TEMA 6: PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA (I)

Las ***clases*** son el centro de la ***Programación Orientada a Objetos***. Recordemos que algunos de los conceptos más importantes de la POO son los siguientes:

1. ***Encapsulación***. Las clases pueden ser declaradas como públicas (***public***) y como ***package*** (accesibles sólo para otras clases del ***package***). Las variables miembro y los métodos pueden ser ***public***, ***private***, ***protected*** y ***package***. De esta forma se puede controlar el acceso y evitar un uso inadecuado.
2. ***Herencia***. Una clase puede derivar de otra (***extends***), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede ***añadir*** nuevas variables y métodos y/o ***redefinir*** las variables y métodos heredados.
3. ***Polimorfismo***. Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface pueden tratarse de una forma general e individualizada, al mismo tiempo. Esto facilita la programación y el mantenimiento del código.

En este tema se presentan las ***clases*** y las ***interfaces*** tal como están implementadas en el lenguaje ***Java***.

1. CONCEPTOS DE CLASE

Una *clase* es una agrupación de ***datos*** (variables o campos) y de ***funciones*** (métodos) que operan sobre esos datos.

```
[public] class Classname {
   // definición de variables y métodos
   ...
}
```

donde la palabra ***public*** es opcional: si no se pone, la clase sólo es visible para las demás clases del ***package***. Todos los métodos y variables deben ser definidos dentro del ***bloque*** {...} de la clase. En definitiva una *clase* es una plantilla para crear objetos que tienen los mismos atributos y responden a los mismos mensajes.

Un ***objeto*** (en inglés, ***instance***) es un ejemplar concreto de una clase. Las ***clases*** son como tipos de variables, mientras que los ***objetos*** son como variables concretas de un tipo determinado.

```
Classname unObjeto;
Classname otroObjeto;
```

A continuación se enumeran algunas características importantes de las clases:

1. Todas las variables y funciones de ***Java*** deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (***extends***), hereda todas sus variables y métodos.
3. ***Java*** tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.
4. Una clase sólo puede heredar de una única clase (en ***Java*** no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de ***Object***. La clase ***Object*** es la base de toda la jerarquía de clases de ***Java***.

5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase ***public***. Este fichero se debe llamar como la clase ***public*** que contiene con extensión ****.java***. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.

6. Si una clase contenida en un fichero no es ***public***, no es necesario que el fichero se llame como la clase.

7. Los métodos de una clase pueden referirse de modo global al ***objeto*** de esa clase al que se aplican por medio de la referencia ***this***.

8. Las clases se pueden agrupar en ***packages***, introduciendo una línea al comienzo del fichero (***package packageName;***). Esta agrupación en ***packages*** está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

2. EJEMPLO DE DEFINICIÓN DE UNA CLASE

A continuación se reproduce como ejemplo la clase **Circulo**. En este ejemplo se ve cómo dentro de la clase se definen las variables miembro y los métodos (cuyos nombres se han resaltado en negrita). Dichas variables y métodos pueden ser ***de objeto*** o ***de clase*** (***static***). Se puede ver también cómo el nombre del fichero coincide con el de la clase ***public*** con la extensión ****.java***.

```
// fichero Circulo.java
public class Circulo extends Geometria
{
	// VARIABLES
	// Variable de clase
	private static int numCirculos = 0;

	// Variable de clase constante
	public static final double PI=3.14159265358979323846;

	// Variables de instancia u objeto
	private double x, y, r;


	// CONSTRUCTORES
	public Circulo(double x, double y, double r) {
		this.x=x; this.y=y; this.r=r;
		numCirculos++;
	}

	public Circulo(double r) { this(0.0, 0.0, r); }
	public Circulo(Circulo c) { this(c.x, c.y, c.r); }
	public Circulo() { this(0.0, 0.0, 1.0); }

	// METODOS DE OBJETO O FUNCIONES MIEMBRO
	// Método que calcula el perímetro de un círculo
	public double perimetro() { return 2.0 * PI * r; }

	// Método que calcula el área de un círculo
	public double area() { return PI * r * r; }

	// Método de objeto para comparar círculos
	public Circulo elMayor(Circulo c) {
		if (this.r>=c.r) return this;
		else return c;
	}

	// METODO DE CLASE
	// Método de clase para comparar círculos
	public static Circulo elMayor(Circulo c, Circulo d) {
		if (c.r>=d.r) return c;
		else return d;
	}
} // fin de la clase Circulo
```

3. VARIABLES MIEMBRO

A diferencia de la programación algorítmica clásica, que estaba centrada en las funciones, la programación orientada a objetos está ***centrada en los datos***. Una clase son unos ***datos*** y unos ***métodos*** que operan sobre esos datos.

