

## Curso de programación Java III

Artículo publicado originalmente en la revista [Sólo Programadores](#)

Hasta ahora en este curso de programación Java hemos introducido los tipos de datos primitivos del lenguaje (los que permiten representar números enteros, números reales y valores lógicos); hemos presentado algunos tipos de datos más complejos (Strings, arrays y enumeraciones); y hemos revisado las distintas estructuras de control de flujo (bucles y condicionales) que proporciona el lenguaje. Estas herramientas, junto con las funciones y los módulos, son los pilares de la programación estructurada.

Sin embargo, Java no es un lenguaje de programación estructurado. Es un lenguaje de programación orientado a objetos. En Java un programa no es un código que llama a procedimientos o funciones; un programa es un montón de objetos, independientes entre si, que dialogan entre ellos pasándose mensajes para llegar a resolver el problema en cuestión.

Esta serie de artículos no presentará en profundidad la programación orientada a objetos (POO). El abordar esta tarea nos llevaría a escribir otra serie de artículos completa. Sí introduciremos brevemente algunos conceptos de la POO, y daremos alguna justificación de su necesidad. Pero donde haremos más énfasis será en el soporte que el lenguaje de programación Java proporciona para la POO. Si bien será posible seguir la serie de artículos con los conceptos de orientación a objetos que se introducirán en este artículo, recomiendo al lector que no tenga experiencia en este campo que lea alguno de los muchos libros publicados al respecto o, en su defecto, el tutorial [Orientación a Objetos: Conceptos y Terminología](#).

En este tercer capítulo de la serie, será el momento de comenzar a usar un entorno de desarrollo. Hasta ahora, si el lector ha seguido mi consejo, habrá estado usando el jdk en la consola. El entorno de desarrollo que usaremos para este capítulo será BlueJ (<http://bluej.org>, ver figura 1). No se trata de un entorno de desarrollo profesional, sino de un entorno de desarrollo orientado a la docencia y diseñado para presentar conceptos relativos a la POO. Si ya conoces en profundidad la POO puedes decidir saltar directamente a Netbeans o Eclipse. No obstante, este artículo está escrito pensando en aquellos lectores con pocos o nulos conocimientos de POO.



FIGURA 1: En la web de BlueJ además de descargar el IDE podréis encontrar documentación sobre él

En el artículo no tengo intención de explicar cómo usar BlueJ; lo cual no quiere decir que su uso sea trivial para alguien que desconozca la POO. Simplemente resulta que un medio impreso no es lo ideal para presentar el uso de una herramienta de escritorio. En el CD de la revista podrás encontrar un tutorial flash donde se explica cómo trabajar con BlueJ. En el artículo indicaré cuándo es el momento ideal para ver dicho tutorial.

## 1 La abstracción en los lenguajes de programación: el porqué de la programación orientada a objetos

La abstracción es una herramienta que nos permite comprender, analizar y resolver problemas complejos del mundo real de un modo eficaz. La abstracción se fundamenta en eliminar todo aquello que no es relevante para un problema permitiendo centrarnos sobre aquella información que sí lo es; el ser o no relevante depende fuertemente del contexto del problema.

Aunque la abstracción es una herramienta general aplicable a cualquier campo, a nosotros nos interesa la abstracción que proporciona un lenguaje de programación. Todos los lenguajes de programación proveen abstracciones. Muchos autores defienden que la complejidad de los problemas que un lenguaje de programación permite resolver depende del tipo y de la calidad de las abstracciones que el lenguaje permite crear.

El lenguaje ensamblador proporciona una pequeña abstracción respecto a la máquina sobre la que se ejecuta el programa. Los lenguajes de programación imperativos (Fortran, BASIC, Pascal, C, etc.) son abstracciones del lenguaje ensamblador. Así, por ejemplo, estos lenguajes nos permiten sumar dos números cualesquiera con el operador "+". Las operaciones que hay que realizar para sumar dos enteros de 32 bits son diferentes de las que hay que hacer para sumar dos enteros de 64 bits. Y no tienen nada que ver con las operaciones que hay que hacer para sumar dos números reales de 32 bits, números que se representan de un modo completamente diferente en el ordenador. Nuevamente, la suma de números reales de 64 bits difiere de la suma de números reales de 32 bits. Sin embargo estos lenguajes de programación nos permiten olvidarnos (abstraernos) de los detalles de cómo se realiza la operación de suma, y centrarnos en su semántica: la adición de dos números.

Si bien estos lenguajes son una gran aportación respecto al ensamblador, no son suficiente, ya que siguen requiriendo que el programador piense en términos de la estructura del ordenador más que en términos del problema que tienen que resolver. El programador se ve obligado a establecer una asociación entre el modelo de la máquina (que, por ejemplo, comprende variables enteras, reales, cadena de caracteres, instrucciones condicionales, bucles, etc.) y el modelo del problema que pretende resolver (que, por ejemplo, puede comprender cuentas bancarias con sus saldos y sus respectivos propietarios, intereses se deben de abonar en cada cuenta, comisiones que se deben de retirar, condiciones de cobro de hipotecas, etc.). El esfuerzo que requiere al programador realizar una correspondencia entre ambos mundos hace que los programas sean difíciles de escribir y de mantener y ha sido el detonante para que aparezcan una gran cantidad de metodologías de programación en la industria del software.

