Programación Orientada a Objetos

En este tema profundizaremos en lo poco que conocemos sobre programación orientada a objetos (que básicamente se reduce a dos métodos que sabemos usar de la clase String y a que los métodos se definen dentro de las clases).
Empezaremos presentando algunas de las clases pertenecientes a las bibliotecas de la acm que permiten representar objetos gráficos como líneas, rectángulos, círculos, etc. El objetivo no será aprender conceptos sobre gráficos por computadora sino intuiciones sobre programación orientada a objetos.
Continuaremos ampliando nuestros conocimientos sobre una clase ya conocida, la clase String, cuyo dominio es básico para casi cualquier tipo de aplicación. Asociados a la clase String presentaremos algunos detalles sobre la manipulación de caracteres en Java.
Posteriormente mostraremos cómo definir nuevas clases usando Java. Para ello partiremos de un concepto conocido, la abstracción funcional, para llegar a otro, el de la abstracción de datos, que en Java conseguiremos definiendo clases.
Durante el tema revisitaremos alguno de los ejemplos ya vistos a lo largo de la asignatura, puliéndolos un poco más a partir de los nuevos conocimientos que vamos adquiriendo.
Es difícil encontrar un orden de presentación de todos los conceptos alrededor de la orientación a objetos en el que, cuando se presenta un concepto, todos los anteriores ya están completamente explicados. Al finalizar el tema, cuando hayamos visto todos los conceptos, algunos detalles de los ejemplos iniciales, en los que no quedaba claro su porqué, cobrarán perfecto sentido.

1. Un nuevo tipo de programa: GraphicsProgram

Cuando queramos que nuestro programa dibuje sobre la pantalla diversos elementos gráficos, deberemos extender la clase GraphicsProgram. En este caso, la ventana asociada a la aplicación será como un **lienzo** (en inglés *canvas*) sobre el que podremos colocar los diversos elementos gráficos en las posiciones que queramos.
La colocación de los elementos gráficos funciona como un **collage**, es decir, si no los reordenamos los nuevos elementos se colocan sobre los que ya están.

```
1 import acm.graphics.GLabel;
```

```
 2 import acm.graphics.GRect;
 3 import acm.program.GraphicsProgram;
 4 import java.awt.Color;
 5
 6 public class HelloWorld extends GraphicsProgram {
 7
 8   public int TIMEOUT = 1000;
 9
10   public void run() {
11     GLabel label = new GLabel("hello, world", 100, 75);
12     add(label);
13     pause(TIMEOUT);
14     label.move(-50, 50);
15     pause(TIMEOUT);
16     label.setColor(Color.BLUE);
17     pause(TIMEOUT);
18     label.setVisible(false);
19     pause(TIMEOUT);
20     label.setVisible(true);
21     pause(TIMEOUT);
22     GRect rect = new GRect(100, 75, 30, 50);
23     rect.setColor(Color.YELLOW);
24     add(rect);
25     pause(TIMEOUT);
26     rect.setFilled(true);
27     rect.setFillColor(Color.RED);
28   }
29 }
```

Vamos a describir lo que sucede en cada una de las líneas de este programa:

- Líneas 1 a 4: importamos las clases que vamos a usar. A destacar las clases GLabel y GRect del paquete acm.graphics (que agrupa los elementos gráficos de las bibliotecas acm) y la clase Color del paquete java.awt (el Abstract Window Toolkit, que forma parte de las clases estándar de Java).
- Línea 6: como queremos dibujar gráficos en la pantalla, el programa principal ha de extender GraphicsProgram.
- Línea 8: definimos una constante para indicar la duración en milisegundos de las pausas que haremos tras cada operación (para que se vea durante la ejecución).
- Línea 11: creamos un instancia de la clase GLabel con texto “hello, world” y que, cuando se añada al lienzo, aparecerá en las coordenadas 100, 75 de la pantalla. Respecto las coordenadas

indicar que la posición (0,0) corresponde al extremos superior izquierdo de la ventana de la aplicación. Cada posición de la pantalla corresponde a un **píxel** (*picture element*).

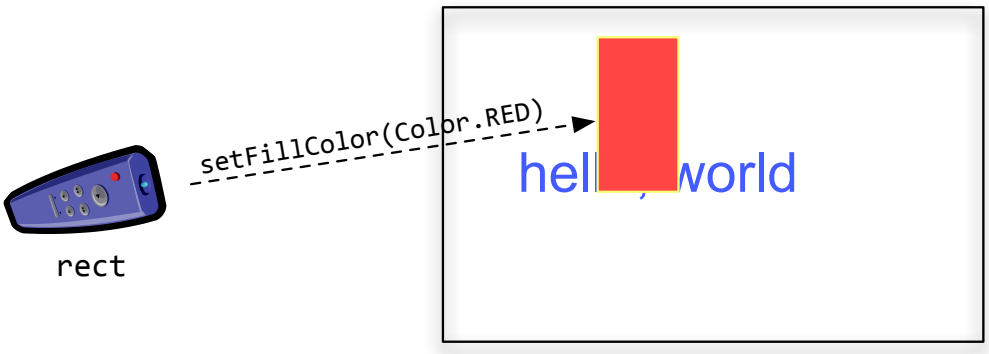
- Línea 12: añadimos la etiqueta creada a la ventana (como si la pegáramos sobre un lienzo).
- Líneas 13, 15, 17, 19, 21 y 25: pausas para poder apreciar los cambios que producen las instrucciones en la ventana.
- Línea 14: movemos la etiqueta 50 píxeles a la izquierda y 50 hacia abajo,
- Línea 16: cambiamos el color del texto de la etiqueta
- Línea 18: ocultamos la etiqueta (la etiqueta sigue existiendo pero no se ve).
- Línea 20: la volvemos a hacer visible.
- Líneas 22 a 24: creamos un rectángulo cuyo vértice superior izquierdo está en las coordenadas 100, 75 y que tiene anchura 30 y altura 50 y que es de color amarillo y lo añadimos al lienzo. Fijaos en que el rectángulo queda sobre la etiqueta, ya que lo hemos añadido al lienzo después.
- Líneas 26 y 27: indicamos que el rectángulo está pintado por dentro y que el color es rojo.

2. Objetos y referencias

Aunque ahondaremos más adelante en ello cuando tratemos el tema de gestión de memoria en Java, hemos de introducir una distinción que nos permitirá entender qué pasa cuando analizamos código que usa objetos: la distinción entre la **referencia** a un objeto y el **objeto** mismo.

De televisores y mandos a distancia

De modo similar a lo que sucede con los televisores modernos, la televisión no la manipulamos directamente sino que lo hacemos a través de un mando a distancia. Cómodamente sentados en nuestro sillón, apretamos botones en el mando para hacer que algo cambie en nuestro televisor (el canal seleccionado, el nivel de volumen, etc.).



En el caso de Java la situación es parecida: no manipulamos los objetos directamente sino que lo hacemos a través de referencias. Por eso cuando declaramos una variable por ejemplo de tipo GRect

```
GRect rect;
```

no tenemos ningún objeto, solamente hemos creado una variable para guardar el valor del mando a distancia. ¿Cómo le daremos valor? La operación new hace dos cosas: crea la tele y devuelve un mando a distancia para manipularla. Por ello, al hacer:

```
rect = new GRect(100, 75, 30, 50);
```

conseguimos que rect sea un mando a distancia apuntado al televisor que queremos. Y a partir de ahora, podremos usar el mando con el televisor.