3.1. Variables miembro de objeto o variables de instancia

Cada objeto, es decir cada ejemplar concreto de la clase, tiene su propia copia de las variables de instancia. Las variables de instancia de una clase (también llamadas ***campos***) pueden ser de ***tipos primitivos*** (*boolean*, *int*, *long*, *double*, …) o referencias a ***objetos*** de otra clase (*composición*).

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables de instancia de ***tipos primitivos*** se inicializan siempre de modo automático, incluso antes de llamar al ***constructor*** (***false*** para ***boolean***, la cadena vacía para ***char*** y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas en el constructor.

También pueden inicializarse explícitamente en la ***declaración***, como las variables locales, por medio de constantes o llamadas a métodos (esta inicialización no está permitida en C++). Por ejemplo,

```
long nDatos = 100;
```

Las variables de instancia se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada ***objeto*** que se crea de una clase tiene ***su propia copia*** de las variables de instancia. Por ejemplo, cada objeto de la clase ***Circulo*** tiene sus propias coordenadas del centro ***x*** e ***y***, y su propio valor del radio ***r***.

Las variables de instancia pueden ir precedidas en su declaración por uno de los modificadores de acceso: ***public***, ***private***, ***protected*** y ***package*** (que es el valor por defecto y puede omitirse). Junto con los modificadores de acceso de la clase (***public*** y ***package***), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables de instancia. Todo esto se verá en detalle un poco más adelante.

3.1.1. Métodos get y set

En Java cada vez es más habitual la práctica de ocultar las propiedades (*variables de instancia*) y trabajar exclusivamente con los métodos. La razón es que no es recomendable que las propiedades sean visibles desde fuera de la clase, por ello se declaran con una visibilidad **private** (o **protected**).

Siendo así ¿cómo pueden las otras clases modificar el valor de una propiedad? Mediante métodos que permitan la lectura y escritura de esas propiedades, son los métodos **get** (obtener) y **set** (ajustar).

Los get sirven para leer el valor de un atributo, nunca llevan parámetros y el nombre del método es la palabra ***get*** seguida del nombre de la propiedad (por ejemplo ***getEdad***).

Los ***set*** permiten variar el valor de una propiedad. Por lo que como parámetro reciben el nuevo valor y son métodos siempre de tipo **void**.

Por ejemplo si en una clase se ha definido una variable de instancia para almacenar la edad de una persona, esa misma clase habitualmente dispondrá de dos métodos para poder obtener el contenido de esa variables, y para poder modificarla. Es decir, tendríamos:

```
// Variables de instancia
prívate byte edad;

// Métodos set y get
public byte getEdad() {return edad;}
```

```
public void setEdad(byte e) {edad = e;}
```

3.2. Variables miembro de clase (static).

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama ***variables de clase*** o variables ***static***. Las variables ***static*** se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo ***PI*** en la clase ***Circulo***) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como ***numCirculos*** en la clase ***Circulo***).

Las variables de clase son lo más parecido que ***Java*** tiene a las ***variables globales*** de C/C++.

Las variables de clase se crean anteponiendo la palabra ***static*** a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, ***Circulo.numCirculos*** es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembro ***static*** se inicializan con los valores por defecto (***false*** para ***boolean***, la cadena vacía para ***char*** y cero para los tipos numéricos) para los tipos primitivos, y con ***null*** si es una referencia.

Las variables miembro ***static*** se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método ***static*** o en cuanto se utiliza una variable ***static*** de dicha clase. Lo importante es que las variables miembro ***static*** se inicializan siempre antes que cualquier objeto de la clase.

4. VARIABLES FINALES

Una variable de un tipo primitivo declarada como ***final*** no puede cambiar su valor a lo largo de la ejecución del programa. Puede ser considerada como una ***constante***, y equivale a la palabra ***const*** de C/C++.

Java permite separar la ***definición*** de la ***inicialización*** de una variable ***final***. La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos. La variable ***final*** así definida es ***constante*** (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Además de las variables miembro, también las variables locales y los propios argumentos de un método pueden ser declarados ***final***.

Declarar como ***final*** un objeto miembro de una clase hace ***constante*** la ***referencia***, pero no el propio objeto, que puede ser modificado a través de otra referencia. En ***Java*** no es posible hacer que un objeto sea constante.

5. MÉTODOS (FUNCIONES MIEMBRO)

En Java toda la lógica de programación (*algoritmos*) está agrupada en funciones o métodos. Un método es un subprograma, es decir un bloque de código que tiene un nombre, recibe unos parámetros o argumentos (opcionalmente), contiene sentencias o instrucciones para realizar algo (opcionalmente) y devuelve un valor de algún Tipo conocido (opcionalmente).

Al realizar una llamada a un método (también se le llama enviar un mensaje), el control del programa pasa a ese método y lo mantendrá hasta que el método finalice. La mayoría de métodos devuelven

un resultado (gracias a la palabra **return**), por ello cuando se define el método hay que indicar el tipo de datos al que pertenece el resultado del mismo. Si el método no devuelve ningún resultado se indica como tipo de datos a devolver el tipo **void** (void significa vacío).