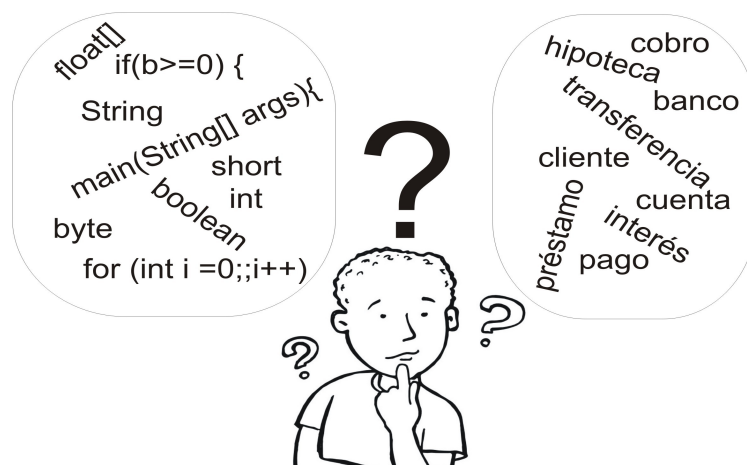


FIGURA 2: Establecer una asociación adecuada entre el modelo de la máquina y el modelo del mundo real es una de las tareas más complicadas de la programación

Una forma de aumentar el nivel de abstracción de los lenguajes de programación sería diseñar un lenguaje de muy alto nivel que incorporase múltiples estructuras de datos generales y un conjunto de operaciones para manejar dichas estructuras. Estos lenguajes tratan de modelar el problema a resolver y no la máquina en la que se ejecutará el programa. Así, por ejemplo, en Lisp y APL se modela una visión particular del mundo: para el primero, todos los problemas se traducen en listas; para el segundo, en algoritmos. PROLOG, quizás el lenguaje de más difusión que sigue esta aproximación, modela todos los problemas como cadenas de decisiones. Todos estos lenguajes son una buena aproximación para resolver un tipo particular de problemas: aquéllos para los cuales aportan unas estructuras de datos y operaciones que componen una abstracción adecuada del problema. Sin embargo, cuando abandonamos el dominio para el cual han sido creados se convierten en terriblemente torpes y engorrosos, ya que no proporcionan las abstracciones adecuadas.

Intentar construir un lenguaje que modele la mayor parte de las abstracciones del mundo real que sus usuarios pudiesen demandar ha resultado ser, al menos hasta la fecha, un problema inabordable. La gran cantidad de estructuras de datos y de operaciones entre ellas que habría que contemplar convierten al problema del diseño de este lenguaje en intratable.

Llegados a este punto, la mejor solución al problema que se nos plantea es obvia: ya que no podemos proporcionar todas las abstracciones que cualquier usuario puede requerir para resolver cualquier problema, proporcionaremos la mejor herramienta posible (lenguaje de programación) para que el usuario construya las abstracciones que le permitan modelar el problema de un modo lo más adecuado posible. Después, usando las abstracciones que ha creado, el programador resuelve el problema. Ésta es la aproximación seguida por los lenguajes de programación orientada objetos: tratan de proporcionar herramientas generales (que no limiten al programador a un determinado dominio) para representar los elementos que componen el problema. Estos elementos que componen el problema se denominan "objetos". Por tanto, la idea es que el lenguaje se adapte a cada problema particular creando las abstracciones más adecuadas para él.

No obstante, esta aproximación no es incompatible con la anterior: los lenguajes de programación orientados a objetos, en general, y Java, en particular, ofrecen en sus librerías un conjunto cada vez más amplio de estructuras de datos ya implementadas y listas para ser empleadas por el programador. Sin embargo, la clave de su potencia es que permiten al programador definir nuevas estructuras extendiendo las ya existentes y/o modificándolas, o bien partiendo desde cero cuando las librerías del lenguaje no le proporcionan ninguna herramienta útil en la que apoyarse.

## 2 Objetos y clases en Java

Como ya hemos dicho, en POO un programa consta de un conjunto de objetos que intercambian entre ellos mensajes para resolver un determinado problema. A un objeto no le debe importar en absoluto cómo está implementado otro objeto, qué código tiene o deja de tener, qué variables usa... sólo le importa a qué mensajes es capaz de responder. Un mensaje es la invocación de un método de otro objeto. Un método es muy semejante a un procedimiento de la programación estructurada: a un método se le pasan uno, varios o ningún parámetros y nos devuelve un dato a cambio. Sin embargo, los métodos están asociados a los objetos (y por tanto sólo se pueden invocar sobre un objeto) y pueden realizar funciones diferentes dependiendo del estado en el que se encuentre el objeto.

Si analizamos lo que hemos dicho en el párrafo anterior veremos que los objetos parecen tener dos partes bastante diferenciadas: una es la parte que gestiona los mensajes, que ha de ser conocida por los demás, y que no podremos cambiar en el futuro sin modificar los demás objetos. La otra parte es el mecanismo por el cual se generan las acciones requeridas por los mensajes y el

conjunto de variables que se emplean para llevar a cabo estas acciones. Esta segunda parte es, en principio, totalmente desconocida para los demás objetos. Por ello podemos en cualquier momento modificarla sin que a los demás les importe, y además cada programador tendrá total libertad para llevarla a cabo como él considere oportuno.

