución de las partes que aparecen en el código más arriba son las siguientes:

```
repetir:
    if( <condición> ) {
        instruccion1
        instruccion2
        instruccion3
        ...
        instruccionn
    }
```

Es decir, primero (1) se comprueba la condición, y si es falsa, se continúa con la primera instrucción después del *while*; en cambio, si es cierta, (2) se ejecutan las instrucciones que aparecen dentro del cuerpo del *while*. Entonces se vuelve al paso 1.

Por ejemplo, el siguiente programa cuenta del 1 al 5.

```
import std.io;

def i = 0;

while ( i < 5 ) {
    println( i + 1 );

    i = i + 1;
}
```

Este código se puede pensar también así:

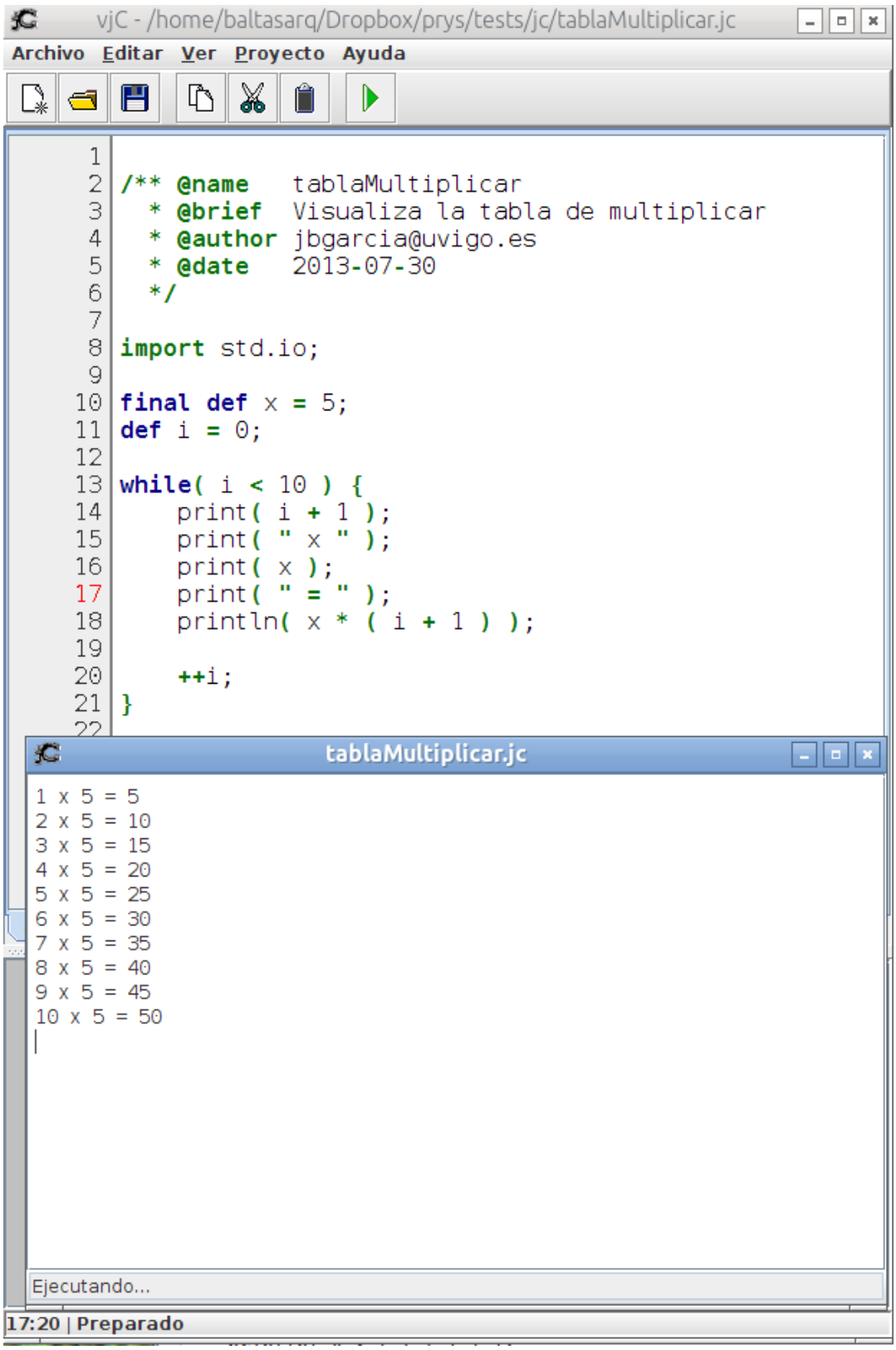
```
def i = 0;

repetir:
    if( i < 5 ) {
        println( i + 1 );

        i = i + 1;
    }
```

La instrucción *i = i + 1;* se lee como "asignarle a *i* el resultado de sumarle uno a *i*". Por ejemplo, si el valor de *i* es uno, entonces primero se ejecuta lo que está a la derecha del =, *i + 1*, y esa expresión devuelve 2. Finalmente, se le asigna 2 a *i*, pues la variable *i* es el destino de la asignación, lo que está a la izquierda del operador de asignación ('='). Como resultado, el valor de *i* es ahora 2, y se dice que se ha incrementado. Esta instrucción de incremento se puede abreviar como ++*i*, o como *i*++, no habiendo diferencia entre ellas.

```
import std.io;

def i = 0;

while ( i < 5 ) {
    println( i + 1 );

    ++i;
}
```