Intentar usar un mando a distancia no enlazado con televisor alguno, no solamente no produce nada, sino que es un error. De hecho, antes de la inicialización, una referencia tiene el valor especial **null,** que puede interpretarse como "no se refiere a nada". Es un error intentar usar una referencia que no apunta a nada para invocar un método sobre ella. Es por ello que procuramos **inicializar** las referencias cuando las declaramos y escribimos:

```
GRect rect = new GRect(100, 75, 30, 50);
```

Es común preguntar si una referencia dada apunta a null o no, por lo que la estructura

```
if ( rect != null ) {
     ...
}
```

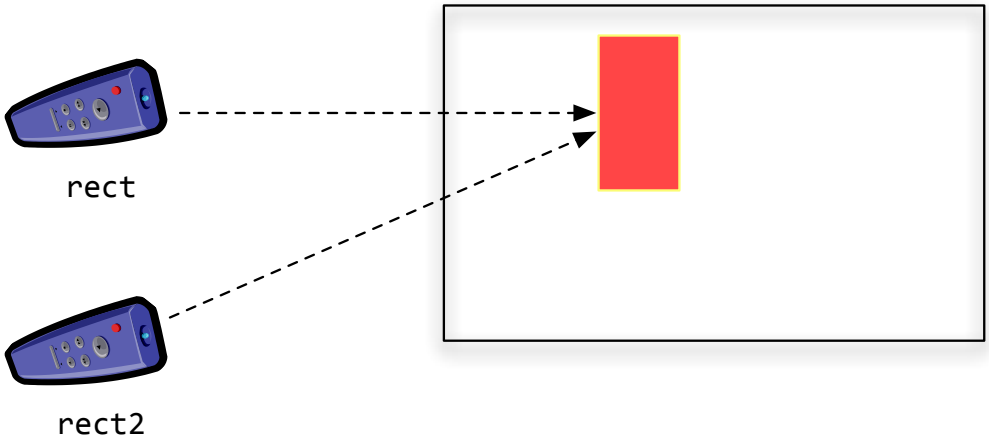
la veremos bastantes veces.

Aliasing

Un error común al manipular referencias a objetos es no tener en cuenta esta diferencia entre la referencia y el objeto referenciado. Por ejemplo, si hacemos:

```
GRect rect = new GRect(100, 75, 30, 50);
GRect rect2 = rect;
```

lo que estamos consiguiendo es que ambas referencias apunten al mismo objeto, es decir, que ambos mandos puedan usarse para manipular el mismo televisor.



Aunque ya se verá con más detalle más adelante, en el apartado sobre gestión de la memoria en Java, éste mecanismo de creación de alias es el mismo que se utiliza al pasar objetos como parámetros de las funciones: los parámetros referencian a los objetos que se han pasado como parámetro.

Igualdad y referencias

Una primera fuente de errores cuando no tenemos claro el concepto de referencia consiste en la interpretación de la igualdad entre referencias. Por ejemplo, consideremos el siguiente código:

```
GRect rect1 = new GRect(100, 75, 30, 50);
GRect rect2 = new GRect(100, 75, 30, 50);
GRect rect3 = rect1;
```

Y consideremos las siguientes expresiones:

- `(rect1 == rect2)` es `false`, ya que las referencias se refieren a objetos diferentes.
- `(rect1 == rect3)` es `true` ya que ambas referencias apuntan al mismo objeto.

Como veremos más adelante, algunas clases implementan un método llamado equals que comprueba si dos referencias se refieren a objetos que tienen el mismo valor.

Por ejemplo, en el caso de Strings, si quiero ver si el nombre de alguien es “Juan” he de hacer:

```
String name = readLine(“¿Cómo te llamas? ”);
if (name.equals(“Juan Manuel”)) {
     sameNameAsMe = sameNameAsMe + 1;
}
```

La comparación también puede escribirse como:

```
“Juan Manuel”.equals(name)
```

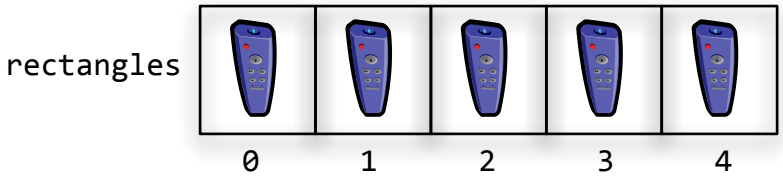
Más adelante veremos otros métodos para comparar Strings.

Vectores de referencias

Cuando creamos un vector de referencias, es decir,:

```
GRect[] rectangles = new GRect[10];
```

lo que hemos creado es un vector de referencias (inicializadas a null) en el que cada una de ellas podrá referenciar a un rectángulo. Lo que no hemos hecho es crear rectángulo alguno, es decir, lo que tenemos es:



Para crear rectángulos tendremos que hacer, por ejemplo:

```
rectangles[0] = new GRect(50, 70);
```

y ahora, la primera referencia del vector apuntará a el rectángulo creado (y las otras seguirán valiendo null).
Por elemplo, si queremos mover todos los vectores del array (algunos de los cuales pueden no estar inicializados) podemos hacer:

```
for ( int i = 0; i < rectangles.length; i++ ) {
  if ( rectangles[i] != null ) {
    rectangles[i].move(10, 10);
  }
}
```

3. Más sobre clases gráficas de acm

Para poder empezar a hacer más programas que utilicen algunas de las clases gráficas del paquete acm.graphics, mostraremos a continuación algunas de las clases y métodos que contienen.

Rectángulos, óvalos, líneas y etiquetas

- **Creación de instancias**:

new GLabel(string, x, y) Crea un nuevo GLabel que contiene el string especificado en las coordenadas (x, y)
new GRect(x, y, width, height) Crea un nuevo GRect de las dimensiones dadas y cuya esquina superior izquierda está en las coordenadas (x, y)

new GOval(x, y, width, height) Crea un nuevo GOval cuyo tamaño es tal que cabe en un GRect de las mismas dimensiones
new GLine(x1, y1, x2, y2) Crea un nuevo GLine que conecta los puntos (x1, y1) y (x2, y2)

- **Métodos comunes a todos los objetos gráficos**

object.**setColor(color)** Cambia el color del objeto a color (que es un java.awt.Color)
object.**setLocation(x, y)** Cambia la localización del objeto al punto (x, y)
object.**move(dx, dy)** Mueve el objeto añadiendo dx a su coordenada x y dy a su coordenada y

- **Métodos para GRect y GOval**

object.**setFilled(fill)** Indica si el objeto está pintado por dentro (fill es true) o no.
object.**setFillColor(color)** Cambia el color de relleno del objeto (este color puede ser diferente del usado para el borde)

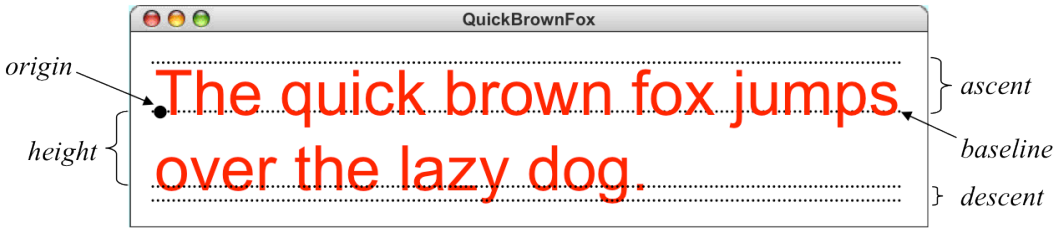
- **Métodos para GLabel**

label.**setFont(string)** Cambia el tipo de letra al indicado por string, que da la familia, el estilo y el tamaño.

Para ver información completa de todas las clases y métodos del paquete acm.graphics podéis consultar su documentación javadoc.

Algunos conceptos asociados a una etiqueta

El posicionamiento de una etiqueta sigue conceptos de tipografía. Algunos de ellos pueden verse en la figura siguiente:



Algunos métodos sobre `GraphicsProgram`

Como hemos dicho anteriormente un GraphicsProgram permite que nuestro programa funcione colocando y manipulando objetos gráficos sobre un lienzo.