¿Cómo hago para programar objetos en Java? Los objetos no son programados directamente; lo que el programador debe hacer es construir clases. Una vez ha construido las clases, puede crear objetos a partir de ellas. Una clase es una descripción de un conjunto de objetos que manifiestan los mismos atributos, operaciones, relaciones y la misma semántica. Una clase puede verse desde tres perspectivas diferentes pero, a la vez, complementarias:

- Como un conjunto de objetos que comparten una estructura y un comportamiento.
- Como una plantilla que permite crear objetos.
- Como la definición de la estructura y del comportamiento de un conjunto de objetos.

La primera perspectiva hace énfasis en la clasificación y en la abstracción. Una clase es una abstracción software de un conjunto de objetos (que se pueden corresponder con objetos del mundo real o no) que, por compartir una serie de atributos y comportamiento, clasificamos bajo una misma etiqueta. La segunda perspectiva toma un enfoque un tanto utilitarista: la clase es la "herramienta" que los lenguajes de programación emplean para construir objetos. La tercera hace énfasis en que la definición de una clase es una estructura común que se puede reutilizar (crear objetos con ella) cuántas veces se quiera. A la acción de crear un objeto de una clase se la denomina instanciar; y a los objetos creados instancias de la clase.

Basta ya de teoría; veamos cuál es la sintaxis de Java para definir clases.

## 2.1 Definición de clases en Java

La forma más general para definir una clase en Java es empleando la siguiente sintaxis:

```
[Modificador]classNombreClase[extendsNombreClasePadre][implementsinterface]{  
Declaración de variables;  
Declaración de métodos;  
}
```

Los campos que van entre corchetes no son obligatorios. NombreClase es el nombre que le queramos dar a nuestra clase, NombreClasePadre es el nombre de la clase padre, de la cual hereda los métodos y variables, e interface es una especie de "contrato" que la clase se compromete a cumplir. La herencia y las interfaces serán abordadas en detalle más adelante. Como ambos campos son opcionales, por lo de ahora los ignoraremos.

Los modificadores indican distintas características de la clase. Veamos qué opciones tenemos:

- **public**: la clase es pública y por lo tanto accesible para todo el mundo. Sólo podemos tener una clase pública por unidad de compilación, es decir, por cada fichero de código fuente.

- **Ninguno**: la clase es "amistosa". Será accesible para las demás clases del package (paquete). Los paquetes serán introducidos en el siguiente artículo de esta serie. Son una estructura que permite organizar programas relativamente complejos donde entran en juego un número de clases alto. Como este es nuestro primer artículo sobre orientación a objetos en Java, los programas que haremos son sencillos y no tomaran ventaja del uso de paquetes. Por otro lado, mientras todas las clases con las que estemos trabajando estén en el mismo directorio pertenecerán al mismo paquete y, por tanto, no emplear modificador es equivalente a emplear el modificador public . Como por lo de ahora trabajaremos en un solo directorio asumiremos que la ausencia de modificador es equivalente a que la clase sea pública.
- **final**: indicará que esta clase no puede "tener hijos", no se puede derivar ninguna clase de ella.
- **abstract**: se trata de una clase de la cual no se puede instanciar ningún objeto. Habitualmente, las clases abstractas se suelen corresponder con objetos abstractos del mundo real. Por ejemplo, con el concepto de "ser vivo". En el mundo real no hay seres vivos "puros"; hay perros, gatos, palmeras, algas, peces... y todos ellos son "seres vivos". Pero no existe una entidad "ser vivo" que no pertenezca a alguna categoría de animal o planta más específica. El concepto de "ser vivo" en si mismo es un concepto abstracto.

Quando presentemos la herencia de clases en Java los modificadores final y abstract cobrarán más sentido. Ya sé que puede parecer que me esté dejando muchas cosas en el tintero y que así va a ser muy difícil seguir la explicación. No es simple introducir la POO y, a pesar de que llevo casi 10 años explicándola todos los años en cursos universitarios (tanto en Java como en C++) todavía no he encontrado una forma de presentarla sin introducir algunos conceptos antes de explicarlos en profundidad. Ten un poco de fe y sigue leyendo. Las cosas irán encajando y todo comenzará a tener sentido más adelante.

El listado 1 muestra un ejemplo de clase Java. Nuestra clase representaría a una persona; la persona tendría dos atributos: una edad y un nombre. Dentro de la clase hemos definido tres métodos, que simplemente muestran por consola o bien un texto o bien el valor de uno de los atributos de la clase.

Como puede observarse en el listado 1, Java admite lo que se llama sobrecarga de métodos: puede haber varios métodos con el mismo nombre pero a los cuales se les pasan distintos parámetros. Según los parámetros que se le pasen se invocará a uno u otro método.