Para terminar, como ejemplo, podemos crear un programa que visualice una tabla de multiplicar. Para ello, es necesario ir multiplicando un número dado por los números del 1 al 10. Dada la descripción del problema, podemos pensar fácilmente en la necesidad de un bucle, que cuente desde el 1 hasta el 10.

```
import std.io;

def i = 0;
final def x = 5;

while ( i < 10 ) {
    print( i + 1 );
    print( " x " );
    print( x );
    print( " = " );
    println( x * ( i + 1 ) );

    ++i;
}
```


vjC - /home/baltasarq/Dropbox/prys/tests/jc/tablaMultiplicar.jc
Archivo Editar Ver Proyecto Ayuda
/** @name tablaMultiplicar
* @brief Visualiza la tabla de multiplicar
* @author jbgarcia@uvigo.es
* @date 2013-07-30
*/
import std.io;
final def x = 5;
def i = 0;
while(i < 10) {
print(i + 1);
print(" x ");
print(x);
print(" = ");
println(x * (i + 1));
++i;
}
tablaMultiplicar.jc
1 x 5 = 5
2 x 5 = 10
3 x 5 = 15
4 x 5 = 20
5 x 5 = 25
6 x 5 = 30
7 x 5 = 35
8 x 5 = 40
9 x 5 = 45
10 x 5 = 50
Ejecutando...
17:20 | Preparado

9.1 FOR

Una instrucción de repetición que se utiliza mucho es *for*. Esta instrucción es muy parecida a *while*, pero con una salvedad, y es que la variable que controla el bucle se puede crear, comprobar e incrementar dentro del mismo *for*.

```
for(<definición>; <condición>; <incremento>) {
    instruccion1
    instruccion2
    instruccion3
    ...
    instruccionn
}
```

La instruccón *for* repite las instrucciones en el cuerpo mientras se cumple una condición (la que se especifica en la parte intermedia del *for*). Cuando deja de cumplirse esa condición, termina.

La secuencia de ejecución de las partes que aparecen en el código más arriba son las siguientes:

```
<definición>;

while( <condición> ) {
    instruccion1
    instruccion2
    instruccion3
    ...
    instruccionn

    <incremento>
}
```

Es decir, primero (1) se realiza la definición, después (2) se comprueba la condición, y si es falsa, se continúa con la primera instrucción después del *for*; en cambio, si es cierta, (3) se ejecutan las instrucciones que aparecen dentro del *for*. Justo después, (4) el incremento, y sólo entonces se vuelve al paso 2.

Por ejemplo, el siguiente programa cuenta del 1 al 5.

```
import std.io;

for (def i = 0; i < 5; ++i) {
    println( i + 1 );
}
```

Este código se puede escribir también así:

```
def i = 0;

while( i < 5 ) {
    println( i + 1 );

    ++i;
}
```

Finalmente, el ejemplo de la tabla de multiplicar quedaría así:

```
/** @name   tablaMultiplicar
  * @brief  Visualiza la tabla de multiplicar
  * @author jbgarcia@uvigo.es
  * @date   2013-07-30
  */

import std.io;

final def x = 5;

for(def i = 0; i < 10; ++i) {
    print( i + 1 );
    print( " x " );
    print( x );
    print( " = " );
    println( x * ( i + 1 ) );
}
```

Obteniendo exactamente el mismo resultado que en la anterior entrada.

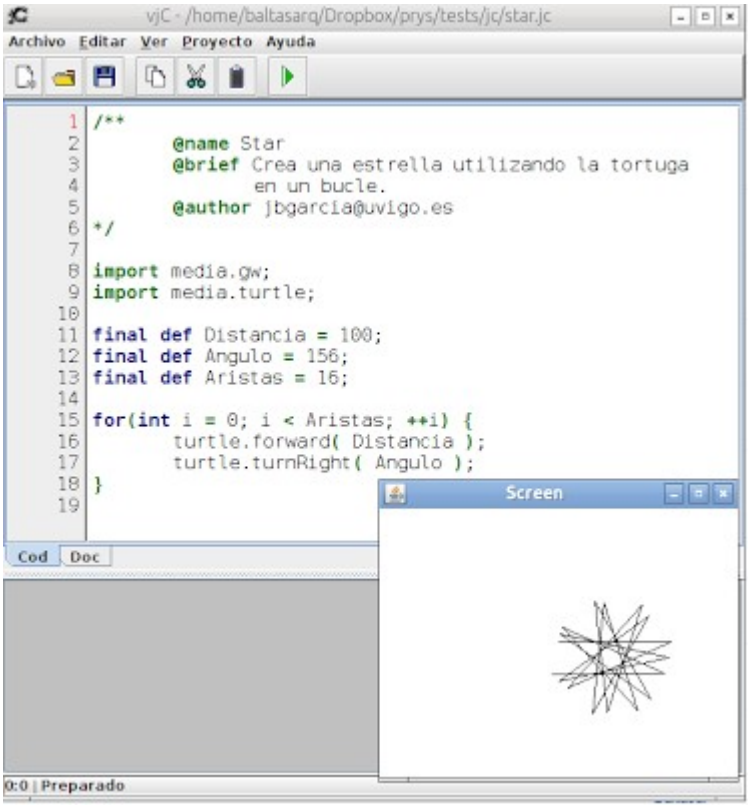
Volvamos al módulo de la tortuga. Por ejemplo, dibujar una estrella, en la que varias líneas se entrecruzan otras tantas veces, sería muy trabajoso de no conocer, o no poder emplear, la posibilidad de bucles.