Para llevar a cabo ello, nuestro programa puede utilizar los siguientes métodos[1]:

- **Métodos para añadir y eliminar objetos gráficos**

void add(GObject gobj) Añade el objeto gráfico gobj en el lienzo en la posición que el objeto gráfico tiene asignada.
void add(GObject gobj, double x, double y) Añade el objeto gráfico gobj en el lienzo en la posición (x, y)
void remove(GObject gobj) Elimina el objeto gráfico gobj del lienzo
void removeAll() Elimina todos los elementos gráficos del lienzo

- **Método para buscar un elemento gráfico en una determinada posición**

GObject getElementAt(double x, double y) Retorna el objeto de más encima que contiene la posición (x, y), o bien null si no hay ningún objeto que la contenga.

- **Métodos para proporcionar animación (sólo de GraphicsProgram)**

void pause(double milliseconds) Suspende la ejecución del programa por el intervalo de tiempo dado.
void waitForClick() Suspende la ejecución del programa hasta que el usuario pulse el botón del ratón en cualquier lugar del lienzo del programa

- **Otros métodos útiles**

int getWidth() Devuelve la anchura del lienzo en píxeles
int getHeight() Devuelve la altura del lienzo en píxeles
void setBackground(Color bg) Cambia el color de fondo del lienzo

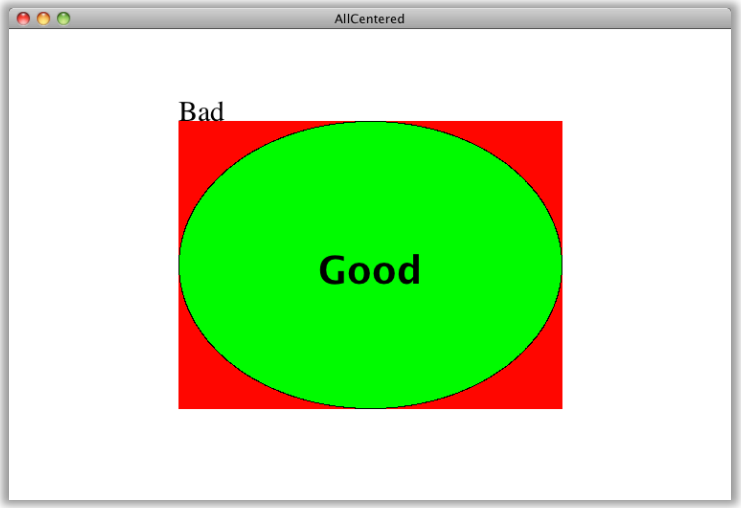
En los métodos anteriores GObject puede ser tanto un GLabel, GRect, GOval y GLine. Más adelante veremos que esta posibilidad se corresponde con el concepto de **polimorfismo**.

[1] Estos métodos pertenecen también a la clase GCanvas, aunque de todo ello ya hablaremos más adelante.

Ejemplo: Centrando elementos

```
1 public class AllCentered extends GraphicsProgram {
2
3   public double FIGURE_WIDTH = 125.0;
4   public double FIGURE_HEIGHT = 75.0;
5
6   public void run() {
7     double x = (getWidth() - FIGURE_WIDTH) / 2;
8     double y = (getHeight() - FIGURE_HEIGHT) / 2;
9     GRect rect = new GRect(x, y,
10                           FIGURE_WIDTH, FIGURE_HEIGHT);
11    rect.setFilled(true);
12    rect.setColor(Color.RED);
13    add(rect);
14    GOval oval = new GOval(x, y,
15                           FIGURE_WIDTH, FIGURE_HEIGHT);
16    oval.setFilled(true);
17    oval.setFillColor(Color.GREEN);
18    add(oval);
19    GLabel bad = new GLabel("Bad", x, y);
20    bad.setFont("Serif-30");
21    add(bad);
22    GLabel good = new GLabel("Good");
23    x = (getWidth() - good.getWidth()) / 2;
24    y = (getHeight() + good.getAscent()) / 2;
25    good.setFont("SansSerif-bold-40");
26    good.setLocation(x, y);
27    add(good);
28  }
29 }
```

Y el resultado es:



4. La clase `String`

Debido a que gran parte de la información que manipulan los programas es textual, en Java, la clase String, contiene muchos métodos para manipular texto.
Algunos de los métodos de la clase String ya los conocemos (p.e. length, toCharArray) pero hay muchos más.

int length() Devuelve la longitud de la cadena receptora
char charAt(int index) Devuelve el carácter en la posición especificada (numerada desde 0) de la cadena receptora
String concat(String str2) Concatena str2 al final de la cadena receptora, devolviendo una nueva cadena y dejando la cadena receptora sin modificar.
String substring(int p1, int p2) Retorna una subcadena que empieza en la posición p1 y que se extiende hasta, pero no incluye, la posición p2
String substring(int p1) Devuelve la subcadena empezando en la posición p1 y que se extiende hasta el final de la cadena receptora
String trim() Devuelve la subcadena formada al eliminar el espacio en blanco al principio y al final de la cadena receptora
boolean equals(String s2) Devuelve true si la cadena s2 es igual a la receptora (es decir, está formada por los mismos caracteres en el mismo orden)

boolean equalsIgnoreCase(String s2) Devuelve true si la cadena s2 es igual a la receptora ignorando mayúsculas y minúsculas.
int compareTo(String s2) Devuelve un número cuyo signo indica cómo las cadenas comparan lexicográficamente (negativo si la receptora es menor, cero si son iguales, y positivo si la receptora es mayor)
int indexOf(char c) o **indexOf(String s)** Devuelve el índice de la primera aparición de c (o s) en la cadena o -1 si no aparece.
int indexOf(char c, int start) o **indexOf(String s, int start)** Como los indexOf anteriores, pero empezando a buscar en la posición indicada por start
boolean startsWith(String prefix) Devuelve true si la cadena comienza por el prefijo indicado
boolean endsWith(String suffix) Devuelve true si la cadena finaliza por el sufijo indicado
String toUpperCase() Devuelve la cadena formada al convertir en mayúsculas todos los caracteres de la cadena receptora (que como siempre queda sin modificar)
String toLowerCase() Devuelve la cadena formada al convertir en minúsculas todos los caracteres de la cadena receptora (que como siempre queda sin modificar)

Un ejercicio interesante es implementar algunos de ellos a partir de obtener el vector de caracteres asociados.

Inmutabilidad

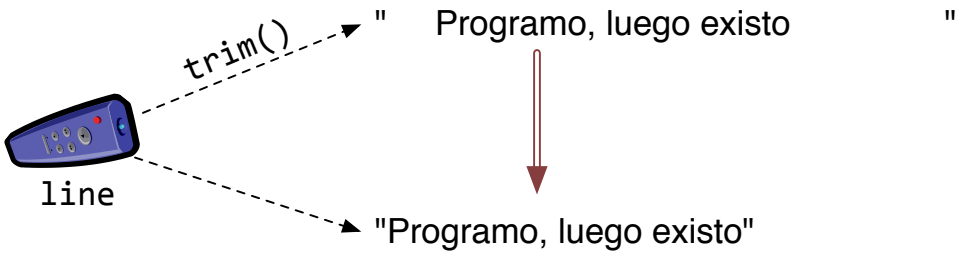
Los objetos de tipo cadena son inmutables, es decir, una vez se ha creado el objeto éste ya nunca cambiará de valor. Este comportamiento es diferente del de, por ejemplo, los objetos de la clase GRect que, una vez creados, pueden cambiar de posición, color, etc.
Un error común cuando usamos cadenas es no tener en cuenta este aspecto y hacer:

```
line.trim();
```

pensando que la invocación de dicho método elimina los espacios en blanco al principio y final de line.
Si queremos que el resultado quede en el objeto referenciado por line, lo que hemos de hacer es

```
line = line.trim();
```

de esta manera, después de la asignación, `line` dejará de referenciar la cadena original y pasará a referenciar a la cadena que se obtiene como resultado del método `trim` sobre la cadena original.



