

**OBJETIVOS**

Al finalizar, los estudiantes serán capaces de:

- Definir una breve historia del lenguaje.
- Nombrar las tecnologías de Sun Microsystems
- Identificar los tipos de datos y las estructuras de Control

DESARROLLO**Introducción**

Java es un lenguaje desarrollado por Sun con la intención de competir con Microsoft en el mercado de la red. Sin embargo, su historia se remonta a la creación de una filial de Sun (FirstPerson) enfocada al desarrollo de aplicaciones para electrodomésticos, microondas, lavaplatos, televisiones... Esta filial desapareció tras un par de éxitos de laboratorio y ningún desarrollo comercial.

Sin embargo, para el desarrollo en el laboratorio, uno de los trabajadores de FirstPerson, James Gosling, desarrolló un lenguaje derivado de C++ que intentaba eliminar las deficiencias del mismo. Llamó a ese lenguaje Oak. Cuando Sun abandonó el proyecto de FirstPerson, se encontró con este lenguaje y, tras varias modificaciones (entre ellas la del nombre), decidió lanzarlo al mercado en verano de 1995.

El éxito de Java reside en varias de sus características. Java es un lenguaje sencillo, o todo lo sencillo que puede ser un lenguaje orientado a objetos, eliminando la mayor parte de los problemas de C++, que aportó su granito (o tonelada) de arena a los problemas de C. Es un lenguaje independiente de plataforma, por lo que un programa hecho en Java se ejecutará igual en un PC con Windows que en una estación de trabajo basada en Unix. También hay que destacar su seguridad, desarrollar programas que accedan ilegalmente a la memoria o realizar caballos de troya es una tarea propia de titanes.

Cabe mencionar también su capacidad multihilo, su robustez o lo integrado que tiene el protocolo TCP/IP, lo que lo hace un lenguaje ideal para Internet. Pero es su sencillez, portabilidad y seguridad lo que le han hecho un lenguaje de tanta importancia.

Portabilidad

La portabilidad se consigue haciendo de Java un lenguaje medio interpretado medio compilado. ¿Cómo se come esto? Pues se coge el código fuente, se compila a un lenguaje intermedio cercano al lenguaje máquina pero independiente del ordenador y el sistema operativo en que se ejecuta (llamado en el mundo Java **bytecodes**) y, finalmente, se interpreta ese lenguaje intermedio por medio de un programa denominado máquina virtual de Java.

Este esquema lo han seguido otros lenguajes, como por ejemplo Visual Basic. Sin embargo, nunca se había empleado como punto de partida a un lenguaje multiplataforma ni se había hecho de manera tan eficiente. Cuando Java apareció en el mercado se hablaba de que era entre 10 y 30 veces más lento que C++. Ahora, con los compiladores JIT (**Just in Time**) se habla de tiempos entre 2 y 5 veces más lentos. Con la potencia de las máquinas actuales, esa lentitud es un precio que se puede pagar sin problemas contemplando las ventajas de un lenguaje portable.

Orientación a objetos

Dado que Java es un lenguaje orientado a objetos, es imprescindible entender qué es esto y en qué afecta a nuestros programas. Desde el principio, la carrera por crear lenguajes de programación ha sido una carrera para intentar realizar abstracciones sobre la máquina. Al principio no eran grandes abstracciones y el concepto de lenguajes imperativos es prueba de ello. Exigen pensar en términos del ordenador y no en términos del problema a solucionar. Esto provoca que

los programas sean difíciles de crear y mantener, al no tener una relación obvia con el problema que representan No abstraen lo suficiente.

- **Todo es un objeto.** Cada elemento del problema debe ser modelizado como un objeto.
- **Un programa es un conjunto de objetos diciéndose entre sí que deben hacer por medio de mensajes.** Cuando necesitas que un objeto haga algo le mandas un mensajes. Más concretamente, ejecutas un método de dicho objeto.
- **Cada objeto tiene su propia memoria, que llena con otros objetos.** Cada objeto puede contener otros objetos. De este modo se puede incrementar la complejidad del programa, pero detrás de dicha complejidad sigue habiendo simples objetos.
- **Todo objeto tiene un tipo.** En jerga POO, cada objeto es una instancia (un caso particular) de una clase (el tipo general). Lo que distingue a una clase de otra es la respuesta a la pregunta, ¿qué mensajes puedes recibir?
- **Todos los objetos de un determinado tipo pueden recibir los mismos mensajes.** Por ejemplo, dado que un objeto de tipo Gato es también un objeto de tipo Animal, se puede hacer código pensando en los mensajes que se mandan a un animal y aplicarlo a todos los objetos de ese tipo, sin pensar si son también gatos o no.