Buscando el programa en logo que dibuja una estrella, es posible adaptar ese programa fácilmente a jC y su módulo de la tortuga:

```
/**
 @name Star
 @brief Crea una estrella utilizando la tortuga
  en un bucle.
 @author jbgarcia@uvigo.es
*/

import media.gw;
import media.turtle;

final def Distancia = 100;
final def Angulo = 160;
final def Aristas = 9;

for(def i = 0; i < Aristas; ++i) {
 turtle.forward( Distancia );
 turtle.turnRight( Angulo );
}
```

¡Es el momento de experimentar! Diferentes aristas, diferentes ángulos y diferentes distancias (el tamaño de cada arista) nos proporcionarán una estrella un poco diferente de cada vez (si se cambia mucho el ángulo, lo que saldrá no será una estrella: de hecho se puede obtener un cuadrado o un triángulo fácilmente.)

A continuación, se muestra un ejemplo variando un tanto las constantes dadas:



El hecho de haber avanzado hasta abarcar decisión y repetición nos permite hacer programas cada vez más complicados.

El ejemplo esta vez consistirá en dibujar polígonos regulares de cualquier número de lados. En realidad, es muy sencillo si pensamos en que la circunferencia tiene 360º, y que tenemos que dividir esos grados por el número de lados para saber qué ángulo debemos escoger para cada conjunción de aristas... ¡un polígono regular se circunscribe en una circunferencia!

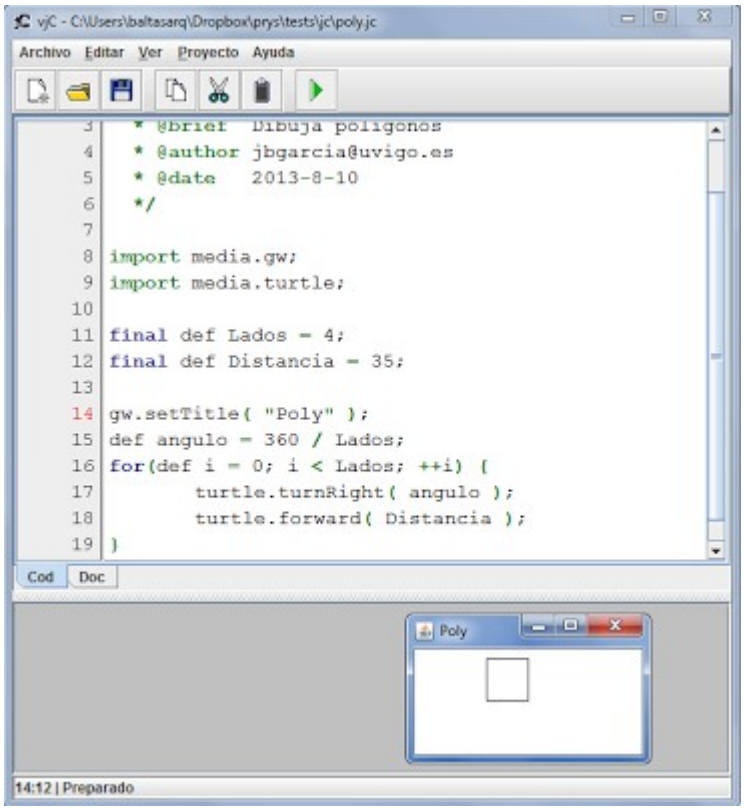
Así el algoritmo sería, en pseudocódigo:

```
i <- 0
Lados <- 4
Distancia <- 25

mientras i < Lados:
    gira( 360 / Lados )
    dibuja( Distancia )
    i <- i + 1
```

El pseudocódigo permite diseñar una aplicación, en un lenguaje natural. Se trata de una notación muy libre, que simplemente nos permita esbozar qué es lo que es necesario hacer para completar el problema.

¡Es el momento de empezar a programar! Lanza **jC** desde el menú de programas y pulsa en nuevo... escoge un directorio y un nombre para el programa. Escribe, o copia y pega lo que viene a continuación, y... ¡ya está!

```
/** @name   Poly
  * @brief  Dibuja polígonos
  * @author jbgarcia@uvigo.es
  * @date   2013-8-10
  */

import media.gw;
import media.turtle;

final def Lados = 4;
final def Distancia = 35;

def angulo = 360 / Lados;

gw.setTitle( "Poly" );
for(def i = 0; i < Lados; ++i) {
 turtle.turnRight( angulo );
 turtle.forward( Distancia );
}
```



10 Tipos para variables y constantes

Hasta ahora, cuando queríamos definir variables y constantes, lo hacíamos así:

```
final def Lados = 4;     // Constante
def distancia = 2;       // Variable
```

No es que no se pueda seguir haciendo así (es perfectamente válido, e incluso recomendado), pero es mejor que vayamos teniendo en cuenta qué está sucediendo "por debajo".

Cuando **jC** detecta la palabra reservada *def*, lo que hace es fijarse en lo que aparece después del símbolo igual, la asignación, para saber qué tipo debe asignarlo a la variable. Por ejemplo, en el caso de arriba, tanto a **Lados** como a *distancia* se le asigna el tipo *número entero* o **int**.