Los métodos son los equivalentes de las funciones y procedimientos de la programación modular clásica. De hecho un método es una función, sólo que esa función está asociada a una clase, por lo que se convierte en una operación que esa clase es capaz de realizar.

Cuando una clase ya tiene definido sus métodos, es posible invocarles utilizando los objetos definidos de esa clase. En esa invocación, se deben indicar los **parámetros** (o **argumentos**) que cada método requiere para poder realizar su labor.

A la hora de crear un nuevo método se deben tener en cuenta los siguientes criterios que nos ayudarán a modularizar correctamente nuestras aplicaciones:

- Minimizar el acoplamiento (los métodos deben ser lo más independientes posible). Es decir, si necesito calcular las raíces de una ecuación de segundo grado, no tendría sentido que parte de esos cálculos los hiciese en un método y otra parte en otro, ya que ambos métodos tendrían que estar continuamente intercambiando información entre ellos.
- Maximizar la cohesión (relación entre las diferentes partes internas de un módulo). En este caso, tampoco tendría sentido que hiciese un método que calcule la raíz de número y además el factorial de este, habría sido preferible hacer dos métodos distintos.

Cuando un método devuelve un valor determinado, es decir, no es void, en los lenguajes no orientados a objetos se les llama función. En estos casos, la llamada al método no puede constituir por sí sola una sentencia del algoritmo o programa llamante, sino que debe aparecer dentro de alguna sentencia del algoritmo llamante en la que el valor devuelto por el métod (al terminar la ejecución), sea "utilizado" de alguna forma, por ejemplo a la derecha de una sentencia de asignación.

Ejemplo:
```
int minimo(int x, int y)
{
   if (x < y)
      return x;
   else
      return y;
}
```

En el algoritmo que llama se podría poner: `resultado = minimo(a,b)`, siendo *resultado*, *a*, *b* variables del algoritmo llamante.

Los métodos que devuelven un valor booleano o lógico son particularmente útiles, convirtiéndose en las expresiones lógicas de decisión para la sentencia condicionales y en las condiciones de control para los bucles.

Aquellos métodos que no devuelven ningún valor (*void*), conocidos como procedimientos en otros lenguajes, si constituyen por sí solos una acción o sentencia en el cuerpo del algoritmo "llamante".

Ejemplo:
```
void minimo(int x, int y)
{
   if (x < y) entonces
      System.out.println("El menor es " + x);
   else
      System.out.println("El menor es " + y);
}
```

dentro del algoritmo principal o de otro subalgoritmo la llamada podría ser `minimo(a,b)`.

5.1. Parámetros formales y reales

La comunicación entre un método llamado y el método llamante se realiza mediante las ***variables de enlace, parámetros*** o ***argumentos*** y al proceso de emisión y recepción de datos y resultados mediante variables de enlace recibe el nombre de ***paso de parámetros***. Además en la POO, existe otra forma de comunicación que son los ***argumentos implícitos*** (se verán en el siguiente apartado).

En general:

- En la cabecera de definición de un método aparecen los ***parámetros formales***, que son nombres de variables con su tipo, separados por comas. Es decir, la sintaxis de la cabecera de un método será:

  [modificador_acceso] valor_retorno nombre_metodo>(*parámetros formales*)

  Ejemplo: `public float factorial(int num);`

- En la llamada al método estarán los ***parámetros reales*** (o actuales), que son expresiones (variables, constantes, etc.) separadas por comas.

  [nombreClase/nombreObjeto **.**]nombreMétodo(*parámetros reales*)

  Ejemplos:
  ```
  factorial(5); // es equivalente a this.factorial(5)
  Math.sqrt(81);
  c1.elMayor(c2);
  Circulo(c1,c2);
  ```

  Si el método llamado y llamante están en la misma clase, se puede invocar poniendo únicamente el nombre del método seguido de sus parámetros, en este caso es opcional colocar la palabra ***this*** y un punto delante del nombre del método en la llamada, con lo cual se aclararía que ambos métodos pertenecen a la misma clase.

  Si el método llamado y llamante están en la distintas clases, es obligatorio especificar el nombre de un objeto que pertenezca a la clase del método llamado, o bien el nombre de la clase, cuando especificar uno u otro se verá en los apartados siguientes.

- Los parámetros en los métodos son opcionales, pero si aparecen han de someterse a unas reglas:
  1. *El número de parámetros formales ha de ser igual al número de parámetros reales.*
  2. *El i-ésimo parámetro formal se corresponde con el i-ésimo parámetro real.*
  3. *El tipo del i-ésimo parámetro formal debe ser igual que el tipo del i-ésimo parámetro real.*
  4. *Los parámetros de un método pueden ser de cualquier tipo, al igual que cualquier variable.*
  5. *Los nombres de un parámetro formal y su correspondiente real pueden o no ser diferentes.*

En programación se denomina ***interfaz*** a la combinación de los parámetros formales definidos en el método y los parámetros actuales que se pasan en la llamada. Los errores en el uso del método se presentan fundamentalmente debido a una interfaz incorrecta entre el método llamante y el llamado.