Si el lector lo desea, puede compilar el código del listado 1. No le será posible, sin embargo, ejecutarlo. No tiene un método main.

```
//LISTADO 1: Clase que representa a una persona.
//código Persona.java del CD
public class Persona {
    private int edad;
    private String nombre;

    public void nace(){
        System.out.println("Holamundo");
    }
    public void getNombre(){
        System.out.println(nombre);
    }
}
```

```

public void getNombre(int i){
    System.out.println(nombre+ ""+i);
}
    public void getEdad(){
    System.out.println(edad);
}
}

```

### 2.1.1 Constructores

Los constructores se definen de un modo muy similar a los métodos, pero su nombre debe coincidir con el nombre de la clase y nunca devuelven ningún tipo de dato, no siendo necesario indicar que el tipo de dato devuelto es void . Dentro de un constructor suele haber código para inicializar los valores de los objetos y realizar las operaciones que sean necesarias para la generación de este objeto (crear otros objetos que puedan estar contenidos dentro de este objeto, abrir un archivo o una conexión de internet...). Como veremos más adelante, la utilidad de los constructores es permitir al programador crear objetos a partir de la clase: para crear un objeto deberemos invocar uno de los constructores. Al igual que sucede con los métodos, los constructores admiten sobrecarga.

En el listado 2 podemos ver dos constructores de la clase Persona. El primero, el que no toma parámetros, se llama constructor por defecto. Si cuando creamos una clase no indicamos ningún constructor, el compilador generará un constructor por defecto para la clase de modo automático. Nuestro constructor por defecto nos permite crear una persona que se llama "Paco" y que tiene 30 años. El segundo constructor permite especificar la edad y el nombre de la persona que se va a construir.

this es una variable especial de sólo lectura que proporciona Java. Contiene una referencia al objeto en el que se usa dicha variable. A veces es útil que un objeto pueda referenciarse a si mismo; en el listado 2 esto se emplea para distinguir entre la variable edad argumento de uno de los constructores de Persona del atributo edad ( this.edad ) de la clase.

```

//LISTADO 2: Constructores de la clase Persona
//código Persona.java del CD
    public Persona(){
        edad = 30;
        nombre = "Paco";
    }
public Persona(int edad,String nombre){
    this.edad = edad;
    this.nombre = nombre;
}

```

Otro uso de this , como se muestra en el listado 3, es invocar a un constructor de la clase desde otro constructor. En este caso, la llamada al constructor debe ser la primera sentencia dentro del constructor que invoca al segundo.

```

//LISTADO 3: Uso de this para invocar desde un constructor a otro

```

```

class Cliente{
publicCliente(Stringn){
//Llamamos al otro constructor.
    this(n, Cuenta.nuevo_numero());
    .....
}
publicCliente(Stringn,inta){
    nombre = n;
    numero_cuenta = a;
}
.....
}

```

Este es un buen momento para echarle un vistazo al tutorial flash que podrás encontrar en el CD de la revista. En él, tras realizar una breve introducción a BlueJ, se carga esta clase, se crean varios objetos de ella y se muestra cómo invocar sus métodos. Esto es posible sólo mediante BlueJ, y no mediante el JDK. Para poder emplear esta clase con el JDK de Sun tendríamos que escribir un programa Java que crease instancias de la clase e invocase sus métodos. BlueJ permite realizar estas acciones empleando menús contextuales (ver figura 3), lo que nos permite "jugar" con la clase de un modo similar a como los intérpretes de lenguajes de programación como Python permiten crear instancias e invocar métodos de los objetos creados. Según mi experiencia, esto resulta muy útil para familiarizarse con los conceptos de la POO.

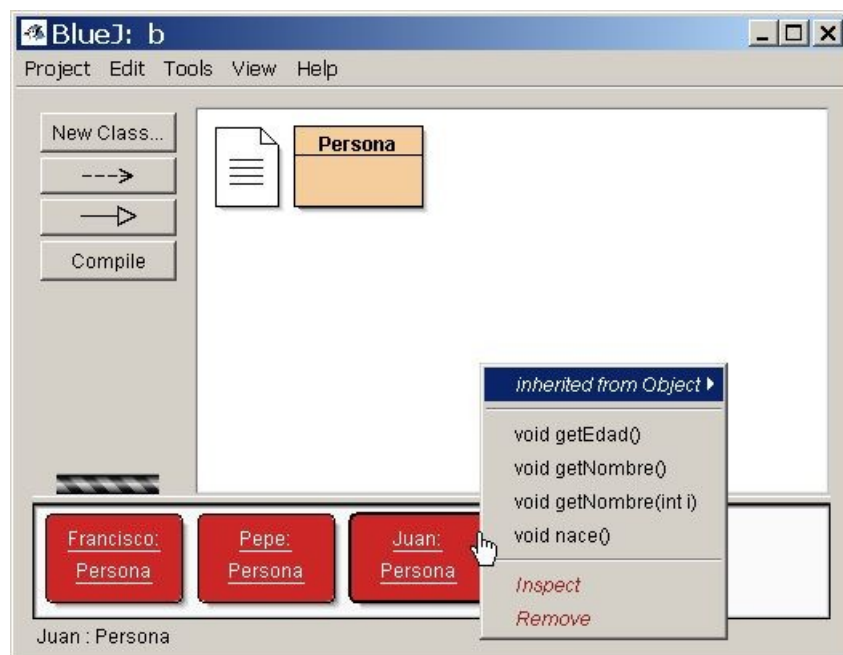


FIGURA 3: BlueJ permite crear instancias de clases Java e invocar a los métodos de las instancias