Eso de que la programación orientada a objetos intente establecer una relación biunívoca entre el espacio del problema y el de la solución está muy bien pero... ¿cómo hacemos que un objeto realice cosas útiles y necesarias? Por medio de su interfaz, interfaz que está definida en la clase del objeto. Por ejemplo, supongamos que tenemos la clase Interruptor cuyo interfaz define los métodos `apagar()` y `encender()`. Entonces hacemos lo siguiente:

Interfaz de un objeto:

```
Interruptor i = new Interruptor();
i.encender();
```

Primero creamos una referencia llamada `i` de tipo `Interruptor`, y luego creamos un objeto, una instancia de dicho tipo, por medio de la palabra reservada `new`. Asignamos ese objeto a la referencia por medio del signo `=`. Por último, enviamos un mensaje al objeto recién creado y asignado poniendo el nombre de la referencia, un punto, y el nombre del mensaje (método).

Creando nuestra primera clase

Vamos a empezar pronto con las cosas prácticas creando nuestra primera clase y viendo así cómo se programa, compila y ejecuta el código Java. En Java, todo lo que hagamos será programar clases. Cualquier programa será una clase que a su vez podrá hacer uso de una, dos, tres o más clases. Así pues, escribiremos el siguiente código:

HolaMundo.java

```
public class HolaMundo
{
    public static void main(String[] args)
    {
        System.out.println("Hola, mundo");
    }
}
```

Este ejemplo expone varios elementos del lenguaje Java a los que tendremos que prestar atención. Pero lo primero es editar este archivo y grabarlo con el nombre de `MiPrimerPrograma.java`. Si tenemos bien instalado el JDK, no tenemos más que compilarlo escribiendo:

```
javac HolaMundo.java
Y ejecutarlo:
java HolaMundo
```

Una cosa a recordar es que ejecutamos "HolaMundo", no "HolaMundo.class". El intérprete añade la extensión. Veremos que nos muestra el mensaje "Hola, mundo" en la pantalla al ejecutarlo, tal y como cabe esperar de un primer programa en cualquier lenguaje. Vamos a analizarlo línea a línea:

```
public class HolaMundo
{
    .....
}
```

Con esto estamos indicando que todo lo encerrado entre llaves pertenece al código de la clase HolaMundo. En Java, el ámbito de algo está indicado por las llaves. ¿Y qué es el ámbito? El ámbito es lo que determina tanto la visibilidad de la definición de un nombre. Por lo tanto, todo nombre definido entre las llaves arriba indicadas (ya corresponda a una propiedad, variable o método) será visible dentro de esas mismas llaves y dentro de sus llaves "hijas". Por ejemplo:

```
{
  int x1;
  {   int x2;  }
  {   int x3;  }
}
```

En este ejemplo, x1 sería visible en todos los ámbitos, mientras que x2 y x3 sólo se verían en los ámbitos en que se han definido.

Hay que resaltar que hemos especificado que la clase HolaMundo sea pública (public). Con esto cambiamos la accesibilidad de la clase. Podemos hacerla pública, de modo que todo el mundo pueda acceder a ella, sus propiedades y métodos. Podemos hacerla privada (private), que impide que nadie pueda acceder a la misma o protegida (protected), de modo que nadie excepto sus clases heredadas (ya veremos que significa esto) tienen permiso de acceso. Por defecto una clase permite el acceso sólo a miembros de su mismo paquete (package), que es el equivalente en Java a las librerías.

En este caso, podríamos haber dejado el valor de accesibilidad por defecto, pero conviene irse acostumbrando a hacer bien las cosas. Dado que en realidad queremos que la clase HolaMundo sea accesible al mundo exterior la haremos pública. Esto, sin embargo, tiene un par de restricciones que hay que tener en cuenta:

- Sólo puede haber una clase pública por archivo.
- El archivo debe llamarse igual que dicha clase pública.

Continuemos examinando el código fuente:

```
public static void main(String[] args)
{
    .....
}
```

Con esto estamos definiendo un método de la clase HolaMundo. Un método con varias características especiales:

Es público.

Es estático. Esto significa que, por muchas instancias que creamos de la clase, existirá un único miembro de este tipo en la memoria. De hecho éste existe aunque no creamos instancias de la clase y, por tanto, se puede llamar.