Para diseñar la interfaz debemos considerar:

- ✓ ¿Qué información del programa llamante necesita conocer el método para poder trabajar correctamente?
- ✓ ¿Qué información producirá el método llamado que después sea necesitada en el algoritmo llamante?

5.2. Métodos de objeto

Como ya se ha dicho, los ***métodos*** son funciones definidas dentro de una clase. Salvo los métodos ***static*** o de clase, se aplican siempre a un objeto de la clase por medio del ***operador punto*** (.). Dicho objeto es su ***argumento implícito***. Los métodos pueden además tener otros ***argumentos explícitos*** que van entre paréntesis, a continuación del nombre del método.

La primera línea de la definición de un método se llama ***declaración*** o ***header***; el código comprendido entre las ***llaves*** {…} es el ***cuerpo*** o ***body*** del método. Considérese el siguiente método tomado de la clase ***Circulo***:

```
public Circulo elMayor(Circulo c) {  // header y comienzo del método
     if (this.r>=c.r)                // body
        return this;                 // body
     else                            // body
        return c;                    // body
}                                    // final del método
```

El ***header*** consta del cualificador de acceso (***public***, en este caso), del tipo del valor de retorno (***Circulo*** en este ejemplo, ***void*** si no tiene), del ***nombre de la función*** y de una lista de ***argumentos explícitos*** entre paréntesis, separados por comas. Si no hay argumentos explícitos se dejan los paréntesis vacíos.

La forma de llamar a un método de objeto es a través de un objeto que pertenezca a esa clase, por ejemplo, para calcular el área de un objeto de la clase ***Circulo*** llamado ***c1*** se escribe: **`c1.area();`**. En el caso de querer averiguar el mayor de dos círculos previamente creados (c1 y c2), la llamada sería `c1.elMayor(c2);`.

Los métodos tienen ***visibilidad directa*** de las variables miembro del objeto que es su ***argumento implícito***, es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia ***this***, de modo discrecional (como en el ejemplo anterior con ***this.r***) o si alguna variable local o argumento las oculta.

El ***valor de retorno*** puede ser un valor de un ***tipo primitivo*** o una ***referencia***. En cualquier caso no puede haber más que un único valor de retorno (que puede ser un objeto o un array). Se puede devolver también una referencia a un objeto por medio de un nombre de ***interface***. El objeto devuelto debe pertenecer a una clase que implemente esa interface.

Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.

Los métodos pueden definir ***variables locales***. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, **las variables locales no se inicializan por defecto**.

Si en el ***header*** del método se incluye la palabra ***native*** (Ej: public native void miMetodo();) no hay que incluir el código o implementación del método. Este código deberá estar en una librería dinámica (*Dynamic Link Library* o DLL). Estas librerías son ficheros de funciones compiladas normalmente en lenguajes distintos de ***Java*** (C, C++, Fortran, etc.). Es la forma de poder utilizar conjuntamente funciones realizadas en otros lenguajes desde código escrito en ***Java***.

Un método también puede declararse como ***synchronized*** (Ej: public synchronized double miMetodoSynch(){...}). Estos métodos tienen la particularidad de que sobre un objeto no pueden ejecutarse simultáneamente dos métodos que estén sincronizados.

5.3. Métodos sobrecargados (*overloaded*).

Al igual que C++, ***Java*** permite métodos ***sobrecargados*** (***overloaded***), es decir métodos distintos con ***el mismo nombre*** que se diferencian por el número y/o tipo de los argumentos. El ejemplo de la clase ***Circulo*** presenta dos métodos sobrecargados: los cuatro constructores y los dos métodos ***elMayor()***.

A la hora de llamar a un método sobrecargado, ***Java*** sigue unas reglas para determinar el método concreto que debe llamar:

1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
2. Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo *char* a *int*, *int* a *long*, *float* a *double*, etc.) y se llama el método correspondiente.
3. Si sólo existen métodos con argumentos de un tipo más amplio (por ejemplo, *long* en vez de *int*), el programador debe hacer un ***cast*** explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
4. El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la ***sobrecarga*** de métodos es la ***redefinición***. Una clase puede ***redefinir*** (***override***) un método heredado de una superclase. ***Redefinir*** un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la ***herencia***.

5.4. Paso de argumentos a métodos.

En ***Java*** los argumentos de los ***tipos primitivos*** se pasan siempre ***por valor***. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. La forma de modificar dentro de un método una variable de un tipo primitivo es incluirla como variable miembro de la clase y pasar como argumento una referencia a un objeto de dicha clase. Las ***referencias*** se pasan también ***por valor***, pero a través de ellas se pueden modificar los objetos referenciados.