### 2.1.2 Modificadores de métodos y variables

Aunque hasta que expliquemos la herencia y los paquetes algunos de los modificadores que vamos a ver a continuación carecerán de sentido, por completitud los presentaremos aquí. El primer tipo de modificadores que vamos a ver son los modificadores de visibilidad; éstos permiten indicar quién podrá ver y acceder a un método o un atributo de una clase. Éstos son los modificadores que debemos emplear para determinar cuál es la parte pública de nuestros objetos (la parte que va a ser accesible por los demás) y cuál es la parte privada (la parte que son detalles de implementación y que no queremos exponer). Estos modificadores son los mismos tanto para variables como para métodos y son:

- **public**: puede acceder todo el mundo a la variable o método. Por norma general, las variables nunca son públicas; las variables siempre suelen considerarse detalles de implementación. La parte pública de una clase suele estar constituida por métodos y constructores.
- **Ninguno**: "amistosa", el atributo o método es accesible por cualquier miembro del paquete (package), pero no por otras clases que pertenezcan a otro paquete distinto. Más adelante explicaremos que es un paquete.
- **protected**: el atributo o método es accesible por las clases hijas de la clase que posee la variable y por las que estén en el mismo paquete. El significado de este modificador tendrá más sentido cuando se explique la herencia.
- **private**: nadie salvo la clase misma puede acceder a estas variables o métodos. Debemos tener en cuenta que podrán acceder a ellos todas las instancias de la clase. Por norma general, todas las variables de la clase deben ser privadas. También es frecuente que en una clase haya métodos privados, métodos que suelen ser auxiliares para otros métodos públicos que componen la interfaz de la clase pero cuya funcionalidad no se quiere exponer.

Mientras no introduzcamos la herencia y los paquetes, los únicos modificadores que tienen sentido son `public` y `private`. Emplear el primero significa colocar el método o la variable en la interfaz de la clase y hacerlo accesible a otras clases. Emplear el segundo significa ocultarlo.

Los cuatro modificadores que hemos presentado son mutuamente excluyentes; es decir, un atributo (obviamente) no puede ser `public` y `private` a la vez. Los modificadores que presentaremos a continuación no son excluyentes ni entre sí, ni respecto a los cuatro anteriores. Aunque son los mismos tanto para los atributos como para los métodos, su semántica es diferente por lo que los presentaremos por separado.

En el caso de los atributos, además del modificador de visibilidad, podemos especificar los modificadores:

- **static**: la variable atributo es la misma para todas las instancias de una clase: todas comparten ese dato. Si una instancia lo modifica todas ven dicha modificación. Las variables estáticas son equivalentes a las variables de la programación estructurada: definir la variable estática significa tener una única variable estática, mientras que definir la variable no estática significa tener una variable por cada objeto que se cree a partir de la clase.
- **final**: se emplea para definir constantes: un dato tipo `final` no puede variar nunca su valor. La variable no tiene porque inicializarse en el momento de definirse, pero cuando se inicializa ya no puede cambiar su valor.

En el listado 4 vemos un ejemplo de uso de ambos modificadores. Por un lado, se emplea el modificador `final` para definir una cadena de caracteres cuyo valor nunca va a cambiar. Por otro lado, se emplea una variable estática para llevar cuenta del número de instancias que se han creado de la clase. La variable `numeroMarcianos` se incrementa cada vez que se crea un `Marciano`, y se decrementa cada vez que se muere un `Marciano`. Como es una variable estática, todas las instancias de la clase comparten la misma variable, por lo que cuando una instancia la incrementa todas ven el valor incrementado y cuando una instancia la decrementa todas ven el valor decrementado.

El tutorial flash muestra cómo emplear BlueJ para demostrar cómo funcionan las variables estáticas.

//LISTADO 4: Ejemplo de uso de variables estáticas y finales.

//código Marciano.java del CD

```
class Marciano {
    private boolean vivo;
private static int numeroMarcianos=0;
final String soy="marciano";

    Marciano(){
        vivo = true;
        numeroMarcianos++;
    }
    void muerto(){
        if(vivo){
            vivo = false;
            numeroMarcianos--;
        }
    }
    void quienEres(){
        System.out.println("Soyun"+soy);
    }
}
```

Los métodos también pueden llevar el modificador static o final , aunque en este caso su significado es distinto:

- **static**: es un método al cual se puede invocar sin crear ningún objeto de dicha clase. Math.sin() y Math.cos() son dos ejemplos de métodos estáticos. Los métodos estáticos son equivalentes a las funciones de la programación estructurada. Desde un método estático sólo podemos invocar otros métodos que también sean estáticos, y sólo podemos acceder a atributos de la clase que sean estáticos.
- **final**: si un método es final no podrá ser cambiado por ninguna clase que herede de la clase donde se definió. Es un método que no se puede sobrescribir. Nuevamente, cuando se presente la herencia se explicará este punto con más detalle.

Definir un método como estático es hacerlo equivalente a una función de la programación estructurada. Todos los métodos de la clase Math son estáticos, de ahí que se puedan invocar directamente sin necesidad de crear instancias. Para invocar a un método estático antes del nombre del método debe ir en nombre de la clase, y entre ambos un ".". En el listado 5 vemos una clase Mat con dos métodos que permiten realizar operaciones matemáticas, de un modo similar a la clase Math . Estos métodos son invocados desde el método main .