Como es un método (función) y no una propiedad (variable), debemos indicar el tipo del valor de retorno. En este caso será void, es decir, ninguno.

El método se llama main. Este nombre es un nombre especial, ya que es el que deberá tener el método que se ejecute cuando se ejecute nuestra aplicación.

Recibe como parámetro un vector de cadenas. Aunque en nuestro programa no lo vamos a utilizar es necesario que así sea, pues este vector recibe los parámetros que haya podido introducir el usuario al programa en la línea de comandos. Por último, vamos a ver lo que realmente hace el programa:

```
System.out.println("Hola, mundo");
```

Llamamos a una función del sistema. Entre las librerías que ofrece Java, existe una que se incluye por defecto en todos los programas, es `java.lang`. Entre las clases disponibles en la misma está la clase estática `System`, uno de cuyos miembros es `out`, uno de cuyos métodos es `println`. Navegando entre la ayuda de Java podemos comprobar todo esto. Este método recibe como parámetro una cadena que se encargará de escribir en pantalla.

Java es un lenguaje cuya sintaxis es similar a C, C++ y Javascript.
 Los comentarios en Java son como los de C++ pero con añadidos.
 Tenemos el comentario que dura hasta final de línea:
`// Esto es un comentario`
 Y el comentario que ocupa varias líneas:
`/* Esto es un
 comentario */`
`** Esto es un comentario
 que servirá para documentar nuestro programa */`

La herramienta `javadoc` recogerá esos comentarios y los incluirá en una documentación en HTML que genera de forma automática. Dado que el resultado es HTML, se puede incluir HTML en los comentarios. Para respetar el modo de comentar de muchos programadores, un asterisco al comienzo de la línea será ignorado. También admite etiquetas especiales que comienzan por una arroba. Por ejemplo, nuestro anterior programa puede ser documentado de la siguiente forma:

HolaMundo.java

```
/** Este es el primer programa en Java
 * Simplemente escribe la frase "Hola, mundo"
 */
public class HolaMundo {
  /** Punto de entrada a la clase y la aplicación
   * @param args vector de cadenas que contiene los argumentos
   * @return No devuelve valor de retorno alguno
   * @exception Ninguna No lanza excepciones
   */
  public static void main(String[] args) {
    System.out.println("Hola, mundo");
  }
}
```

Tipos de datos básicos

En Java todo lo que se mueve es un objeto... excepto los tipos de datos básicos, es decir, los números enteros, reales, los caracteres, los valores lógicos, etc.. Todo lo demás serán objetos o, mejor dicho, referencias a objetos. Son los únicos valores que se crean sin utilizar el operador `new`, que veremos más adelante. Son los siguientes:

TIPO	DESCRIPCIÓN	TAMAÑO	CLASE EQUIVALENTE
<code>boolean</code>	Valor lógico	1 bit	<code>Boolean</code>
<code>char</code>	Carácter	16 bit	<code>Character</code>
<code>byte</code>	Entero muy pequeño	8 bit	
<code>short</code>	Entero pequeño	16 bit	
<code>int</code>	Entero normal	32 bit	<code>Integer</code>

long	Entero grande	64 bit	Long
float	Número real de precisión simple	32 bit	Float
double	Número real de doble precisión	64 bit	Double
void	Tipo vacío		

Podemos crear variables de estos tipos de la manera normal en todos los lenguajes que siguen a C:

```
int numero = 12;
```

El definir un valor al declararlos no es necesario, especialmente porque en Java todos tienen un valor por defecto (0 en los números, false en los booleanos y el carácter número cero); en ese caso escribiríamos simplemente:

```
int numero;
```

En algunas ocasiones, que veremos en el futuro, es necesario utilizar clases que representen los mismos valores que los tipos de datos básicos. En esos casos utilizamos la clase equivalente. En nuestro ejemplo haríamos:

```
Integer numero = new Integer(12);
```

Existen caracteres con significado especial en Java. Son los separadores:

SEPARADOR	DESCRIPCIÓN
()	Contienen listas de parámetros, tanto en la definición de un método como en la llamada al mismo. También se utilizan para modificar la precedencia en una expresión, contener expresiones para control de flujo y realizar conversiones de tipo.
{ }	Se utilizan para definir bloques de código, definir ámbitos y contener los valores iniciales de los vectores.
[]	Se utiliza tanto para declarar vectores o matrices como para referenciar valores dentro de los mismos.
;	Separa sentencias.
,	Separa identificadores consecutivos en la declaración de variables y en las listas de parámetros. También se utiliza para encadenar sentencias dentro de una estructura for.
.	Separa un nombre de propiedad o método de una variable de referencia. También separa nombre de paquete de los de un subpaquete o una clase.

