Apuntes de ~~C#~~ Java

Apuntes de Java

Contenido

Variables y Tipos de Datos

Una **variable** puede verse simplemente como un almacén de objetos de un determinado tipo al que se le da un cierto nombre. Por tanto, para definir una variable sólo hay que decir cuál será el nombre que se le dará y cuál será el tipo de datos que podrá almacenar, lo que se hace con la siguiente sintaxis:

```
<tipoVariable> <nombreVariable>;
```

C# también proporciona una sintaxis más sencilla con la que podremos asignar un valor a una variable en el mismo momento se define. Para ello se la ha de definir usando esta otra notación:

```
<tipoVariable> <nombreVariable> = <valorInicial>;
```

Tipos de datos básicos (o primitivos)

Los **tipos de datos básicos** son ciertos tipos de datos tan comúnmente utilizados en la escritura de aplicaciones que en Java se ha incluido una sintaxis especial para tratarlos.

```
int a = 2;
```

Los tipos de datos básicos en Java son:

Tipo	**Descripción**	**Bits**	**Rango de valores**
byte	*Bytes* con signo	8	-128 – 127
short	Enteros cortos con signo	16	[-32.768, 32.767]
int	Enteros normales	32	[-2.147.483.648, 2.147.483.647]
long	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]
float	Reales con 7 dígitos de precisión	32	$[1{,}5\times10^{-45}$ - $3{,}4\times10^{38}]$
double	Reales de 15-16 dígitos de precisión	64	$[5{,}0\times10^{-324}$ - $1{,}7\times10^{308}]$
boolean	Valores lógicos	32	**true**, **false**
char	Caracteres Unicode	1	[‘\u0000’, ‘\uFFFF’]
String	Cadenas de caracteres	Variable	El permitido por la memoria

El valor que por defecto se da a los campos de tipos básicos consiste en poner a cero toda el área de memoria que ocupen. Esto se traduce en que los campos de tipos básicos numéricos se inicializan por defecto con el valor 0, los de tipo **bool** lo hacen con **false**, los de tipo **char** con ‘\u0000’, y los de tipo **String** con **null**.

Entrada y salida básica

Para poder mostrar información al usuario que vaya a usar nuestro programa y recibir datos de ese usuario necesitaremos una serie de funciones.

Salida por pantalla

Para mostrar datos por pantalla, usaremos las funciones: **System.out.print()** y **System.out.println()**. Las dos funciones mostrarán por pantalla el valor (ya sea numérico o de otro tipo) que pongamos entre los paréntesis. La diferencia entre ambas es que **println** salta a la siguiente línea después de escribir y **print** no lo hace.

```
int a = 2;
System.out.println(a);
```

Este ejemplo nos mostrará por pantalla el número 2 (que es el valor de a).

Entrada por teclado

En ocasiones, necesitaremos que el usuario nos envíe datos para procesarlos y generar un resultado. Leer datos desde el teclado no es especialmente sencillo y además hay varias opciones. La que normalmente se recomienda por su sencillez es usando la clase **Scanner**.

Para usar este método, primero tendremos que colocar la línea: **import java.util.Scanner** justo al principio de nuestro programa, debajo de la primera línea "**package**". Una vez hecho esto, dentro de la función **main**, tendremos que crear una variable de tipo Scanner. Y por fin, podremos leer valores usando esta variable. A continuación, veremos un ejemplo en el que se lee un valor y se guarda en la variable **a**.

```
package com.company;
import java.util.Scanner;

public class Main
{
	public static void main(String[] args)
	{
		int a;
		Scanner sc = new Scanner(System.in);
		System.out.println("Escribe un número");
		a = sc.nextInt();
	}
}
```

Instrucciones

Concepto de instrucción

Toda acción que se pueda realizar en el cuerpo de un método, como definir variables locales, llamar a métodos, asignaciones y muchas cosas más que veremos a lo largo de este tema, son **instrucciones**.