Si nos planteamos el siguiente caso:

```
final def PI = 3.1416;
final def CR = '\n';
```

En este caso, jC deduce que el tipo para **PI** es *número flotante*, que se denomina **double** (por *doble precisión*). El tipo para CR es un carácter, o **char**. La tabla completa de tipos disponibles es la siguiente:

Tipo	Explicación
boolean	Sólo puede almacenar dos valores: **true**(o verdadero) or **false** (falso). Además, **true** y **false** son palabras reservadas en el lenguaje (no se pueden crear variables con esos nombres, por ejemplo). Una condición, como por ejemplo, *i > 5*, se evalúa a **true** o **false**.
char	Usado para representar caracteres, según la norma UTF-16[4].
int	Numeros enteros (positivos, el cero, y negativos). Es capaz de representar números de -2 mil millones a +2mil millones[5].
double	Números reales[6] o en coma flotante. Soportan un rango muy grande de vaolres, aunque precisamente debido a temas de precisión, para realizar comparaciones numéricas entre números enteros, es mejor utilizar **int**.

Las definiciones que aparecen más arriba se pueden reescribir así:

```
final int Lados = 4;    // Constante
int distancia = 2;      // Variable
final double PI = 3.1416;
final double CR = '\n';
```

4 http://es.wikipedia.org/wiki/UTF-16
5 http://es.wikipedia.org/wiki/Entero_(tipo_de_dato)
6 http://es.wikipedia.org/wiki/Tipo_de_dato_real

¡Es el momento de experimentar! En cualquier lugar donde se escribe **def**, se puede sustituir por su tipo, que ahora ya sabemos cuáles son (al menos los tipos básicos, pronto veremos los tipos complejos). Un ejemplo:

```
/** @name   Poly
  * @brief  Dibuja polígonos
  * @author jbgarcia@uvigo.es
  * @date   2013-8-10
  */

import media.gw;
import media.turtle;

final int Lados = 4;
final int Distancia = 35;

double angulo = 360 / Lados;

gw.setTitle( "Poly" );
for(int i = 0; i < Lados; ++i) {
 turtle.turnRight( angulo );
 turtle.forward( Distancia );
}
```

Y recuerda, no es que no se pueda utilizar **def**, de hecho es más cómodo, y por tanto aconsejable.

11 Procedimientos

Los procedimientos son bloques de código a los que se les da un nombre, para poder utilizarlos más adelante mediante ese nombre.

Los bloques de código se encierran entre llaves { y }. No es nada nuevo, en realidad: los hemos estado utilizando constantemente para indicar el código que se debe ejecutar cuando una condición se cumple (o no), o cuando una repetición se sucede. Por supuesto, tienen muchas más posibilidades, que iremos desgranando poco a poco.

11.1 Sintaxis

```
void nombreProcedimiento()
{
    instruccion1;
    instruccion2;
    ...
    instruccionn;
}
```

11.2 Ejemplo

```
void dibujaCuadrado()
{
    for(def i = 0; i < 4; ++i) {
        turtle.forward( 100 );
        turtle.turnRight( 90 );
    }
}
```

Ahora disponemos de un código que podemos invocar cuando queramos. Por ejemplo, sería posible dibujar cuatro cuadrados seguidos con solo llamar a este procedimiento. Para hacerlo, solo es necesario poner su nombre, y a continuación, la apertura y cierre de paréntesis.

```
turtle.forward( 25 );
dibujaCuadrado();

turtle.forward( 25 );
dibujaCuadrado();

turtle.forward( 25 );
dibujaCuadrado();
```

```
turtle.forward( 25 );
dibujaCuadrado();
```

Por supuesto, ya sabemos que podemos mejorar un poco este código. Todo lo que suponga repeticiones, es mejor, al fin y al cabo, meterlo dentro de un bucle:

```
for(def i = 0; i < 4; ++i) {
    turtle.forward( 25 );
    dibujaCuadrado()
}
```

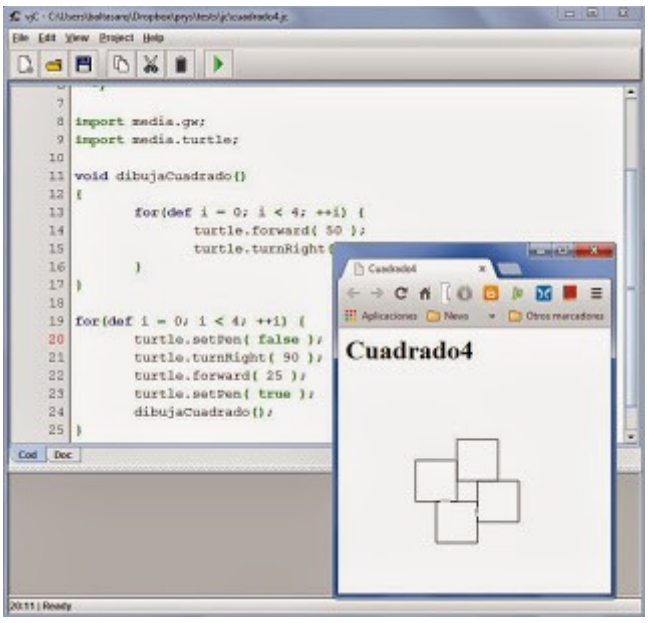
Hablando de mejorar el código, no es bueno que aparezcan tantas constantes que no sabemos muy bien de dónde salen. Aunque en este momento estamos exagerando, pronto veremos que es muy útil tener estos valores como constantes o variables. Podemos mejorar nuestro primer procedimiento:

```
void dibujaCuadrado()
{
 final def Lados = 4;
 final def Angulo = 90;
 final def Distancia = 50;

 for(def i = 0; i < Lados; ++i) {
  turtle.forward( Distancia );
  turtle.turnRight( Angulo );
 }
}
```