Control de flujo

El más básico, sencillo y utilizado es el if-else. Dado que else es opcional, se puede escribir de dos formas:

```
if (expresión booleana)
    sentencia
o
if (expresión booleana)
    sentencia
else
    sentencia
```

Para bifurcar también existe el switch:

```
switch (expresión) {
    case valor1: sentencia; break;
    case valor2: sentencia; break;
    case valor3: sentencia; break;
    // ...
    default: sentencia;
}
```

Estructuras que nos ofrece Java para realizar iteraciones. Las más genéricas son while y do-while:

```
while (expresión booleana)
    sentencia
y
do
    sentencia;
while (expresión booleana)
```

Otra iteración interesante es el bucle for:

```
for (inicialización; expresión booleana; paso)
{ sentencia }
```

Por último, conviene recordar el uso de return. Se utiliza para devolver un valor de retorno en los métodos (en caso de que éstos no devuelvan void). Pero también conviene destacar que en el momento que se llega a return, el programa sale del método que se esté ejecutando.

Constructores

Con lo que ya sabemos ya estamos preparados para entender y realizar ejemplos más complejos.

Rectangulo.java

```
/** Calcula el area de un rectangulo
 */
public class Rectangulo {
    private float x1,y1,x2,y2;
    public Rectangulo(float ex1, float ey1, float ex2, float ey2) {
        x1 = ex1;
        x2 = ex2;
        y1 = ey1;
        y2 = ey2;
    }
    public float calcularArea() {
        return (x2-x1) * (y2-y1);
    }
    public static void main(String[] args) {
        Rectangulo prueba = new Rectangulo(1,4,7,6);
        System.out.println(prueba.calcularArea());
    }
}
```

Seguimos incluyendo una sola clase en nuestro programa fuente, que guardaremos como Rectangulo.java y que compilaremos y ejecutaremos como vimos hace un par de capítulos. Veremos que nos responde con 12.0. Observando la clase que hemos creado, vemos que tiene cuatro propiedades, de tipo float, y tres métodos. Uno es el ya conocido main, otro parece un método normal y por último tenemos uno que se llama igual que la clase. Es un constructor.

Un constructor es el método que se llama cuando se crea una instancia nueva. Se suele utilizar para inicializar valores. Aquí vemos cómo se crea un objeto de tipo Rectangulo y se asigna a una referencia llamada prueba:

```
Rectangulo prueba = new Rectangulo(1,4,7,6);
```

Como vemos, al crear el objeto colocamos a la derecha de new una llamada al constructor. Si no especificamos ningún constructor, los objetos se crean un constructor por defecto que no admite parámetros y no realiza ninguna tarea de inicialización. Por ejemplo, si no hubiésemos especificado un constructor en nuestro ejemplo, podríamos haber creado un objeto rectángulo de este modo:

```
Rectangulo prueba = new Rectangulo();
```

En cuyo caso tendría todos sus propiedades en su valor inicial, es decir, cero. En el ejemplo vemos que recibe como parámetros las cuatro coordenadas necesarias para definir un rectángulo y las asigna a sus propiedades.

Finalmente, el programa realiza algo útil:

```
System.out.println(prueba.calcularArea());
```

Como vimos, esta llamada sirve para escribir en pantalla lo que le indiquemos. En este caso es una llamada al método de la clase Rectangulo llamado calcularArea(). Este método toma las coordenadas que alberga en las propiedades de la clase y a partir de ellas calcula el área según la fórmula geométrica debida.

Sin embargo, este método devuelve un valor de tipo float, es decir, no devuelve una cadena. ¿Cómo es que nos lo escribe por pantalla? Java lo convierte automáticamente en cadena cuando necesita una cadena. Esta pequeña violación en el estricto sistema de tipos de Java sólo se da con cadenas y más que nada por comodidad.

Herencia

Si se supone que somos buenos programando, cuando creemos una clase es posible que sea algo útil. De modo que cuando estemos haciendo un programa distinto y necesitemos esa clase podremos incluirla en el código de ese nuevo programa. Es la manera más sencilla de reutilizar una clase.