Concatenación

Una operación muy usual entre cadenas es la concatenación, es decir, obtener una cadena a partir de otras más simples. Aunque la clase cadena implementa el método concat y, por tanto podemos escribir:

```
str = str.concat("!");
```

en Java no veréis que se use demasiado ya que el operador + hace exactamente esta funcionalidad, pudiendo escribir lo anterior como:

```
str = str + "!";
```

o incluso:

```
str += "!";
```

Un error común consiste en confundir un carácter como `'!'` con la cadena de longitud uno que contiene ese carácter, es decir, `"!"`. Conforme avancéis es vuestros conocimientos de Java veréis que esta forma de construir cadenas no es la más adecuada y la forma de hacerlo consiste en utilizar la clase `StringBuilder`.

Comparaciones

Cuando comparamos cadenas un error común es utilizar los operadores relacionales (==, !=, >=, >, ...) en vez de los métodos de comparación que provee la clase `String`, básicamente `equals` y `compareTo`.

Legibilidad

Fijaos en que algunos de los métodos que proporciona la clase funcionalmente parecen redundantes. Por ejemplo, para ver si una cadena es prefijo de otra, podemos perfectamente hacer:

```
1 if ( str.indexOf(prefix) == 0 ) {
2   ...
3 }
```

¿Para qué existe entonces `startsWith`? Para conseguir que el código que hacemos sea mucho más legible. Comparar el código anterior con:

```
1 if ( str.startsWith(prefix) ) {
2   ...
3 }
```

Cuando diseñéis vuestras propias clases es importante tener en cuenta la legibilidad del código que las va a utilizar.

Caracteres en Java

Un tema ligado a la clase String es el del manejo de caracteres en Java. El tipo primitivo char representa caracteres en Unicode, es decir, permite representar caracteres en cualquier alfabeto y cada carácter ocupa 16 bits (2 bytes). En C y C++, los caracteres están representados en ASCII (o en alguna codificación ampliada como ISO-8859-1 o latin1) y ocupan 8 bits (1 byte).
En Java, de forma similar a C, es posible usar operadores aritméticos con valores carácter y cuando se usa un carácter se utiliza el valor numérico asociado por Unicode al carácter. Por ejemplo, si hago

```
int nextInt = ch + 1;
```

y ch es el carácter 'A', el valor de nextInt será 66, ya que Unicode asigna a la letra 'A' el valor 65.
En cambio, sí es necesario realizar una conversión de tipos para convertir de entero a carácter, ya que el rango de caracteres es menor que el de los enteros (no todos los números corresponden a caracteres Unicode). Es decir, para obtener el siguiente carácter a ch como char he de hacer

```
char nextChar = (char) ('A' + 1);
```

y el valor de nextChar es 'B'.
Por ejemplo un método para convertir de mayúsculas a minúsculas podría implementarse como:

```
1 public char toLowerCase(char ch) {
2   if (ch >= 'A' && ch <= 'Z') {
3     return (char) (ch + 'a' - 'A');
4   } else {
5     return ch;
6 }
```

El problema es que este método solamente funcionaría con los caracteres del alfabeto latino. Es por ello que Java provee métodos para este tipo de conversiones en la clase Character que son aplicables a todos los alfabetos representables en Unicode. Dichos métodos son estáticos (qué quiere decir esto ya lo veremos más adelante) y para usarlos, por ejemplo, deberemos hacer:

```
char lower = Character.toLowerCase(upper);
```

La clase `StringTokenizer`

Una de las tareas más comunes que se realizan con las cadenas, consiste en romperla en trozos delimitados por caracteres. Claramente los conocimientos que tenemos sobre recorridos y búsquedas de secuencias nos permiten implementar este tipo de tareas, pero de cara a avanzar en nuestros conocimientos sobre Java, es conveniente conocer algunas de las clases que proporciona Java para esta tarea. Una de las más simples es la clase `StringTokenizer` (del paquete `java.util`, por lo que **sí** hace falta importarla).

Algunos métodos útiles de esta clase son:

- **Constructores**

new StringTokenizer(String str) Crea una instancia de StringTokenizer que lee trozos de la cadena separados por caracteres en blanco
new StringTokenizer(String str, String delims) Crea una instancia de StringTokenizer que lee trozos de la cadena separados por caracteres pertenecientes a la cadena delims
new StringTokenizer(String str, String delims, boolean returnDelims) Crea una instancia de `StringTokenizer` que lee trozos de la cadena separados por caracteres pertenecientes a la cadena delims y que devuelve o no los pedazos formados por delimitadores dependiendo del valor de `returnDelims`

- **Métodos para leer los trozos de la cadena**

boolean hasMoreTokens() Devuelve true si **hay** más trozos a extraer del `StringTokenizer`
String nextToken() Devuelve el siguiente trozo disponible de la cadena y es un error usarlo cuando ya no quedan (de ahí el método `hasMoreTokens`)

Por ejemplo, si queremos sumar las longitudes de las palabras leídas en una línea y dónde las palabras pueden estar separadas por espacio, coma o punto, podemos hacer:

```
1 String line = readLine("Entra una línea: ");
2 String tokenizer = new StringTokenizer(line, " ,.");
3 int sumWordLenghts = 0;
4 while ( tokenizer.hasMoreTokens() ) {
5   String word = tokenizer.nextToken();
6   sumWordLengths += word.length();
7 }
```

5. Los métodos como mecanismo de abstracción

Antes de introducir conceptos nuevos como los de clase y objeto, y de su uso para hacer abstracción de datos, vamos a reflexionar brevemente sobre un concepto ya conocido: la abstracción procedimental.

Supongamos que, diseñando un programa de control de la temperatura para una casa inteligente, nos damos cuenta de que en varios lugares hemos de pasar una temperatura de grados Celsius a Fahrenheit. Por ejemplo, si necesitamos la media de temperaturas y tenemos sensores de diferentes fabricantes que usan unidades diferentes, podemos tener código similar a:

```
1 ...
2 sensor1_f = readSensor1(); // In F
3 sensor1_c = 5.0 / 9.0 * (sensor1_f - 32.0);
4 sensor2_c = readSensor2();
5 sensor3_f = readSensor3();
6 sensor3_c = 5.0 / 9.0 * (sensor3_f - 32.0);
7 mean_c = (sensor1_c + sensor2_c + sensor3_c) / 3;
8 ...
```

¿Qué haríamos? Para no tener la **misma expresión** repetida en todos los lugares dónde se ha de realizar la conversión, introduciríamos una función para calcularla. La función calcularía la expresión dada, pero haría abstracción del valor concreto que se usa tanto en la línea 2 como en la línea 5. Para ello usaríamos un parámetro que representaría el valor que varía en esas dos líneas. Es decir:

```
1 public double fahrenheitToCelsius(double f) {
2   return 5.0 / 9.0 * (f – 32.0);
3 }
```

De manera que ahora el código anterior podría sustituirse por:

```
4 double sensor1_f = readSensor1(); // In F
5 double sensor1_c = fahrenheitToCelsius(sensor1_f);
6 double sensor2_c = readSensor2(); // In C
7 double sensor3_f = readSensor3(); // In F
8 double sensor3_c = fahrenheitToCelsius(sensor2_f);
9 double mean_c = (sensor1_c + sensor2_c + sensor3_c) / 3;
10 ...
```

Fijaos en que el método puede verse como una **plantilla** de una expresión en la que se pueden usar valores diferentes para los

parámetros formales. Los valores a usar se proporcionarán cuando realicemos la invocación o llamada a dicha función.
En resumen, el mecanismo que hemos definido tiene:

- Nombre: damos un nombre al conjunto de instrucciones para poder usarlo posteriormente
- Parámetros formales: en la definición usamos unos parámetros formales para representar aquellos aspectos que variarán de una llamada a otra
- Invocación: el lenguaje proporciona un mecanismo para invocar las instrucciones asociadas al nombre de la función, permitiendo pasar los valores de los parámetros a utilizar

Otra propiedad interesante que tenemos es la capacidad que tenemos de combinar estas funciones, es decir, que dentro de las instrucciones que contiene una función podemos, a su vez, usar llamadas a otras, que a su vez ... Esta posibilidad permite realizar la descomposición funcional de un problema complejo en subproblemas y el diseño descendente de las soluciones.