11.3 Programa final

El programa final, que hemos ido desgranando en esta entrada, queda como aparece más abajo. Se ha introducido el uso de *setPen()*, un procedimiento en el módulo *turtle*. Lo que hace este procedimiento (sí, son también procedimientos de la librería de **jC**), es evitar que la tortuga deje rastro o no, según se le pase *true* o *false*.

```
/** @name   Cuadrado4
  * @brief  Dibuja cuatro cuadrados en la pantalla.
  * @author jbgarcia@uvigo.es
  * @date   2013-09-29
  */

import media.gw;
import media.turtle;

void dibujaCuadrado()
{
 final def Lados = 4;
 final def Angulo = 90;
 final def Distancia = 50;

 for(def i = 0; i < Lados; ++i) {
  turtle.forward( Distancia );
  turtle.turnRight( Angulo );
}
}

for(def i = 0; i < 4; ++i) {
 turtle.setPen( false );
 turtle.turnRight( 90 );
 turtle.forward( 25 );
 turtle.setPen( true );
 dibujaCuadrado();
}
```



11.4 Parámetros en procedimientos

En la última entrada, aprendimos a crear procedimientos: consiste en darle un nombre a un código, de manera que podamos invocarlo cuando queramos.

Nuestro procedimiento, en este momento, sirve para crear cuadrados.

```
void dibujaCuadrado()
{
 final def Lados = 4;
 final def Angulo = 90;
 final def Distancia = 50;

 for(def i = 0; i < Lados; ++i) {
  turtle.forward( Distancia );
  turtle.turnRight( Angulo );
 }
}
```

De acuerdo, pero quizás... podríamos hacer que la distancia (la longitud del lado, representada por la distancia que recorre la tortuga), pudiese cambiar. De esta manera, podríamos crear cuadrados más grandes o más pequeños. Eso se puede conseguir, precisamente, mediante parámetros. Le añadiremos parámetros al procedimiento, de forma que podamos cambiar algunos aspectos. De hecho, si pudiéramos cambiar el número de lados... ¡podríamos dibujar cualquier polígono regular, pues el ángulo se obtiene dividiendo 360 entre el número de lados!

11.5 Sintaxis de creación del procedimiento

```
void nombreProcedimiento(tipo1 id1 [, tipo2 id2 [,... tipon idn]])
{
    instruccion1;
    instruccion2;
    ...
    instruccionn;
}
```

11.6 Sintaxis de llamada al procedimiento

```
nombreProcedimiento( expresión1 [, expresión2 [,... expresiónn]]);
```

Los parámetros, tal cual aparecen en la definición del procedimiento, se denominan formales. Los que aparecen cuando llamamos al procedimiento, se denominan reales. Los parámetros reales pueden ser valores literales, variables, cálculos...

11.7 Ejemplo

```
void dibujaPoligono(int lados, int distancia)
{
 def angulo = 360 / lados;

 for(def i = 0; i < lados; ++i) {
  turtle.forward( distancia );
  turtle.turnRight( angulo );
 }
}

dibujaPoligono( 4, 50 );
```

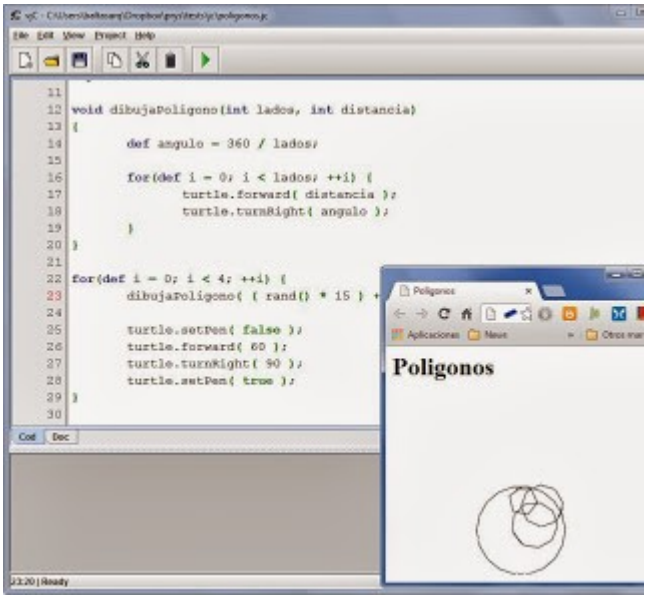
En este ejemplo, estaríamos dibujando un polígono regular de 4 lados, es decir, un cuadrado cuyo lado es de longitud 50.