En ***Java*** no se pueden pasar métodos como argumentos a otros métodos (en C/C++ se pueden pasar como argumentos punteros a función). Lo que se puede hacer en ***Java*** es pasar una referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.

Dentro de un método se pueden crear ***variables locales*** de los tipos primitivos o referencias. Estas variables locales dejan de existir al terminar la ejecución del método. Los parámetros formales de un método tienen categoría de variables locales del método.

Si un método devuelve ***this*** (es decir, un objeto de la clase) o una referencia a otro objeto, ese objeto puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así sucesivamente. En este caso aparecerán varios métodos en la misma sentencia unidos por el operador punto (.), por ejemplo,

```
String numeroComoString = ”8.978”;
float p = Float.valueOf(numeroComoString).floatValue();
```

donde el método ***valueOf(String)*** de la clase ***java.lang.Float*** devuelve un objeto de la clase ***Float*** sobre el que se aplica el método ***floatValue()***, que finalmente devuelve una variable primitiva de tipo ***float***. El ejemplo anterior se podía desdoblar en las siguientes sentencias:

```
String numeroComoString = ”8.978”;
Float f = Float.valueOf(numeroComoString);
float p = f.floatValue();
```

Obsérvese que se pueden encadenar varias llamadas a métodos por medio del operador punto (.) que, como todos los operadores de ***Java*** excepto los de asignación, se ejecuta de izquierda a derecha.

5.5. Métodos de clase (static).

Análogamente, puede también haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama ***métodos de clase*** o ***static***. Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia ***this***. Un ejemplo típico de métodos ***static*** son los métodos matemáticos de la clase ***java.lang.Math*** (*sin(), cos(), exp(), pow()*, etc.). De ordinario el argumento de estos métodos será de un tipo primitivo y se le pasará como argumento explícito. Estos métodos no tienen sentido como métodos de objeto.

Los métodos y variables de clase se crean anteponiendo la palabra ***static***. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase (por ejemplo, ***Math.sin(ang)***, para calcular el seno de un ángulo).

Los métodos y las variables de clase son lo más parecido que ***Java*** tiene a las funciones y variables globales de C/C++ o Visual Basic.

5.6. Ejemplo

A continuación se muestra un ejemplo que calcula la distancia entre dos puntos. Para ello se crea la clase `punto.java` que contiene dos métodos, uno que inicializa el objeto punto con sus coordenadas, y otro que calcula la distancia de ese punto a otro punto "q". Para probar el funcionamiento de esta clase se ha creado otra clase `usoPunto.java`.

```
// Clase punto.
// fichero punto.java
import java.lang.Math.*;
class Punto {
	// Variables de instancia
	private double x, y;


	// Métodos
	public void valorInicial(double vx, double vy)
	{
		x=vx;
		y=vy;
	}

	public double distancia(Punto q)
	{
		double dx=this.x-q.x;
		double dy=this.y-q.y;
		return Math.sqrt(dx*dx+dy*dy);
	}
}

// fichero usoPunto.java
class UsoPunto {
	public static void main(String arg []) {
		Punto p, q;
		double d;
		p=new Punto();
		p.valorInicial(5, 4);
		q=new Punto();
		q.valorInicial(2,2);
		d=p.distancia(q);
		System.out.println("Distancia: " + d);
	}
}
```

5.7. Constructores.

Un punto clave de la POO es el evitar información incorrecta por no haber sido correctamente inicializadas las variables. Java no permite que haya variables miembro que no estén inicializadas, si pueden haber variables locales de métodos sin inicializar, pero el compilador da un error si se intentan utilizar sin asignarles previamente un valor. Ya se ha dicho que Java siempre inicializa, con valores por defecto, las variables miembro de clase y objeto. El segundo paso en la inicialización correcta de objetos es el uso de ***constructores.***

Un ***constructor*** es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del ***constructor*** es reservar memoria e inicializar las variables de instancia de la clase.

Los ***constructores*** no tienen valor de retorno (ni siquiera ***void***) y su ***nombre*** es el mismo que el de la clase. Su ***argumento implícito*** es el objeto que se está creando.

De ordinario una clase tiene ***varios constructores***, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos ***sobrecargados***). Se llama ***constructor por defecto*** al constructor que no tiene argumentos. El programador debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un ***constructor*** de una clase puede llamar a ***otro constructor previamente definido*** en la misma clase por medio de la palabra ***this***. En este contexto, la palabra ***this*** sólo puede aparecer en la ***primera sentencia*** de un ***constructor***.

El ***constructor*** de una ***sub-clase*** puede llamar al constructor de su ***super-clase*** por medio de la palabra ***super***, seguida de los argumentos apropiados entre paréntesis. De esta forma, un constructor sólo tiene que inicializar por sí mismo las variables no heredadas.