6. Repetición de datos

De igual modo que en el caso de instrucciones, también hay repetición en el caso de los datos. Supongamos estamos realizando un programa que simula la trayectoria de una partícula en dos dimensiones. Para representar la posición de dicha partícula, tendríamos:

```
1 // Position
2 double xPos;
3 double yPos;
```

Para representar su velocidad

```
4 // Velocity
5 double xVel;
6 double yVel;
```

Y para la aceleración

```
7 // Acceleration
8 double xAccel;
9 double yAccel;
```

Demos unos valores iniciales a las posiciones, velocidades y aceleraciones

```
10 xPos = 10.0;
```

```
11 yPos = 20.0;
12 xVel = 0.0;
13 yVel = -5.0;
14 xAccel = 0.2;
15 yAccel = 0.8;
```

Y el código para actualizar su velocidad sería:

```
16 xVel = xVel + xAccel;
17 yVel = yVel + yAccel;
```

Y el que actualiza la posición

```
18 xPos = xPos + xVel;
19 yPos = yPos + yVel;
```

Siguiendo esta lógica, si tuviéramos que representar la información asociada a una fuerza usaríamos la pareja de variable

```
20 double xForce;
21 double yForce;
```

Además, si queremos calcular la magnitud de una fuerza, por ejemplo la de la velocidad, tendríamos:

```
22 double speed = Math.sqrt(xVel*xVel + yVel*yVel);
```

De la misma manera que hemos podido capturar el patrón formado por un grupo de instrucciones, nos gustaría poder capturar este patrón: un par de variables que representan los dos componentes de un vector. Así, en vez de tener que referirse en todas partes a los componentes por separado, puedo referirme como una sola cosa, en este caso un vector en dos dimensiones.

Ampliando conceptos

Desde otro punto de vista, lo que sucede es que nuestro problema está planteado sobre unos conceptos, que son los vectores en dos dimensiones, que no se corresponden con ninguno de los tipos que proporciona nuestro lenguaje. Es por ello que hemos 'simulado' el concepto usando una pareja de valores double (es decir, nos hemos apañado con lo que teníamos). El poder crear nuevas clases permite ampliar los conceptos que podemos usar en nuestros programas, de manera que, para cada problema, buscaremos los conceptos más adecuados para representar la solución y los representaremos como clases.

7. Requisitos para una solución

La solución consiste en aplicar lo mismo que hemos visto en el caso de la repetición de código, para la repetición de datos. Es decir:

- La posibilidad de agrupar bajo un nombre a este conjunto de datos
- Tener nombres formales para las cosas que varían de un caso particular a otro (ya que cada vector particular tendrá su componente x y su componente y). Estos elementos de datos se denominan **campos**[2] de la clase.
- La posibilidad de instanciar la plantilla para crear una realización concreta de la misma (es decir, crear un vector con los valores deseados para la x y para la y).

Además, y aquí radica una de las características principales de la programación orientada a objetos:

- Asociado a la plantilla de datos colocaremos el código que los manipula (por ejemplo, el código correspondiente a las operaciones sobre vectores). Estos elementos funcionales se denomina **métodos** de la clase.

8. Uso de objetos y clases

De la misma manera que una función puede verse como una plantilla de instrucciones que pueden ejecutarse para diferentes valores de los parámetros, podemos ver una clase como una plantilla de datos que puede instanciarse para diferentes valores de los parámetros. A cada una de las instanciaciones se las llama objetos.
Vamos a ver cómo representaremos cada una de la operaciones y, posteriormente, veremos cómo definir la clase:

Instanciación

De modo similar a la llamada a una función, lo que hemos de proporcionar para instanciar una clase es:

- el nombre de la clase que queremos instanciar (que denominaremos Vector2D)
- los valores iniciales para los componentes de la clase,

y lo que obtendríamos serían referencias a los objetos creados (ya que luego los querremos usar en el resto del programa).
La sintaxis de Java para ello será:

[2] En inglés fields.. Aunque su uso en castellano no esté generalizado he optado por seguir lo más fielmente posible las denominaciones usadas en el estándar de Java.

```
1 Vector2D pos = new Vector2D(10.0, 20.0);
2 Vector2D vel = new Vector2D(0.0, -5.0);
3 Vector2D accel = new Vector2D(0.2, 0.8);
```

A partir de ahora pos, vel y accel se referirán a tres objetos de la clase Vector2D.

Operaciones sobre objetos

Para actualizar los valores que contienen cada uno de los objetos usaremos los métodos definidos dentro de la clase. Para actualizar la velocidad y posición, tendremos un método llamado accumulate y para calcular la magnitud de la velocidad otro llamado magnitude.
El código que usaría esos métodos en Java sería:

```
4 vel.accumulate(accel);
5 pos.accumulate(vel);
6 double speed = vel.magnitude();
```

Invocación de métodos como paso de mensajes

Para entender la sintaxis de las llamada a métodos, es útil pensar en que, en vez de tratarse de simples llamadas a funciones, se trata de enviar un mensaje a un objeto.
Es decir, en una llamada como

```
vel.accumulate(accel)
```

debe interpretarse como:

- vel indica el objeto receptor del mensaje
- accumulate es el mensaje, que le dice al objeto referenciado por vel que éste ejecute el código del método accumulate
- accel es el parámetro que necesita vel para poder realizar lo que se le indique

Fijaos es que como ya hemos indicado que vel es el receptor del mensaje, ya no hemos de indicarlo otra vez como un parámetro del método.
En términos coloquiales sería la diferencia entre decir dirigiéndome a un grupo de personas

```
// Es en general por lo que he de indicar quién
grupo.queSeLevante(juan);
```

o, mirando fijamente a juan

```
// El receptor es juan, así que ya no lo indico
juan.levántate();
```

9. Definición de clases en Java

Mostraremos primero el código en Java de la clase Vector2D y posteriormente comentaremos cómo cumple con los criterios que habíamos demandado sobre la abstracción de datos:

```
 1 public class Vector2D {
 2
 3   private double x;
 4   private double y;
 5
 6   public Vector2D(double xInit, double yInit) {
 7     x = xInit;
 8     y = yInit;
 9   }
10
11   public void add(Vector2D vect) {
12     x = x + vect.x;
13     y = y + vect.y;
14   }
15
16   public double magnitude() {
17     return Math.sqrt( x*x + y*y );
18   }
19 }
```

Contenido de la definición:

- línea 1: la clase tiene un nombre. Recordad que la definición de esta clase deberá estar en el archivo Vector2D.java ya que para las clases públicas en java el nombre de la clase ha de coincidir con el nombre del archivo.
- líneas 3 y 4: cada objeto de la clase agrupará dos componentes de tipo double llamados x e y. Dichas componentes se denominan **variables de instancia**[3] ya que cada instancia de la clase tendrá una x y una y diferentes. El calificador private, que se comenta detalladamente más adelante, indica que solamente queremos que los valores de x e y puedan accederse desde dentro de la clase.
- líneas 6 a 9: esta definición de método que se llama igual que la clase y que no tiene indicación de tipo retornado, es lo que se conoce como **constructor de la clase** y su propósito es ejecutar las instrucciones que queremos que se ejecuten cuando alguien crea una instancia de la clase (haciendo **new**).

[3] En inglés *instance variable* o *instance field*.