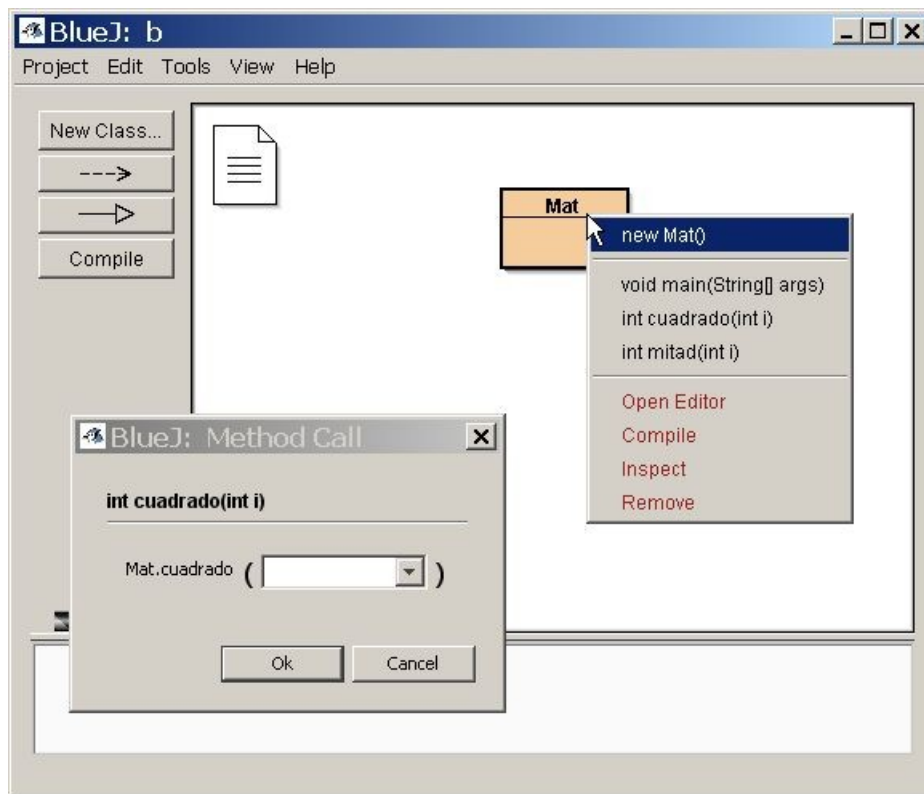


FIGURA 4: BlueJ permite invocar directamente los métodos estáticos sobre las clases

El método main siempre tiene que ser estático porque es el punto de entrada del programa: si el método main no fuese estático, antes de invocarlo tendríamos que crear un objeto de la clase que lo contiene. Pero como el programa no se ha comenzado ejecutar todavía (el main es lo primero que se ejecuta) no hay nadie que pueda crear ese objeto. De ahí que tenga que ser estático.

```
//LISTADO 5: Ejemplo de uso de varios métodos estáticos.
//código Mat.java del CD
class Mat{
public static int cuadrado(int i){
    return i*i;
}
public static int mitad(int i){
    return i/2;
}
public static void main(String[] args){
    System.out.println("Mat.cuadrado(40):"+Mat.cuadrado(40));
    System.out.println("Mat.mitad(40):"+Mat.mitad(40));
}
}
```

## 2.2 Creación y referencia a objetos

El propósito de las clases es permitirnos crear objetos. Un objeto en el ordenador es esencialmente un bloque de memoria con espacio para guardar las variables de dicho objeto. En Java todos los

objetos se crean en el heap o memoria dinámica y para crearlos se emplea el operador new . Si quisiéramos crear un objeto de tipo persona podríamos emplear las siguientes sentencias:

```
Persona hombre;  
hombre = new Persona ();
```

Con la primera sentencia hemos creado una variable capaz de referenciar un objeto de tipo Persona. Con la segunda inicializamos la variable, de tal forma que contenga una persona que se llamará "Paco" y tendrá 30 años, ya que esto es lo que hacía el constructor por defecto de esta clase. Si quisiésemos indicar el nombre y la edad de la persona que estamos creando podríamos invocar al otro constructor de la clase:

```
Persona pepe = new Persona ("Pepe", 45);
```

En este caso habremos construido un objeto de tipo persona cuyo nombre es "Pepe" y que tiene 45 años. Una vez hemos construido un objeto podemos acceder a sus atributos y métodos empleando el operador ".", siempre que el código que está intentando realizar el acceso tenga permiso para ello. Estos permisos dependerán de los modificadores de visibilidad que tengan dichos atributos o métodos. Así, por ejemplo, si el modificador de visibilidad es public cualquier código podrá acceder al atributo o método. Si el modificador de visibilidad es private , sólo el código de la propia clase podrá hacer era dicho tributo o método.

Dado nuestro objeto pepe , podremos obtener su nombre y su edad, respectivamente, invocando los correspondientes métodos de la clase Persona :

```
intedadPepe=pepe.getEdad();  
intnombrePepe=pepe.getNombre();
```

Las dos sentencias anteriores son válidas en cualquier parte de nuestro programa, ya que ambos métodos son public . Sin embargo las sentencias:

```
intedadPepe=pepe.edad;  
intnombrePepe=pepe.nombre;
```

que acceden directamente a los atributos de la clase Persona sólo serán válidas dentro del código de la propia clase Persona , ya que ambos atributos eran private.