El ***constructor*** es tan importante que, si el programador no prepara ***ningún constructor*** para una clase, el ***compilador*** crea un ***constructor por defecto***, inicializando las variables de los tipos primitivos a su valor por defecto, los ***Strings*** a la cadena vacía y las ***referencias*** a objetos a ***null***. Si hace falta, se llama al ***constructor*** de la ***super-clase*** para que inicialice las variables heredadas.

Al igual que los demás métodos de una clase, los ***constructores*** pueden tener también los modificadores de acceso ***public***, ***private***, ***protected*** y ***package***. Si un ***constructor*** es ***private***, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos ***public*** y ***static*** (*factory methods*) que llamen al ***constructor*** y devuelvan un objeto de esa clase.

Dentro de una clase, los ***constructores*** sólo pueden ser llamados por otros ***constructores*** o por métodos ***static***. No pueden ser llamados por los ***métodos de objeto*** de la clase.

Veamos ahora el mismo ejemplo de antes, pero esta vez usando constructor:

```
Ejemplo: Clase Punto.
        // fichero punto.java
        import java.lang.Math.*;
        class Punto {
               // Variables de instancia
               private double x, y;

               // Constructor
               public Punto(double x, double y)
               {
                      this.x=x;
                      this.y=y;
               }

               // Métodos
               public void valorInicial(double vx, double vy) {
```

```
                x=vx;
                y=vy;
        }

        public double distancia(Punto q) {
                double dx=this.x-q.x;
                double dy=this.y-q.y;
                return Math.sqrt(dx*dx+dy*dy);
        }
}

// fichero usoPunto.java
class UsoPunto {
        public static void main(String arg []) {
                Punto p, q;
                double d;
                p=new Punto(5, 4);
                q=new Punto(2, 2):
                d=p.distancia(q);
                System.out.println("Distancia: " + d);
        }
}
```

5.8. Inicializadores

Por motivos que se verán más adelante, ***Java*** todavía dispone de una tercera línea de actuación para evitar que haya variables sin inicializar correctamente. Son los ***inicializadores***, que pueden ser ***static*** (para la clase) o ***de objeto***.

5.8.1. *Inicializadores static*

Un ***inicializador static*** es algo parecido a un método (un bloque {…} de código sin nombre y sin argumentos, precedido por la palabra ***static***) que se llama automáticamente al crear la clase (al utilizarla por primera vez). Se diferencia del ***constructor*** en que no es llamado para cada objeto, sino una sola vez para toda la clase.

Los tipos primitivos pueden inicializarse directamente con asignaciones en la clase o en el constructor, pero para inicializar objetos o elementos más complicados es bueno utilizar un ***inicializador***, ya que permite gestionar ***excepciones***[1] con ***try…catch***.

Los ***inicializadores static*** se crean dentro de la clase, como métodos sin nombre y sin valor de retorno, con tan sólo la palabra ***static*** y el código entre llaves {...}. En una clase pueden definirse ***varios inicializadores static***, que se llamarán en el orden en que han sido definidos.

Los ***inicializadores static*** se pueden utilizar para dar valor a las variables ***static***. Además se suelen utilizar para llamar a ***métodos nativos***, esto es, a métodos escritos por ejemplo en C/C++ (llamando a los métodos ***System.load( )*** o ***System.loadLibrary( )***, que leen las librerías nativas). Por ejemplo:

```
static{
      System.loadLibrary("MyNativeLibrary");
}
```

5.8.2. *Inicializadores de objeto*

A partir de ***Java 1.1*** existen también ***inicializadores de objeto***, que no llevan la palabra ***static***. Se utilizan para las ***clases anónimas***, que por no tener nombre no tienen constructor. En este caso se llaman cada vez que se crea un objeto de la clase anónima.

[1] Las excepciones son situaciones de error o, en general, situaciones anómalas que puede exigir ciertas actuaciones del propio programa o del usuario. Las excepciones se explicarán con más detalle posteriormente.

5.9. Resumen del proceso de creación de un objeto.

El proceso de creación de objetos de una clase es el siguiente:

1. Al crear el primer objeto de la clase o al utilizar el primer método o variable ***static*** se localiza la clase y se carga en memoria.
2. Se ejecutan los ***inicializadores static*** (sólo una vez).
3. Cada vez que se quiere crear un ***nuevo objeto***:
   - se comienza reservando la memoria necesaria.
   - se da valor por defecto a las variables miembro de los tipos primitivos.
   - se ejecutan los inicializadores de objeto.
   - se ejecutan los constructores.

5.10. Destrucción de objetos (liberación de memoria).

En ***Java*** no hay ***destructores*** como en C++. El sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han ***perdido la referencia***, esto es, objetos que ya no tienen ningún nombre que permita acceder a ellos, por ejemplo por haber llegado al final del bloque en el que habían sido definidos, porque a la ***referencia*** se le ha asignado el valor ***null*** o porque a la ***referencia*** se le ha asignado la dirección de otro objeto. A esta característica de ***Java*** se le llama ***garbage collection*** (recogida de basura).