Las instrucciones se agrupan formando **bloques de instrucciones**, que son listas de instrucciones encerradas entre llaves que se ejecutan una tras otra. Es decir, la sintaxis que se sigue para definir un bloque de instrucciones es:

```
{
      <listaInstrucciones>
}
```

Toda variable que se defina dentro de un bloque de instrucciones sólo existirá dentro de dicho bloque. Tras él será inaccesible y podrá ser destruida por el recolector de basura. Por ejemplo, este código no es válido:

```
public void f();
{
      { int b; }
      b = 1; // ERROR: b no existe fuera del bloque donde se declaró.
}
```

Los bloques de instrucciones pueden anidarse, aunque si dentro de un bloque interno definimos una variable con el mismo nombre que otra definida en un bloque externo se considerará que se ha producido un error, ya que no se podrá determinar a cuál de las dos se estará haciendo referencia cada vez que se utilice su nombre en el bloque interno.

Instrucciones básicas

Definiciones de variables locales

Las **variables locales** son variables que se definen en el cuerpo de los métodos y sólo son accesibles desde dichos cuerpos. La sintaxis para definirlas es la siguiente:

<*modificadores*> <tipoVariable> <nombreVariable> **=** <*valor*>**;**

También pueden definirse varias variables en una misma instrucción separando sus pares nombre-valor mediante comas. Por ejemplo:

```
int a=5, b, c=-1;
```

Asignaciones

Una **asignación** es simplemente una instrucción mediante la que se indica un valor a almacenar en un dato. La sintaxis usada para ello es:

<destino> = <origen>;

Instrucciones condicionales

Las **instrucciones condicionales** son instrucciones que permiten ejecutar bloques de instrucciones sólo si se da una determinada condición. En los siguientes subapartados de este epígrafe se describen cuáles son las instrucciones condicionales disponibles en C#

Instrucción if

La **instrucción if** permite ejecutar ciertas instrucciones sólo si se da una determinada condición. Su sintaxis de uso es la sintaxis:

```
if (<condición>)
        <instruccionesIf>
else
        <instruccionesElse>
```

El significado de esta instrucción es el siguiente: se evalúa la expresión <condición>, que ha de devolver un valor lógico. Si es cierta (devuelve **true**) se ejecutan las <instruccionesIf>, y si es falsa (**false**) se ejecutan las <instruccionesElse> La rama **else** es opcional, y si se omite y la condición es falsa se seguiría ejecutando a partir de la instrucción siguiente al **if**. En realidad, tanto <instruccionesIf> como <instruccionesElse> pueden ser una única instrucción o un bloque de instrucciones.

Instrucción switch

La **instrucción switch** permite ejecutar unos u otros bloques de instrucciones según el valor de una cierta expresión. Su estructura es:

```
switch (<expresión>)
{
        case <valor1>: <bloque1>
                break;
        case <valor2>: <bloque2>
                break;
        ...
        default: <bloqueDefault>
                break;
}
```

El significado de esta instrucción es el siguiente: se evalúa <expresión>. Si su valor es <valor1> se ejecuta el <bloque1>, si es <valor2> se ejecuta <bloque2>, y así para el resto de valores especificados. Si no es igual a ninguno de esos valores y se incluye la rama **default**, se ejecuta el <bloqueDefault>; pero si no se incluye se pasa directamente a ejecutar la instrucción siguiente al **switch**.

Los valores indicados en cada rama del **switch** han de ser expresiones constantes que produzcan valores de algún tipo básico entero, de una enumeración, de tipo **char** o de tipo **string**. Además, no puede haber más de una rama con el mismo valor. En realidad, aunque todas las ramas de un **switch** son opcionales siempre se ha de incluir al menos una. Además, la rama **default** no tiene porqué aparecer la última si se usa, aunque es recomendable que lo haga para facilitar la legibilidad del código.