11.8 Ejemplo

Podemos hacer figuras aleatorias utilizando el procedimiento *rand()* en el módulo math. Veamos un programa completo, que crea cuatro figuras haciendo girar la tortuga cuatro veces.

```
/** @name   poligonos
  * @brief  Dibuja polígonos al azar
  * @author jbgarcia@uvigo.es
  * @date   2013-9-30
  */

import media.gw;
import media.turtle;
import std.math;

void dibujaPoligono(int lados, int distancia)
{
 def angulo = 360 / lados;

 for(def i = 0; i < lados; ++i) {
  turtle.forward( distancia );
  turtle.turnRight( angulo );
 }
}
```

```
for(def i = 0; i < 4; ++i) {
 dibujaPoligono( ( rand() * 15 ) + 3, 20 );

 turtle.setPen( false );
 turtle.forward( 60 );
 turtle.turnRight( 90 );
 turtle.setPen( true );
}
```



12 Vectores

Los **vectores** son tipos complejos de datos. Esto es así porque se crean a partir de datos simples: en concreto, se trata de colecciones de datos de tipo simple, de tal forma que el tamaño de la colección siempre es fijo. El tamaño de cualquier vector se puede obtener utilizando la función *size()*.

Los tipos de los vectores se crean simplemente añadiendo "[]" al tipo base del vector (es decir, el tipo de cada elemento del mismo). Para acceder a un elemento del vector, se indica el nombre del vector, y a continuación, entre corchetes, la posición a la que se quiere acceder. Es necesario tener en cuenta que la posición del primer elemento es 0, y que la última posición de un vector *v* es *size(v) - 1*.

Tipo	Explicación
int	El tipo es **int[]**. Se trata de un vector de números enteros.
double	El tipo es **double[]**. Se trata de un vector de números reales.
char	El tipo es **char[]**. Se trata de un vector de caracteres. Los vectores de caracteres tienen un significado especial: se interpretan como texto, en el que cada posición del vector es un carácter del mismo.

Aunque es posible crear un vector de **booleanos**, no es con diferencia un vector tan utilizado como los tres tipos anteriores.

Supongamos como ejemplo que se desea calcular la media de un vector de números reales. La creación de un vector se hace utilizando el operador **new**, que crea un nuevo vector. A continuación, se indica el tipo, y, entre corchetes, el número total de elementos.

```
double[] v = new double[ 3 ];

v[ 0 ] = 10;
v[ 1 ] = 12;
v[ 2 ] = 14;

def suma = 0.0;
for(def i = 0; i < size( v ); ++i) {
    suma += v[ i ];
}

print( "La media es: " );
println( suma / size( v ) );
```

Cuando conocemos de antemano los valores que va a guardar un vector, es posible crear el vector sin hacer cada una de las asignaciones a cada una de sus posiciones. Así, el caso anterior es equivalente al programa siguiente:

```
double[] v = new double[]{ 10, 12, 14 };
def suma = 0.0;
for(def i = 0; i < size( v ); ++i) {
    suma += v[ i ];
}

print( "La media es: " );
println( suma / size( v ) );
```

Los vectores de caracteres se interpretan como texto o, muy comúnmente llamados, cadenas de texto. Así, por ejemplo, no es necesario crear un bucle para visualizar un vector de caracteres, sino que ya se visualiza directamente mediante *print()*. También es posible asignar directamente un texto a una variable de tipo vector de caracteres: **jC** se encargará de crear el vector correspondiente.

```
import std.io;

char[] mensaje = "Hola, mundo";
println( mensaje );
```

Como siempre, es posible obviar **char[]** y sustituirlo por **def** en la segunda línea.

13 Más sobre el paso de parámetros

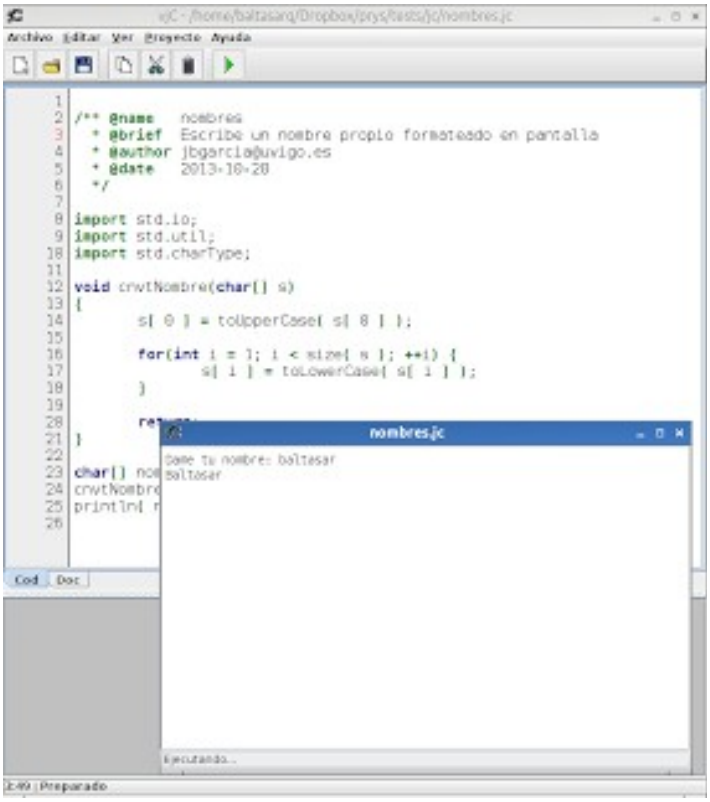
El paso de parámetros no siempre es por valor (también conocido "por copia"). El problema consiste en que los tipos de datos más complejos, como los vectores y matrices, tardan mucho en ser copiados. Así, no es lo mismo pasar un número entero, que un vector de 200 numeros enteros.