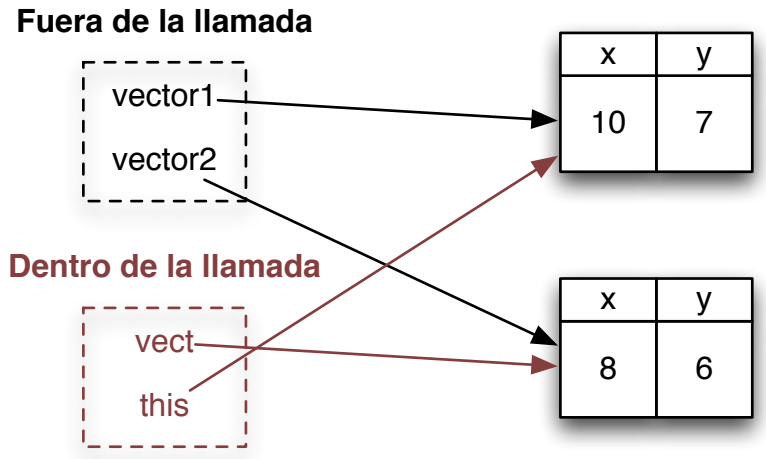
- líneas 11 a 14: sobre cada instancia de la clase se podrá invocar el método add, que necesitará de un parámetro adicional de tipo Vector2D. Dentro del método podremos acceder a la x e y del objeto receptor, así como de los componentes del parámetro vect (que denominaremos vect.x y vect.y). Fijaos en que el resultado del método es void ya que no devuelve resultado alguno pues su efecto se produce al actualizar los valores de las coordenadas del objeto receptor.
- líneas 17 a 20: definición del método magnitude que no requiere de ningún parámetro adicional para llevar a cabo su tarea ya que solamente necesita de los valores de los atributos del objeto. Tanto este método como el anterior se denominan **métodos de instancia** ya que son métodos que para aplicarse necesitan de una instancia de la clase (fijaos en que ambos métodos necesitan acceder a los valores de x e y de la instancia sobre la que se aplican).

10. La referencia especial `this`

¿Qué pasa si dentro de un método de instancia queremos referirnos al objeto sobre el cual estamos aplicando el método? Fijaos en que dicho objeto no pertenece a la lista de parámetros de la función. Para poder hacerlo, Java crea una referencia especial llamada `this`, que referencia al objeto sobre el que se está aplicando el método. Por ejemplo, si hacemos:

```
Vector2D vector1 = new Vector2D(10, 7);
Vector2D vector2 = new Vector2D(8, 6);
vector1.add(vector2);
```

dentro de la llamada al método `add`, tendremos que `vect` se refiere al mismo objeto que `vector2` (es el valor del parámetro) y `this` se refiere al mismo objeto que `vector1` (es el receptor del método).



Por ello, los métodos presentados anteriormente podrían haberse implementado como:

```
1 public class Vector2D {
2
3   private double x;
4   private double y;
5
6   public Vector2D(double x, double y) {
7     this.x = x;
8     this.y = y;
9   }
10
11  public void add(Vector2D vect) {
12    this.x = this.x + vect.x;
13    this.y = this.y + vect.y;
14  }
15
16  public double magnitude() {
17    return Math.sqrt( this.x*this.x + this.y*this.y );
18  }
19 }
```

Java, cuando dentro de un método de la clase accedemos a un campo usando su nombre, por ejemplo x, asume que estamos accediendo al valor de ese campo en el objeto sobre el que se aplica el método, es decir, this.x por lo que no es obligatorio anteponer explícitamente this. Si ponerlo os resulta más claro, no hay inconveniente en hacerlo (de hecho es lo que haremos en nuestros primeros ejemplos).

11. Encapsulamiento y niveles de acceso

Hasta el momento, como la única clase que definíamos era la clase que representa el programa principal, no nos planteábamos qué elementos definidos en la clase queríamos que fueran accesible desde fuera (ya que el único código que escribíamos estaba en esta única clase). Es por ello que, para no introducir más conceptos y simplificar, tanto las "constantes" como todos los métodos los definimos como public.
Ahora la situación ha cambiado, ya que nuestro código estará repartido entre varias clases, y deberemos decidir qué partes queremos que puedan usar exclusivamente desde dentro de la clase (private) y qué partes se puedan usar desde fuera (public).
En general, para que las clases sean lo más independientes entre sí interesa que el nivel de acceso sea el más bajo posible, es decir,

solamente hacer público aquello que es estrictamente necesario que se conozca desde fuera para poder usar la clase.
Es por ello que definiremos las variables de instancia como privadas. En caso de que sus valores necesiten ser conocidos desde fuera, definiríamos un método de la clase para obtener dicho valor. Por ejemplo, si quisiéramos que desde fuera se pudiera conocer el valor de la abscisa (coordenada horizontal) y de la ordenada (coordenada vertical), podríamos añadir los métodos (denominados **getters** o **métodos get**):

```
1 public class Vector2D {
2   ...
3   public double getX() {
4     return this.x;
5   }
6
7   public double getY() {
8     return this.y;
9   }
10 }
```

Fijaos en varios aspectos:

- con los métodos get solamente puedo leer el valor de una propiedad, no modificarla
- el nombre del método no tiene porqué coincidir con el de la variable de instancia, pero es práctica habitual denominarlos:
  `get + nombre de la propiedad`
- pero como el nombre de las variables de instancia es privado, también los podría haber llamado `getAbcissa` y `getOrdinate`.

Si además de dejar que los valores puedan leerse, quiero que puedan ser modificados, añadiré métodos a la clase. Dichos métodos, denominados **setters** o métodos **set**, que en nuestro caso quedarían como:

```
1 public class Vector2D {
2   ...
3   public void setX(double x) {
4     this.x = x;
5   }
6
7   public void setY(double y) {
8     this.y = y;
9   }
10 }
```

Eso sí, si hubiésemos denominado `getAbcissa` y `getOrdinate` a los métodos get, denominaríamos `setAbcissa` y `setOrdinate` a los métodos set asociados.
En resumen, para las variables de instancia tendremos tres posibilidades:

private	Desde fuera de la clase no se puede acceder.
private + getter	Desde fuera de la clase solamente se puede leer el valor.
private + getter + setter	Desde fuera de la clase se puede tanto leer como modificar el valor.

E intentaremos mantener siempre el nivel de acceso más limitado para cada caso.

12. Un caso especial: miembros estáticos

Siguiendo con el ejemplo de la clase `Vector2D`, fijaos en la situación siguiente: quiero sumar dos vectores pero el resultado quiero dejarlo en un tercer vector. ¿Cómo puedo hacerlo? Dadas las operaciones que tengo, debería hacer:

```
Vector2D result = new Vector2D(0, 0);
result.add(vector1);
result.add(vector2);
```

cosa que resulta, cuando menos, incómoda. Lo que nos gustaría es disponer de una operación tal que, en vez de modificar el vector sobre el que se aplica, crease un nuevo vector a partir de los parámetros que se le pasan (es decir, un método que funcionase como las funciones o acciones de C).
Para este tipo de métodos, que han de poder manipular los datos internos de un tipo, pero que no se aplican sobre una instancia de la clase, Java incorpora el concepto de **métodos estáticos** de una clase.
En el caso que nos ocupa, implementaríamos el método estático add de la siguiente manera[4]:

```
1 public class Vector2D {
2   private int x;
```

[4] Como ya hemos visto anteriormente Java nos deja usar el mismo nombre para métodos diferentes. En este caso, usamos el mismo nombre para un método no estático de la clase con un solo parámetro, y para un método estático de la clase que tiene dos parámetros.

```
3   private int y;
4   ...
5   public static Vector2D add(Vector2D v1, Vector2D v2) {
6     return new Vector2D(v1.x + v2.x, v1.y + v2.y);
7   }
8   ...
9 }
```

Método que ahora podríamos utilizar de la siguiente manera:

```
Vector2D result = Vector2D.add(vector1, vector2);
```

Un par de consideraciones:

- al invocar un método estático desde fuera de la clase, en vez de usar un objeto como receptor, usamos la clase. Es por ello que los métodos estáticos también se conoce como **métodos de clase**.
- como el método estático pertenece a la clase, si recibe parámetros que son instancias de la clase, puede acceder directamente a su parte privada.
- el efecto del método no consiste en modificar el objeto receptor sino en devolver un nuevo objeto calculado a partir de los parámetros que se le pasan. Es por ello que los métodos estáticos son casi equivalentes a las funciones y métodos de lenguajes no orientados a objetos.
- Como un método estático no tiene un objeto receptor, no tiene ninguna referencia mágica a this.