Instrucciones iterativas

Las **instrucciones iterativas** son instrucciones que permiten ejecutar repetidas veces una instrucción o un bloque de instrucciones mientras se cumpla una condición. Es decir, permiten definir bucles donde ciertas instrucciones se ejecuten varias veces. A continuación, se describen cuáles son las instrucciones de este tipo incluidas en C#.

Instrucción while

La **instrucción while** permite ejecutar un bloque de instrucciones mientras se dé una cierta instrucción. Su sintaxis de uso es:

while (<condición>**)**
    <instrucciones>

Su significado es el siguiente: Se evalúa la <condición> indicada, que ha de producir un valor lógico. Si es cierta (valor lógico **true**) se ejecutan las <instrucciones> y se repite el proceso de evaluación de <condición> y ejecución de <instrucciones> hasta que deje de serlo. Cuando sea falsa (**false**) se pasará a ejecutar la instrucción siguiente al **while**. En realidad <instrucciones> puede ser una única instrucción o un bloque de instrucciones.

Por otro lado, dentro de las <instrucciones> de un **while** pueden usarse dos instrucciones especiales:

- **break;**: Indica que se ha de abortar la ejecución del bucle y continuarse ejecutando por la instrucción siguiente al **while**.
- **continue;**: Indica que se ha de abortar la ejecución de las <instrucciones> y reevaluarse la <condición> del bucle, volviéndose a ejecutar la <instrucciones> si es cierta o pasándose a ejecutar la instrucción siguiente al **while** si es falsa.

Instrucción do…while

La instrucción **do...while** es una variante del **while** que se usa así:

do
    <instrucciones>
while(<condición>**);**

La única diferencia del significado de **do...while** respecto al de **while** es que en vez de evaluar primero la condición y ejecutar <instrucciones> sólo si es cierta, **do...while** primero ejecuta las <instrucciones> y luego mira la <condición> para ver si se ha de repetir la ejecución de las mismas. Por lo demás ambas instrucciones son iguales, e incluso también puede incluirse **break;** y **continue;** entre las <instrucciones> del **do...while**. **do ... while** está especialmente destinado para los casos en los que haya que ejecutar las <instrucciones> al menos una vez aun cuando la condición sea falsa desde el principio.,

Instrucción for

La **instrucción for** es una variante de **while** que permite reducir el código necesario para escribir los tipos de bucles más comúnmente usados en programación. Su sintaxis es:

for (<inicialización>**;** <condición>**;** <modificación>**)**
    <instrucciones>

El significado de esta instrucción es el siguiente: se ejecutan las instrucciones de <inicialización>, que suelen usarse para definir e inicializar variables que luego se usarán en <instrucciones>. Luego se evalúa <condición>, y si es falsa se continúa ejecutando por la instrucción siguiente al **for**; mientras que si es cierta se ejecutan las <instrucciones> indicadas, luego se ejecutan las instrucciones de <modificación> -que como su nombre indica suelen usarse para modificar los valores de variables que se usen en <instrucciones>- y luego se reevalúa <condición> repitiéndose el proceso hasta que ésta última deje de ser cierta.

En <inicialización> puede en realidad incluirse cualquier número de instrucciones que no tienen por qué ser relativas a inicializar variables o modificarlas, aunque lo anterior sea su uso más habitual. En caso de ser varias se han de separar mediante comas (,), ya que el carácter de punto y coma (;) habitualmente usado para estos menesteres se usa en el **for** para separar los bloques de <inicialización>, <condición> y <modificación> Además, la instrucción nula no se puede usar en este caso y tampoco pueden combinarse definiciones de variables con instrucciones de otros tipos.
Con <modificación> pasa algo similar, ya que puede incluirse código que nada tenga que ver con modificaciones, pero en este caso no se pueden incluir definiciones de variables. Como en el resto de instrucciones hasta ahora vistas, en <instrucciones> puede ser tanto una única instrucción como un bloque de instrucciones. Además, las variables que se definan en <inicialización> serán visibles sólo dentro de esas <instrucciones>

Al igual que con **while**, dentro de las <instrucciones> del **for** también pueden incluirse instrucciones **continue;** y **break;** que puedan alterar el funcionamiento normal del bucle.