También es posible que utilicemos esa clase incluyendo instancias de la misma en nuevas clases. A eso se le llama composición. Representa una relación "tiene un". Es decir, si tenemos una clase Rueda y una clase Coche, es de esperar que la clase Coche tenga cuatro instancias de Rueda:

```
class Coche {
    Rueda rueda1, rueda2, rueda3, rueda 4;
    ...
}
```

Sin embargo, en ocasiones, necesitamos una relación entre clases algo más estrecha. Una relación del tipo "es un". Por ejemplo, sabemos bien que un gato es un mamífero. Sin embargo es también un concepto más específico, lo que significa que una clase Gato puede compartir con Mamifero propiedades y métodos, pero también puede tener algunas propias.

Herencia.java

```
class Mamifero {
    String especie, color;
}

class Gato extends Mamifero {
    int numero_patas;
}

public class Herencia {
    public static void main(String[] args) {
        Gato bisho;
        bisho = new Gato();
        bisho.numero_patas = 4;
        bisho.color = "Negro";
        System.out.println(bisho.color);
    }
}
```

Como vemos en el ejemplo, el objeto bisho no sólo tiene la propiedad numero_patas, también color que es una propiedad de Mamifero. Se dice que Mamifero es la clase padre y Gato la clase hija en una relación de herencia. Esta relación se consigue en Java por medio de la palabra reservada extends.

Pero, además de heredad la funcionalidad de la clase padre, una clase hija puede sobrescribirla. Podemos escribir un método en la clase hija que tenga el mismo nombre y los mismos parámetros que un método de la clase padre:

Herencia.java

```
class Mamifero {
    String especie, color;
    public void mover() {
        System.out.println("El mamífero se mueve");
    }
}

class Gato extends Mamifero {
    int numero_patas;
    public void mover() {
        System.out.println("El gato es el que se mueve");
    }
}

public class Herencia {
    public static void main(String[] args) {
        Gato bisho = new Gato();
        bisho.mover();
    }
}
```

Al ejecutar esta nueva versión veremos que se escribe el mensaje de la clase hija, no el del padre.

Conviene indicar que Java es un lenguaje en el que todas las clases son heredadas, aún cuando no se indique explícitamente. Hay una jerarquía de objetos única, lo que significa que existe una clase de la cual son hijas todas las demás. Este Adán se llama `Object` y, cuando no indicamos que nuestras clases hereden de nadie, heredan de él. Esto permite que todas las clases tengan algunas cosas en común, lo que permite que funcione, entre otras cosas, el recolector de basura.

Polimorfismo

En muchas ocasiones, cuando utilizamos herencia podemos terminar teniendo una familia de clases que comparten un interfaz común. Por ejemplo, si creamos un nuevo archivo que contenga a `Mamifero` y `Gato` le añadimos:

Polimorfismo.java

```
class Perro extends Mamifero {
    int numero_patas;
    public void mover() {
        System.out.println("Ahora es un perro el que se mueve");
    }
}

public class Polimorfismo {
    public static void muevete(Mamifero m) {
        m.mover();
    }
    public static void main(String[] args) {
        Gato bisho = new Gato();
        Perro feo = new Perro();
        muevete(bisho);
        muevete(feo);
    }
}
```


Vemos que el método muevete llama al método mover de un mamífero. El no sabe con qué clase de mamífero trata, pero la cosa funciona y se llama al método correspondiente al objeto específico que lo llama (es decir, primero un gato y luego un perro). Y esto no sólo se aplica a los métodos. Por ejemplo, podemos reescribir el código del procedimiento principal:

```
public static void main(String[] args) {
    Mamifero bisho = new Gato();
    muevete(bisho);
    bisho = new Perro();
    muevete(bisho);
}
```

Y vemos que funciona exactamente igual. El polimorfismo consiste en que toda referencia a un objeto de una clase específica puede tomar la forma de una referencia a un objeto de una clase heredada a la suya.