En ***Java*** es normal que varias variables de tipo referencia apunten al mismo objeto. ***Java*** lleva internamente un contador de cuántas referencias hay sobre cada objeto. El objeto podrá ser borrado cuando el número de referencias sea cero. Como ya se ha dicho, una forma de hacer que un objeto quede sin referencia es cambiar ésta a ***null***, haciendo por ejemplo:

```
ObjetoRef = null;
```

En ***Java*** no se sabe exactamente cuándo se va a activar el ***garbage collector***. Si no falta memoria es posible que no se llegue a activar en ningún momento. No es pues conveniente confiar en él para la realización de otras tareas más criticas.

Se puede llamar explícitamente al ***garbage collector*** con el método **System.gc()**, aunque esto es considerado por el sistema sólo como una "sugerencia" a la JVM.

5.11. Finalizadores

Los ***finalizadores*** son métodos que vienen a completar la labor del ***garbage collector***. Un ***finalizador*** es un método que se llama automáticamente cuando se va a destruir un objeto (antes de que la memoria sea liberada de modo automático por el sistema). Se utilizan para ciertas ***operaciones de terminación*** distintas de liberar memoria (por ejemplo: cerrar ficheros, cerrar conexiones de red, liberar memoria reservada por funciones nativas, etc.). Hay que tener en cuenta que el ***garbage collector*** sólo libera la memoria reservada con ***new***. Si por ejemplo se ha reservado memoria con funciones nativas en C (por ejemplo, utilizando la función ***malloc()***), esta memoria hay que liberarla explícitamente con el método ***finalize()***.

Un ***finalizador*** es un método de objeto (no ***static***), sin valor de retorno (***void***), sin argumentos y que siempre se llama ***finalize()***. Los ***finalizadores*** se llaman de modo automático siempre que hayan sido definidos por el programador de la clase. Para realizar su tarea correctamente, un ***finalizador*** debería terminar siempre llamando al ***finalizador*** de su ***super-clase***.

Tampoco se puede saber el momento preciso en que los ***finalizadores*** van a ser llamados. En muchas ocasiones será conveniente que el programador realice esas operaciones de finalización de modo explícito mediante otros métodos que él mismo llame.

El método ***System.runFinalization()*** "sugiere" a la JVM que ejecute los ***finalizadores*** de los objetos pendientes (que han perdido la referencia). Parece ser que para que este método se ejecute hay que llamar primero a ***gc()*** y luego a ***runFinalization()***.

6. RECURSIVIDAD

La recursividad es una técnica de programación muy potente que puede ser utilizada en lugar de la iteración. Los algoritmos recursivos dan soluciones elegantes y simples a problemas de gran complejidad. Estas soluciones son, en general, bien estructuradas y modulares. De hecho, la forma mediante la que los módulos de una solución recursiva interactúan, es precisamente lo que hace de la recursión una herramienta de programación única y potente.

¿En qué consiste realmente la recursividad? Es una técnica que nos permite que un subalgoritmo (método) se invoque a sí mismo para resolver una "versión más pequeña" del problema original que le fue encomendado. El aspecto de un método recursivo es:

```
metodo T(...)
{
    ...
    T(...);
    ...
}
```

Ejemplo: Factorial de un natural

$$Factorial(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * Factorial(n-1) & \text{si } n > 0 \end{cases}$$

Ej: 4! = 4*3!
3! = 3*2!
2! = 2*1!
1! = 1*0! = 1*1

```
public int Factorial(int n) {
    if (n==0) return 1;
    return n * Factorial(n -1);
}
```

De esta función podemos sacar varias conclusiones:

1.- El subalgoritmo se invoca a sí mismo (esto es lo que lo convierte en recursivo).

2.- Cada llamada recursiva se hace con un parámetro de menor valor que el de la anterior llamada. Así cada vez se está invocando a otro problema idéntico pero de menor valor.

3.- Existe un caso degenerado en el que se actúa de forma diferente, esto es, ya no se utiliza la recursividad. Lo importante es que la forma en la que el tamaño del problema disminuye asegura que se llegará a este caso degenerado o caso base.

Para determinar si un programa recursivo está bien diseñado se utiliza el método de las tres preguntas:

1.- La pregunta *Caso-Base*: ¿Existe una salida no recursiva o caso base del subalgoritmo, y éste funciona correctamente para ella?

2.- La pregunta *Invocar_más_pequeño*. ¿Cada llamada recursiva al subalgoritmo se refiere a un caso más pequeño del problema original?

3.- La pregunta *Caso-General*. Suponiendo que la(s) llamada(s) recursiva(s) funciona(n) correctamente, así como el caso base, ¿funciona correctamente todo el subalgoritmo?