Resumiendo:

No estáticos o de instancia	Se aplican sobre un objeto denominado receptor, es decir: objeto.metodo(params);	Desde dentro del método puedo acceder a las variables de instancia del objeto receptor. Existe una constante especial denominada **this** que referencia al objeto sobre el que se está aplicando el método.
Estáticos o de clase	El receptor es la propia clase, es decir: clase.metodo(params);	Al no haber objeto receptor no puede acceder a las variables de instancia.

Campos estáticos de una clase

De la modo similar a los métodos de clase, podemos definir campos tal que, en vez de pertenecer a cada instancia concreta de la clase, pertenezcan a la clase en general (o visto de otro modo, que sean compartidos por todas las instancias de la clase).
Por ejemplo, si en una clase defino un campo para representar una constante, si el campo no es estático, cada instancia de clase tendría una copia de la constante, ¡pero todas valdrían igual! En cambio, si la definimos como un campo estático, únicamente existirá una copia y será compartida por todas las instancias de la clase.
Otro ejemplo de uso sería el siguiente en el que mantenemos una variable estática para asignar identificadores diferentes para cada una de las instancias que creamos de la clase. Dicho número de instancias es claramente una propiedad de toda la clase y no de cada una de las instancias individualmente. En cambio el identificador será particular a cada una de ellas (ya que cada una tendrá el suyo).

```
1 public class Bicycle {
2   private static int numberOfBicycles = 0;
3   private int cadence;
4   private int gear;
5   private int speed;
6   private int id;
7
8   public Bicycle(int startCadence,
9                  int startSpeed,
10                 int startGear){
11    gear = startGear;
12    cadence = startCadence;
13    speed = startSpeed;
14    // increment number of Bicycles and assign ID number
15    numberOfBicycles = numberOfBicycles + 1;
16    id = numberOfBicycles;
17  }
18
19  public int getID() {
20    return id;
21  }
22  .....
23 }
```

Completando la clase Vector2D

```
1 public class Vector2D {
```

```
 2
 3  private double x;
 4  private double y;
 5
 6  public Vector2D(double x, double y) {
 7    this.x = x;
 8    this.y = y;
 9  }
10
11  public double getX() {
12    return this.x;
13  }
14
15  public void setX(double x) {
16    this.x = x;
17  }
18
19  public double getY() {
20    return this.y;
21  }
22
23  public void setY(double y) {
24    this.y = y;
25  }
26
27  public void add(Vector2D vect) {
28    this.x += vect.x;
29    this.y += vect.y;
30  }
31
32  public void sub(Vector2D vect) {
33    this.x -= vect.x;
34    this.y -= vect.y;
35  }
36
37  public void mult(double scale) {
38    this.x *= scale;
39    this.y *= scale;
40  }
41
42  public void div(double scale) {
43    this.x /= scale;
44    this.y /= scale;
45  }
46
47  public double magnitude() {
```

```
48    return Math.sqrt(this.x*this.x + this.y*this.y);
49  }
50
51  public void normalize() {
52    double m = this.magnitude();
53    this.div(m);
54  }
55
56  public void limit(double max) {
57    if ( this.magnitude() > max ) {
58      this.normalize();
59      this.mult(max);
60    }
61  }
62
63  public double distance(Vector2D vect) {
64    Vector2D diff = sub(this, vect);
65    return Math.sqrt(diff.magnitude());
66  }
67
68  public double dot(Vector2D vect) {
69    return this.x * vect.x + this.y * vect.y;
70  }
71
72  public static Vector2D add(Vector2D vect1,
73                             Vector2D vect2) {
74    return new Vector2D(vect1.x + vect2.x,
75                        vect1.y + vect2.y);
76  }
77
78  public static Vector2D sub(Vector2D vect1,
79                             Vector2D vect2) {
80    return new Vector2D(vect1.x - vect2.x,
81                        vect1.y - vect2.y);
82  }
83
84  public static Vector2D mult(Vector2D vect,
85                              double scale) {
86    return new Vector2D(vect.x*scale, vect.y*scale);
87  }
88
89  public static Vector2D div(Vector2D vect,
90                             double scale) {
91    return new Vector2D(vect.x/scale, vect.y/scale);
92  }
93 }
```

Algunos comentarios adicionales sobre la clase `Vector2D` y su implementación:

- Uso las denomidas **asignaciones aumentadas**. Por ejemplo:

  ```
  this.x += vect.x;
  ```

  que es equivalente a

  ```
  this.x = this.x + vect.x;
  ```

- Líneas 63 a 66: en la implementación de distance uso la referencia this para referirme a la instancia del `Vector2D` sobre la que estoy trabajando.

13. Otras clases con métodos estáticos útiles

La clase `Math`

La clase `Math` agrupa algunas constantes y métodos estáticos que implementan p.e. funciones trigonométricas, logaritmos, etc.
Como muchas funciones están disponibles tanto para varios tipos de parámetros (`int`, `long`, `float` o `double`), no indicaremos su tipo.
Podéis consultar la documentación de Java para ver todas las funciones disponibles y sus detalles.

Math.E Valor de la constante e
Math.PI Valor de π
Math.abs(x) Devuelve el valor absoluto de x
Math.min(x, y) Devuelve el menor entre x e y
Math.max(x, y) Devuelve el mayor entre x e y
Math.sqrt(x) Devuelve la raíz cuadrada de x
Math.log(x) Devuelve el logaritmo natural (base e) del número x
Math.exp(x) Devuelve el valor de e^x
Math.pow(x, y) Devuelve el valor de x^y
Math.sin(theta) Devuelve el valor del seno del ángulo theta expresado en radianes

Math.sin(theta) Devuelve el valor del coseno del ángulo theta expresado en radianes
Math.sin(theta) Devuelve el valor de la tangente del ángulo theta expresado en radianes
Math.asin(x) Devuelve el ángulo (en radianes)cuyo seno es x
Math.acos(x) Devuelve el ángulo (en radianes)cuyo coseno es x
Math.atan(x) Devuelve el ángulo (en radianes)cuya tangente es x
Math.toRadians(degrees) Convierte un ángulo de grados sexagesimales a radianes
Math.toDegrees(radians) Convierte un ángulo de radianes a grados sexagesimales

La clase Character

Como ya se ha indicado anteriormente en la sección sobre Strings, en la clase Character, hay métodos estáticos para manipular cómodamente los caracteres. Algunos de estos métodos son:

boolean Character.isDigit(char ch) Determina si el carácter ch corresponde a un dígito
boolean Character.isLetter(char ch) Determina si el carácter ch corresponde a una letra
boolean Character.isLetterOrDigit(char ch) Determina si el carácter ch corresponde a un dígito o una letra
boolean Character.isLowerCase(char ch) Determina si el carácter ch es una minúscula
boolean Character.isUpperCase(char ch) Determina si el carácter ch es una mayúscula
boolean Character.isWhitespace(char ch) Determina si el carácter ch es espacio en blanco, es decir, es invisible cuando se escribe (espacios, tabuladores)
char Character.toLowerCase(char ch) Retorna el equivalente en minúsculas al carácter ch si éste existe (si no, se retorna ch)
char Character.toUpperCase(char ch) Retorna el equivalente en mayúsculas al carácter ch si éste existe (si no, se retorna ch)