## 2.3 La herencia

La herencia permite que una clase pueda basarse en otra ya existente para construirse; constituye, por tanto, un mecanismo muy potente de reutilización de código. Mediante ella una clase, denominada normalmente clase hija, clase derivada o subclase, hereda propiedades y comportamiento de otra clase, denominada clase padre, clase base o superclase. La herencia organiza jerárquicamente las clases que se necesitan para resolver un problema.

La herencia se apoya en el significado de ese concepto de la vida diaria. Así, los seres vivos se dividen en animales y plantas; los animales, a su vez, en mamíferos, anfibios, insectos, aves, reptiles... los mamíferos a su vez en terrestres y acuáticos... cada una de estas clases o categorías

tiene un conjunto de atributos y comportamiento común a todos los miembros de la clase: todas las hembras de los mamíferos poseen glándulas mamarias que sirven para alimentar a sus crías cuando nacen. Estas propiedades y comportamiento son heredadas por todas las subclases de una clase: lo dicho antes para los mamíferos es igualmente válido para los perros, ya que los perros son mamíferos (pertenecen a la clase de los mamíferos).

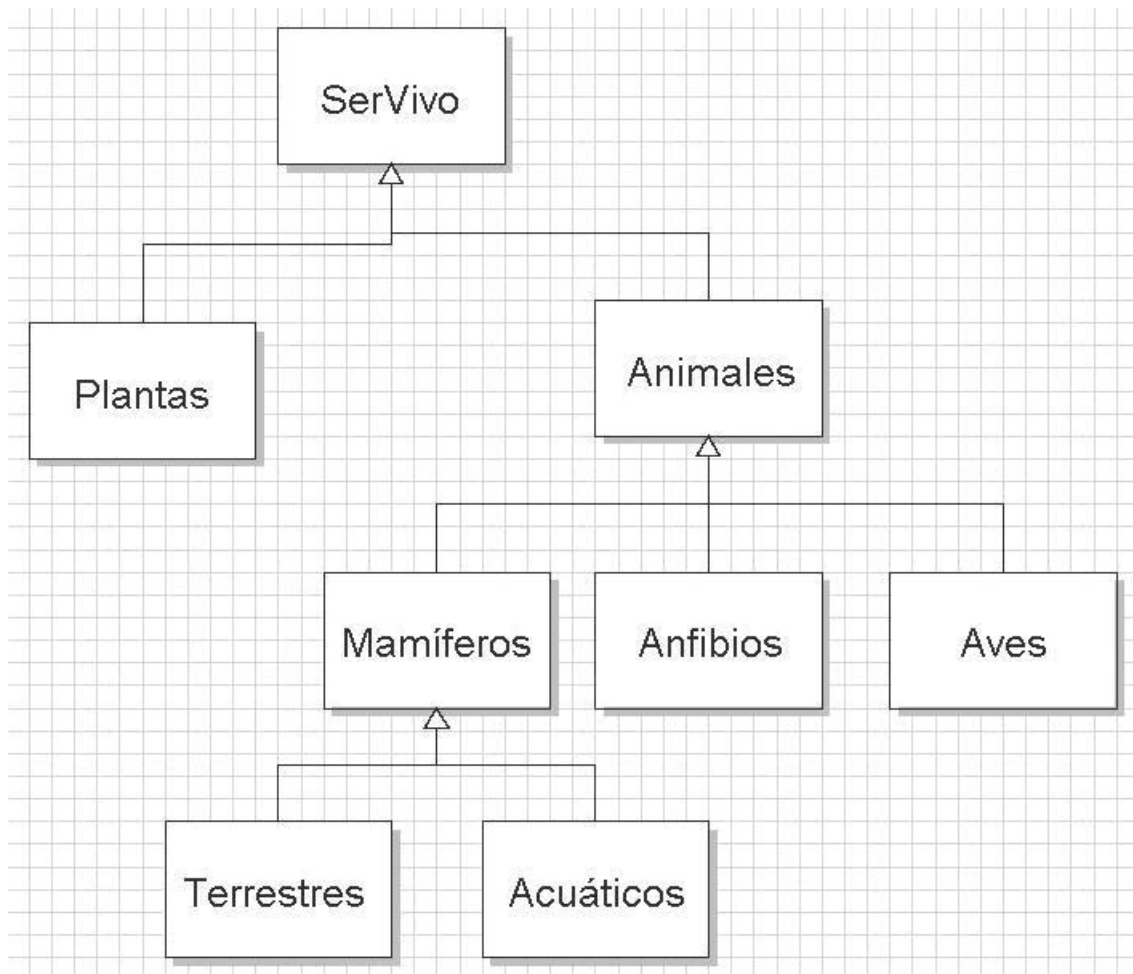


FIGURA 5: El concepto de herencia en los lenguajes de programación es similar al concepto de herencia del mundo real

Una clase hija especializa y extiende el comportamiento de la clase padre; así, un perro es un mamífero, exactamente igual que la ballena; sin embargo un perro también tiene pelo y cuatro patas, cosa que ya no sucede con todos los mamíferos. Podemos decir que un perro es "un tipo más especializado/concreto" de mamífero, que extiende las propiedades y el comportamiento de un mamífero incorporando pelo, cuatro patas, el alimentarse de carne y el ladrar, entre otros.