Recursión frente a iteración

Ya sabemos que la recursión es una técnica de programación para resolver problemas potentes que a menudo produce soluciones simples y claras, incluso para problemas muy complejos. Sin embargo, la recursión también tiene algunas desventajas las cuales se enmarcan en el campo de la eficiencia. Muchas veces un algoritmo iterativo, es más eficiente que su correspondiente recursivo. Existen dos factores que contribuyen a ello:

- La sobrecarga asociada con las llamadas a subalgoritmos.
  - Una simple llamada puede generar un gran número de llamadas recursivas. (Fact(n) genera n llamadas recursivas).
  - ¿La claridad compensa la sobrecarga?
  - El valor de la recursividad reside en el hecho de que se puede usar para resolver problemas sin fácil solución iterativa.

- La ineficiencia inherente de algunos algoritmos recursivos.

El hecho de que la recursividad tenga estas dos desventajas, no quiere decir que sea una mala técnica y que no se deba utilizar, sino que hay que usarla cuando realmente sea necesario. El verdadero valor de la recursividad es como herramienta para resolver problemas para los que no hay soluciones no recursivas simples.

Ejemplo: Función potencia para enteros positivos utilizando recursividad.

```
public float potencia (int b, int e)
{
     if (e == 0) return 1;
     return(b * potencia (b, e-1));
}
```

7. PACKAGES

7.1. Qué es un package.

Un ***package*** es una agrupación de clases. En la API de ***Java 1.1*** había 22 ***packages***; en Java 1.2 había 59 ***packages***, y ha seguido aumentando, lo que da una idea del "crecimiento" experimentado por el lenguaje.

Además el usuario puede crear sus propios ***packages***. Para que una clase pase a formar parte de un ***package*** llamado ***pkgName***, hay que introducir en ella la sentencia:

package pkgName;

que debe ser la primera sentencia del fichero sin contar comentarios y líneas en blanco.

Los nombres de los ***packages*** se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un ***package*** puede constar de varios nombres unidos por puntos (los propios ***packages*** de ***Java*** siguen esta norma, como por ejemplo ***java.awt.event***).

Todas las clases que forman parte de un ***package*** deben estar en el mismo directorio. Los nombres compuestos de los ***packages*** están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los ***nombres de las clases*** de ***Java*** sean únicos en ***Internet***. Es el nombre del ***package*** lo que permite obtener esta característica. Una forma de conseguirlo es incluir el ***nombre del dominio*** (quitando quizás el país), como por ejemplo en el ***package*** siguiente:

```
es.ceit.jgjalon.infor2.ordenar
```

Las clases de un ***package*** se almacenan en un directorio con el mismo nombre largo (***path***) que el ***package***. Por ejemplo, la clase,

```
es.ceit.jgjalon.infor2.ordenar.QuickSort.class
```

debería estar en el directorio,

```
CLASSPATH\es\ceit\jgjalon\infor2\ordenar\QuickSort.class
```

donde CLASSPATH es una variable de entorno del PC que establece la posición absoluta de los directorios en los que hay clases de ***Java*** (clases del sistema o de usuario), en este caso la posición del directorio ***es*** en los discos locales del ordenador.

Los ***packages*** se utilizan con las finalidades siguientes:

1. Para agrupar clases relacionadas.
2. Para evitar conflictos de nombres (se recuerda que el dominio de nombres de ***Java*** es ***Internet***). En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del ***package***.
3. Para ayudar en el control de la accesibilidad de clases y miembros.

7.2. Cómo funcionan los packages.

Con la sentencia ***import packname;*** se puede evitar tener que utilizar nombres muy largos, al mismo tiempo que se evitan los conflictos entre nombres. Si a pesar de todo hay conflicto entre nombres de clases, ***Java*** da un error y obliga a utilizar los nombres de las clases cualificados con el nombre del ***package***.

El importar un ***package*** no hace que se carguen todas las clases del ***package***: sólo se cargarán las clases ***public*** que se vayan a utilizar. Al importar un ***package*** no se importan los ***sub-packages***. Éstos deben ser importados explícitamente, pues en realidad son ***packages*** distintos. Por ejemplo, al importar ***java.awt*** no se importa ***java.awt.event***.

Es posible guardar en jerarquías de directorios diferentes los ficheros ****.class*** y ****.java***, con objeto por ejemplo de no mostrar la situación del código fuente. Los ***packages*** hacen referencia a los ficheros compilados ****.class***.

En un programa de ***Java***, una clase puede ser referida con su nombre completo (el nombre del ***package*** más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia ***import*** permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del ***package*** importado. Se importan por defecto el ***package java.lang*** y el ***package*** actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar ***import***: para ***una clase*** y para ***todo un package***:

```
import es.ceit.jgjalon.infor2.ordenar.QuickSort.class;
import es.ceit.jgjalon.infor2.ordenar.*;
```

que deberían estar en el directorio:

```
classpath\es\ceit\jgjalon\infor2\ordenar
```