14. La clase `acm.util.RandomGenerator`

Una clase que puede ser útil para hacer animaciones gráficas (entre otras muchas cosas) es la clase `RandomGenerator` que, como su nombre indica, es un generador de valores aleatorios.
La única cosa que diremos sobre el método de generarlos es que lo que obtenemos son números que se denominan **pseudoaleatorios**, es decir, que pueden usarse como si lo fueran, pero realmente no lo son. La razón es que, si pueden calcularse con un método, es que realmente no son aleatorios, pues puedo usar ese método para predecirlos.
Los métodos principales de esta clase son:

static RandomGenerator getInstance() Retorna una instancia de la clase `RandomGenerator` para que sea compartida en todo el programa.
int nextInt(int n) Devuelve un entero e al azar en el intervalo 0<=e<n
int nextInt(int low, int high) Devuelve un entero al azar en el intervalo low<=e<=high
double nextDouble() Devuelve un double d al azar en el intervalo 0<=d<1
double nextDouble(double low, double high) Devuelve un double d al azar en el intervalo low<=d<high
boolean nextBoolean() Devuelve un booleano que es true con probabilidad 0,5 (50%)
boolean nextBoolean(double p) Devuelve un booleano que es true con probabilidad p, que debe estar entre 0 y 1
Color nextColor() Devuelve un color al azar
void setSeed(long seed) Indica la semilla a usar como punto inicial de la secuencia pseudoaleatoria.

getInstance()

Por razones técnicas en las que no entraremos, cuando en un programa utilizamos un generador de números aleatorios, nos interesa usar el mismo generador en todas las partes del programa. Por ello, en vez de usar el constructor de la clase `RandomGenerator` y hacer

```
RandomGenerator rgen = new RandomGenerator();
```

lo que haremos es

```
RandomGenerator rgen = RandomGenerator.getInstance();
```

y no importa cuantas veces llamemos a ese método ya que siempre retornará una referencia a la misma instancia de `RandomGenerator`.

setSeed(long seed)

Supongamos que realmente dispusiéramos de la capacidad de crear secuencias al azar para usarlas en nuestro programa. Si así fuera, la probabilidad de obtener dos veces la misma secuencia sería prácticamente nula, por lo que realizar dos ejecuciones iguales del programa sería casi imposible. ¿Veis el problema que ello produciría? Imaginad por un momento que en una ejecución obtenéis un resultado erróneo que os lleva a modificar el programa. ¿Cómo podréis volver a ejecutar el programa para ver si ahora el problema está solucionado? No podréis.

Para ello, un generador de números pseudoaleatorios tiene la posibilidad de configurarse con lo que se conoce como una **semilla**, a partir de la cual se genera toda la secuencia. Si inicializamos la semilla con el mismo valor, obtendremos la misma secuencia pseudoaleatoria. Por ello, normalmente en el programa no lo haremos, salvo que estemos buscando errores y nos interese repetir la misma ejecución.

Un ejemplo de uso: el juego Craps

```
 1 /*
 2  * File: Craps.java
 3  * ----------------
 4  * This program plays the casino game of Craps.
 5  * At the beginning of the game, the player rolls
 6  * a pair of dice and computes the total.  If the
 7  * total is 2, 3, or 12 (called "craps"), the player
 8  * loses.  If the total is 7 or 11 (called a "natural"),
 9  * the player wins.  If the total is any other number,
10  * that number becomes the "point."  From here, the
11  * player keeps rolling the dice until (a) the point
12  * comes up again, in which case the player wins or
13  * (b) a 7 appears, in which case the player loses.
14  * The numbers 2, 3, 11 and 12 no longer have special
15  * significance after the first roll.
16  */
17
18 import acm.program.*;
19 import acm.util.*;
20
21 public class Craps extends ConsoleProgram {
22
23   public void run() {
```

```
24      int total = rollTwoDice();
25      if (total == 7 || total == 11) {
26              println("That's a natural.  You win.");
27         } else if (total == 2 || total == 3 || total == 12) {
28              println("That's craps.  You lose.");
29         } else {
30              int point = total;
31              println("Your point is " + point + ".");
32              while (true) {
33                   total = rollTwoDice();
34                   if (total == point) {
35                        println("Your point. You win.");
36                        break;
37                   } else if (total == 7) {
38                        println("That's a 7.  You lose.");
                     break;
39                   }
40              }
41         }
42    }
43
44   /* Rolls two dice and returns their sum. */
45    private int rollTwoDice() {
46      int d1 = rgen.nextInt(1, 6);
47      int d2 = rgen.nextInt(1, 6);
48      int total = d1 + d2;
49      println("Rolling dice: " + d1 + " + " + d2 + " = " +
  total);
50      return total;
51    }
52
53   /* Private instance variables */
54    private RandomGenerator rgen =
  RandomGenerator.getInstance();
55
56 }
57
```

15. Sobre el estilo de programación

Apliquemos estos conceptos sobre visibilidad (público o privado) y ámbito (de instancia o de clase) a un caso conocido: el de cálculo de la suma de un array.

Veremos dos versiones:

- Una con todos los métodos auxiliares estáticos (más parecida a lo que escribiríamos en C) que reciben parámetros y devuelven resultados

- Una con los métodos no estáticos (más natural en el caso de Java) que manipulan variables de instancia.

Versión procedimental (similar a C)

```
 1 /* ArraySum.java
 2  * -------------
 3  * This programs fills an array of máximum size of
 4  * 10 and sums its contents until the first zero is
 5  * found
 6  */
 7
 8 import acm.program.ConsoleProgram;
 9
10 public class ArraySum extends ConsoleProgram {
11
12   private static int MAX = 10;
13
14   private static int[] readArray() {
15     int[] nums = new int[MAX];
16     ...
17   }
18
19   private static int sumArray(int[] numbers) {
20     ...
21   }
22
23   public void run() {
24     int[] numbers = readArray();
25     int sum = sumArray(numbers);
26     println("The array sum is " + sum);
27   }
28 }
```

Veamos los usos de public y private para los elementos definidos dentro de la clase (el uso de public tiene un significado similar, pero de momento y para simplificar, todas las clases que definiremos serán public):

- Línea 12: MAX es una "constante" (de hecho es una variable de instancia) que se usará dentro de la clase en el método readArray que es de la propia clase, por lo que la podemos definir como privada y estática.
- Líneas 14 y 19: ambos métodos son auxiliares del programa principal y solamente se usan desde el método run que

pertenece a la propia clase, por lo que pueden definirse como privados. Como

- Línea 23: el método que representa el programa principal, debe definirse como público, ya que es llamado desde fuera de la clase (por el intérprete de la máquina virtual de java). De hecho, el mecanismo de invocación, realiza unas instrucciones similares a:

```
programa = new ArraySum();
programa.run();
```

por lo que run ha de poder ser invocada desde fuera de la clase y, por tanto, ha de ser pública.

Versión (más) orientada a objetos

```
1 /* ArraySum.java
2  * -------------
3  * This programs fills an array of máximum size of
4  * 10 and sums its contents until the first zero is
5  * found
6  */
7
8 import acm.program.ConsoleProgram;
9
10 public class ArraySum extends ConsoleProgram {
11
12   private static int MAX = 10;
13   private int[] nums = new int[MAX];
14
15   private void readArray() {
16     ...
17   }
18
19   private int sumArray() {
20     ...
21   }
22
23   public void run() {
24     readArray();
25     int sum = sumArray();
26     println("The array sum is " + sum);
27   }
28 }
```

16. Bibliografía

- Eric S.Roberts, The Art & Science of Java, Addison-Wesley (2008).
- The Java Tutorials. (última consulta 3 de marzo de 2011).
- Kathy Sierra & Bert Bates, Head First Java, O'Reilly (2003).