Una forma de identificar relaciones de tipo herencia es buscando en la descripción del problema las palabras "es un": un perro es un mamífero, un mamífero es un ser vivo, etcétera. El sustantivo que va antes del "es un" se corresponderá con una clase hija y el que va después con la clase padre. No debe confundirse la herencia con la composición. Ambos son mecanismos para reutilizar software disponibles en los lenguajes de POO; sin embargo, mientras que la herencia es un mecanismo propio de la programación orientado a objetos la composición ya se empleaba en la programación estructurada. La composición se suele corresponder con las palabras "tiene un" de la descripción de un problema: un equipo de fútbol tiene un entrenador; un equipo de fútbol tiene varios jugadores; una cuenta bancaria tiene un saldo, etcétera.

### 2.3.1 Sintaxis de la herencia en Java

En Java para indicar que una clase hereda de otras se emplea la palabra reservada `extends` , la sintaxis es:

```
[Modificador] class ClaseHija extends ClasePadre {...}
```

Supongamos que tenemos la clase que se muestra en el listado 6. Esta clase representa un ser vivo. Nuestra clase `Persona` es un ser vivo. Y, por tanto, además de tener una edad y un nombre debería tener la funcionalidad correspondiente con ser vivo (estar vivo, y poder morir en algún momento). Ese código nos resulta útil. Por tanto podemos reutilizarlo mediante la herencia; para ello simplemente indicamos que nuestra clase hereda de la clase `SerVivo`. Para ello declararemos la clase como sigue:

```
public class Persona extends SerVivo
```

Con sólo cambiar esa declaración habremos heredado los dos métodos de la clase padre y sus atributos.

```
//LISTADO 6: Clase SerVivo de la cual hereda Persona
//código SerVivo.java del CD
public class SerVivo{
    private boolean vivo;

    public SerVivo(){
        vivo = true;
    }

    public void morir(){
        vivo = false;
    }
    public boolean isVivo(){
        return vivo;
    }
}
```

Si un método de la clase padre no hace lo que nosotros necesitamos podemos sobrescribirlo (`override`). Para ello basta con volver a definir en la clase hija un método con el mismo nombre que el método de la clase padre. En ocasiones, el problema no es exactamente que el método de la clase padre no haga lo que nosotros queremos, sino que no hace todo lo que nosotros queremos. En ese caso es posible invocar desde el método de la clase hija al método de la clase padre empleando la palabra reservada `super` . `super` es una referencia de sólo lectura que la proporciona de un modo automático el compilador (no necesitamos declararla ningún sitio) y que en una clase siempre apunta a su padre.

En el listado 7 podemos ver una segunda versión de la clase `Persona` (en este caso se llama `Persona2` ) que hereda de la clase `SerVivo` . Para construir un objeto de la clase hija siempre será necesario invocar también a un constructor de la clase padre. Si en los constructores de la clase hija no se indica a qué constructor de la clase padre se debe llamar se llamará al constructor por defecto. Para indicar a qué constructor se debe invocar se emplea la sintaxis `super(argumentos)` , en donde `argumentos` son los argumentos del constructor al que queremos invocar. En nuestro

caso, estamos invocando al constructor por defecto. Si no hubiésemos colocado la sentencia `super()` al principio de cada constructor el compilador lo hubiese hecho por nosotros.

Persona2 sobrescribe el método `morir` de `SerVivo`. En esta ocasión no queremos hacer algo completamente diferente a lo que hace el método del padre, sino que queremos hacer algo más. Queremos imprimir por consola el nombre de la persona que se está muriendo. Por tanto, la primera sentencia del método `morir` en la clase hija lo que hace es invocar al método del padre. Después añade el código nuevo.

//LISTADO 7: Clase Persona2. Los métodos que no se muestran no han cambiado respecto a la clase Persona.

//código Persona2.java del CD

```
public class Persona2 extends SerVivo{
    private int edad;
    private String nombre;

    public Persona2(){
        super();
        edad = 30;
        nombre = "Paco";
    }

    public Persona2(int _edad,String _nombre){
        super();
        edad = _edad;
        nombre = _nombre;
    }

    public void morir(){
        super.morir();
        System.out.println(nombre+"sehamuerto");
    }
    ...
}
```

El tutorial flash del CD muestra cómo emplear BlueJ para comprender mejor los efectos de la herencia.

### 3 Conclusiones

En este tercer artículo de la serie hemos comenzado a presentar el soporte que Java proporciona para la programación orientada a objetos, un paradigma de programación bastante diferente de la programación estructurada. Pero sólo hemos comenzado. En el siguiente artículo de la serie profundizaremos más en varios aspectos relativos a la herencia, presentaremos las interfaces y, finalmente presentaremos los package. Os espero a todos el mes que viene.

Descargas

- [Códigos del artículo](#)
- [Video del artículo \(.exe para Windows\)](#)

- [Video del artículo \(.vvl para Linux\)](#)
- [Video del artículo \(.hqx para Mac Os\)](#)
- [Video del artículo \(.swf, flash multiplataforma\)](#)

### Cápítulos anteriores del curso:

- [Curso de programación Java I](#)
- [Curso de programación Java II](#)