Sobrecarga de métodos

El concepto de polimorfismo, en cuanto a cambio de forma, se puede extender a los métodos. Java permite que varios métodos dentro de una clase se llamen igual, siempre y cuando su lista de parámetros sea distinta. Por ejemplo, si tuviéramos un método que sirviera para sumar números pero sin indicar de qué tipo:

Sumar.java

```
/** Diversos modos de sumar */
public class Sumar {
    public float suma(float a, float b) {
        System.out.println("Estoy sumando reales");
        return a+b;
    }
    public int suma(int a, int b) {
        System.out.println("Estoy sumando enteros");
        return a+b;
    }
    public static void main(String[] args) {
        float x = 1, float y = 2;
        int v = 3, int w = 5;
        System.out.println(suma(x,y));
        System.out.println(suma(v,w));
    }
}
```

Esto también se aplica a los constructores. De hecho, es la aplicación más habitual de la sobrecarga:

Rectangulo.java

```
/** Calcula el area de un rectángulo */
public class Rectangulo {
    private float x1,y1,x2,y2;
    public Rectangulo(float ex1, float ey1, float ex2, float ey2) {
        x1 = ex1;
        x2 = ex2;
        y1 = ey1;
        y2 = ey2;
    }
    public Rectangulo() {
        x1 = 0;
        x2 = 0;
        y1 = 1;
    }
}
```

```

    y2 = 1;
}
public float calcularArea() {
    return (x2-x1) * (y2-y1);
}
public static void main(String[] args) {
    Rectangulo prueba1 = new Rectangulo(1,4,7,6);
    Rectangulo prueba2 = new Rectangulo();
    System.out.println(prueba1.calcularArea());
    System.out.println(prueba2.calcularArea());
}
}

```

Paquetes

Los paquetes, no son más que un contenedor de clases. Que actúan como de librerías y existen más que nada por comodidad. Para agrupar varios archivos fuente de Java en un sólo paquete incluiremos una línea al comienzo de los mismos:

```
package programacionIV;
```

Y el código resultante (los archivos con extensión class) se situaría en el directorio programacionIV. Pertencerían al paquete programacionIV. Para poder utilizarlos desde otro programa deberíamos utilizar la sentencia import, que también se coloca al principio del programa (aunque después de package):

```
import .prograIV.net.*;
```

Hay que indicar que import es una palabra clave que indica las clases que deseamos cargar, no los paquetes. Sin embargo, al admitir el uso del comodín *, se puede utilizar para cargar paquetes enteros. Por ejemplo, si deseamos utilizar la clase Date, situada en el paquete java.util, podemos cargarla de dos maneras:

```
import java.util.Date;
import java.util.*;
```

Algunos paquetes de Java

Existen muchos paquetes en la librería estándar de Java, paquetes que, además, han ido variando según se sucedían las versiones del mismo. Así, pues, vamos a destacar algunos paquetes presentes en todos los JDK:

java.lang

Este paquete incluye las clases imprescindibles para que el lenguaje Java funcione como tal, es decir, clases como Object, Thread, Exception, System, Integer, Float, Math, Package, String, etc., que implementan la base del lenguaje. No es necesario importar nada desde ese paquete porque se carga por defecto.

java.util

Este paquete es el segundo en importancia, ya que incluye muchas clases útiles como pueda ser Date (fecha) y, sobre todo, diversas clases que permiten el almacenamiento dinámico de información, como Vector, LinkedList, HashMap, etc..

java.applet

Contiene la archifamosa clase Applet, que nos permite crear applets para verlos en nuestro navegador.

java.awt

Este paquete, cuyo nombre corresponde a las siglas de Abstract Windowing Toolkit, contiene las clases necesarias para crear interfaces de usuario, por medio de menús, botones, áreas de texto, cajas de confirmación, etc..

java.io

Este paquete contiene las clases necesarias para realizar las operaciones de entrada/salida, ya sea a pantalla o a archivos, clases heredadas de `FileInputStream` y `FileOutputStream`.

java.net

Paquete que permite la programación de aplicaciones que accedan a bajo nivel a redes TCP/IP.

Control de acceso

Ha llegado el momento de especificar lo que en el primer ejemplo llamamos control de acceso. Cuando se crea una nueva clase en Java, se puede especificar el nivel de acceso que se quiere para las variables de instancia y los métodos definidos en la clase por medio de los siguientes modificadores de acceso:

public

Cualquier clase puede acceder a las propiedades y métodos públicos.

protected

Sólo las clases heredadas y aquellas situadas en el mismo paquete pueden acceder a las propiedades y métodos protegidos.

private

Las variables y métodos privados sólo pueden ser accedidos desde dentro de la clase.

"friendly" (valor por defecto)

Si no se especifica ningún modificador de acceso, las variables y métodos se declaran "friendly" (amigos), lo que en la práctica significa lo mismo que `protected`.

private protected

Sólo las clases heredadas pueden acceder a las propiedades y métodos protegidos.