Por eso, el paso de vectores se hace por referencia. No hace falta marcar nada, simplemente sucede. Se le llama "por referencia" porque se pasa un número entero que permite localizar al vector, y referirse a él.

La implicación de todo esto es que, cuando se cambian las posiciones de un vector, los cambios realizados se mantienen después de volver de la función que lo llamó.

```
/** @name   nombres
  * @brief  Escribe un nombre propio formateado en pantalla
  * @author jbgarcia@uvigo.es
  * @date   2013-10-28
  */

import std.io;
import std.util;
import std.charType;

void cnvtNombre(char[] s)
{
 s[ 0 ] = toUpperCase( s[ 0 ] );

 for(def i = 1; i < size( s ); ++i) {
  s[ i ] = toLowerCase( s[ i ] );
 }

 return;
}

char[] nombre = readln( "Dame tu nombre: " );
cnvtNombre( nombre );
println( nombre );
```

Este programa toma un nombre, y lo convierte, de forma que la inicial pasa a estar en mayúsculas, y el resto de las letras del nombre pasado, en minúsculas. Esto se hace en la función *cnvtNombre()*, y dado que el vector se pasa por referencia (es el vector de caracteres *s* en *cnvtNombre()*), los cambios repercuten en el programa principal (donde el vector se llama *nombre*), con lo que se puede visualizar el resultado sin problema.



14 Funciones

Las funciones son como procedimientos, pero en lugar de no devolver nada, devuelven un valor, que se marca con **return**. El tipo de la devolución del valor se indica en el lugar en el que en un procedimiento se indica **void**.

Recordemos por tanto un procedimiento:

```
void imprimeConAsteriscos(char[] msg)
{
    print( "*** " );
    println( msg );
}
```

Una función se utiliza para realizar algún tipo de cálculo o proceso, cuyo resultado hay que devolver. Una función muy sencilla se puede ver a continuación:

```
double alCuadrado(double x)
{
    return ( x * x );
}
```

Una vez creada la función, es posible utilizarla en cualquier otra función o procedimiento, de igual forma que se utiliza un procedimiento (indicando su nombre y parámetros, estos entre paréntesis), pero con la peculiaridad de que es necesario guardar

el resultado en una variable o utilizarlo en otro cálculo. No es necesario que la función a utilizar haya sido creado antes de la función que la utiliza. Por ejemplo, la hipotenusa de un triángulo rectángulo se calcula haciendo la raiz cuadrada de la suma del cuadrado de las hipotenusas. El programa completo aparece a continuación.

```
import std.math;
import std.io;

double calculaHipotenusa(double cateto1, double cateto2)
{
    return sqrt( alCuadrado( cateto1 ) + alCuadrado( cateto2 ) );
}

double alCuadrado(double x)
{
    return ( x * x );
}

print( "Hipotenusa para un triángulo de lados 5 y 6:" );
println( calculaHipotenusa( 5, 6 ) );
```

Un programa anterior transformaba un nombre, convirtiéndolo a minúsculas, a excepción de la primera letra, que siempre se pone en mayúsculas. Para hacerlo, se utilizaba un procedimiento y se aprovechaba el hecho de que los vectores se pasan por referencia.

```
/** @name   nombres
  * @brief  Escribe un nombre propio formateado en pantalla
  * @author jbgarcia@uvigo.es
  * @date   2013-10-28
  */

import std.io;
import std.util;
import std.charType;

void cnvtNombre(char[] s)
{
 s[ 0 ] = toUpperCase( s[ 0 ] );

 for(def i = 1; i < size( s ); ++i) {
  s[ i ] = toLowerCase( s[ i ] );
 }

 return;
}

char[] nombre = readln( "Dame tu nombre: " );
cnvtNombre( nombre );
println( nombre );
```

En lugar de crear un procedimiento para realizar esta tarea, es posible escribir una función que cree una copia de la cadena de texto, y devuelva una cadena nueva. Una ventaja de esto es que la cadena original no se modifica, si bien por el contrario la desventaja es que se utiliza el doble de memoria. El programa completo aparece a continuación.

```
/** @name   nombres
  * @brief  Escribe un nombre propio formateado en pantalla
  * @author jbgarcia@uvigo.es
  * @date   2013-10-28
```

```
  */

import std.io;
import std.util;
import std.charType;

char[] cnvtNombre(char[] s)
{
 def toret = new char[ size( s ) ];

 toret[ 0 ] = toUpperCase( s[ 0 ] );

 for(int i = 1; i < size( s ); ++i) {
  toret[ i ] = toLowerCase( s[ i ] );
 }

 return toret;
}

char[] nombre = readln( "Dame tu nombre: " );
println( cnvtNombre( nombre ) );
```

15 La consola

Aunque ya la hemos estado utilizando en muchos ejercicios, vamos a estudiar un poco más en profundidad cómo funciona la consola en **jC**.

Además de pintar, **jC** también es capaz de pedir e imprimir texto. Esto se denomina "consola", pues es la forma estándar de comunicarse con un ordenador: introduciendo órdenes por el teclado y observando el resultado por la pantalla. Las funciones que tenemos a nuestra disposición para manejarnos con la consola son precisamente dos:

- **print(<expr>)**: Permite mostrar cualquier valor por la pantalla. Hay dos variantes: **print()** y **println()**, la segunda realiza un salto de línea tras imprimir.

  Ejemplo

  ```
  print( "Tu nombre es: " );
  println( nombre );
  ```

- **readln(<msg>)**: Devuelve la cadena de texto introducida por el usuario. El mensaje permite que el usuario conozca qué datos se le están preguntando.

  Ejemplo

  ```
  def nombre = readln( "Introduce tu nombre: " );
  print( "Hola, " );
  println( nombre );
  ```

  Dado que lo que devuelve *readln()* es una cadena, el código anterior es equivalente a:

  Ejemplo

  ```
  char[] nombre = readln( "Introduce tu nombre: " );
  print( "Hola, " );
  println( nombre );
  ```

No siempre querremos obtener cadenas de texto por parte del usuario. En ocasiones, querremos preguntarle datos numéricos, como por ejemplo, una edad o una altura. Aunque sólo es posible leer cadenas de texto de la consola, siempre es posible convertirlas a un número, bien sea este entero (**int**) o real (**double**). En la librería estándar *std.string*, tenemos a nuestra disposición las funciones *strToInt()* y *strToDbl()*, que convierten una cadena pasada por parámetro en un número, entero y real, respectivamente.

Funciones de conversión de cadena a número:

- **strToInt(<cadena de texto>)**: Esta función acepta una cadena de texto como parámetro, y devuelve el número entero que contiene.

  Ejemplo

  ```
  int edad = strToInt( cadenaDeTexto );
  println( edad );
  ```

- **strToDbl(<cadena de texto>)**: Esta función acepta una cadena de texto como parámetro, y devuelve el número real que contiene.

  Ejemplo

  ```
  int altura = strToDbl( cadenaDeTexto );
  println( altura );
  ```

Realizaremos ahora un programa completo, que permita convertir grados celsius en farenheit, y viceversa. Para convertir los grados celsius (c) en farenheit (f), es necesario aplicar la fórmula:

```
f = ( c * 1,8 ) + 32
```

Para realizar la conversión inversa, de farenheit (f) a celsius (c):

```
c = ( f - 32 ) / 1,8
```

Es muy fácil escribir, por tanto dos funciones que hagan las conversiones respectivas. Dado que los grados llevan posiciones decimales, es necesario que las variables a utilizar sean de tipo número real (**double**).

```
double cnvtGradosCelsiusFarenheit(double x)
{
 return ( ( x * 1.8 ) + 32 );
}

double cnvtGradosFarenheitCelsius(double x)
{
 return ( ( x - 32 ) / 1.8 );
}
```

Sólo resta, por tanto, pedir una cadena de texto, y convertirla a un número real (**double**), esto puede hacerse de manera muy sencilla:

```
def grados = strToDbl( readln( "Introduce unos grados: " ) );
```

Una vez que se ha llamado a las funciones anteriores con la variable *grados*, sólo es necesario imprimir el resultado. El programa completo aparece a continuación:

```
/** @name   conversorGrados
  * @brief  Convierte los grados celsius a farenheit y vicecersa.
  * @author jbgarcia@uvigo.es
  * @date   2013-11-21
  */

import std.io;
import std.string;

double cnvtGradosCelsiusFarenheit(double x)
{
 return ( ( x * 1.8 ) + 32 );
}

double cnvtGradosFarenheitCelsius(double x)
{
 return ( ( x - 32 ) / 1.8 );
}

def grados = strToDbl( readln( "Introduce unos grados: " ) );

print( "Los grados celsius introducidos son grados farenheit: " );
println( cnvtGradosCelsiusFarenheit( grados ) );

print( "Los grados farenheit introducidos son grados celsius: " );
println( cnvtGradosFarenheitCelsius( grados ) );
```

15.1 Mezclando consola y gráficos

La consola (entrada y salida de texto) y la salida gráfica no son incompatibles. **jC** es capaz de mezclar ambas cosas, para poder realizar aplicaciones más interesantes.

En este caso, la tarea a realizar será retomar la creación de polígonos, aunque tomando los datos (distancia y lados) de la consola, de tal manera que sea el usuario el que pueda elegir qué polígono representar.

Ya conocemos la función necesaria para poder dibujar un polígono, como se vió en una entrada anterior. Ya que lo que hace que la figura a representar varíe es el número de lados y la distancia a recorrer para cada lado, esos serán los parámetros de la función:

```
void dibujaPoligono(int lados, int distancia)
{
 def angulo = 360 / lados;

 for(def i = 0; i < lados; ++i) {
  turtle.turnRight( angulo );
  turtle.forward( distancia );
 }
}
```

En realidad, conociendo esto, sólo es necesario pedir los datos (*lados* y *distancia*), y llamar a la función adecuada. El código aparece a continuación.

```
/** @name   Poly
  * @brief  Dibuja poligonos
  * @author jbgarcia@uvigo.es
  * @date   2013-8-10
  */

import std.io;
import std.string;
import media.gw;
import media.turtle;

void dibujaPoligono(int lados, int distancia)
{
 def angulo = 360 / lados;
 for(def i = 0; i < lados; ++i) {
  turtle.turnRight( angulo );
  turtle.forward( distancia );
```

```
 }
}

gw.setTitle( "Poly" );
def lados = strToInt( readln( "Introduzca num. de lados: " ) );
def distancia = strToInt( readln( "Introduzca distancia de lado: " ) );
dibujaPoligono( lados, distancia );
```