Instrucción for-each

Existe en Java una variante de la **instrucción for** pensada especialmente para compactar la escritura de códigos donde se realice algún tratamiento a todos los elementos de una colección, que suele un uso muy habitual de **for** en los lenguajes de programación que lo incluyen. La sintaxis que se sigue a la hora de escribir esta instrucción **for-each** es:

for (<tipoElemento> <elemento> **:** <colección>**)**
    <instrucciones>

El significado de esta instrucción es muy sencillo: se ejecutan **<instrucciones>** para cada uno de los elementos de la <colección> indicada. <elemento> es una variable de sólo lectura de tipo <tipoElemento> que almacenará en cada momento el elemento de la colección que se esté procesando y que podrá ser accedida desde <instrucciones>.

Funciones

Concepto de función

Una **función** es un conjunto de instrucciones a las que se les da un determinado nombre de tal manera que sea posible ejecutarlas en cualquier momento sin tenerlas que reescribir sino usando sólo su nombre. A estas instrucciones se les denomina **cuerpo** de la función, y a su ejecución a través de su nombre se le denomina **llamada** a la función.

La ejecución de las instrucciones de una función puede producir como resultado un objeto de cualquier tipo. A este objeto se le llama **valor de retorno** del método y es completamente opcional, pudiéndose escribir funciones que no devuelvan ninguno.

La ejecución de las instrucciones de una función puede depender del valor de unas variables especiales denominadas **parámetros** de la función, de manera que en función del valor que se dé a estas variables en cada llamada la ejecución de la función se pueda realizar de una u otra forma y podrá producir uno u otro valor de retorno.

Al conjunto formado por el nombre de una función y el número y tipo de sus parámetros se le conoce como **signatura** de la función. La signatura de una función es lo que verdaderamente la identifica, de modo que es posible definir en un mismo tipo varias funciones con idéntico nombre siempre y cuando tengan distintos parámetros. Cuando esto ocurre se dice que la función que tiene ese nombre está **sobrecargada**.

Definición de funciones

Para definir una función hay que indicar tanto cuáles son las instrucciones que forman su cuerpo como cuál es el nombre que se le dará, cuál es el tipo de objeto que puede devolver y cuáles son los parámetros que puede tomar. Esto se indica definiéndolo así:

```
<tipoRetorno> <nombreFunción>(<parámetros>)
{
        <cuerpo>
}
```

En <tipoRetorno> se indica cuál es el tipo de dato del objeto que la función devuelve, y si no devuelve ninguno se ha de escribir **void** en su lugar.

Aunque es posible escribir funciones que no tomen parámetros, si una función los toma se ha de indicar en *<parámetros>* cuál es el nombre y tipo de cada uno de ellos, separándolos con comas si son más de uno y siguiendo la sintaxis que más adelante se explica.

El *<cuerpo>* de la función también es opcional, pero si la función retorna algún tipo de objeto entonces ha de incluir al menos una instrucción **return** que indique cuál es el objeto a devolver.

A continuación, se muestra un ejemplo de cómo definir un método de nombre Saluda cuyo cuerpo consista en escribir en la consola el mensaje "Hola Mundo" y que devuelva un objeto **int** de valor 1:

```
int Saluda()
{
      System.out.println("Hola Mundo");
      return 1;
}
```

Llamada a funciones

La forma en que se puede llamar a una función depende del tipo de función de la que se trate.
<nombreMétodo>**(**<valoresParámetros>**)**

Tipos de parámetros. Sintaxis de definición

La forma en que se define cada parámetro de una función depende del tipo de parámetro del que se trate. En C# se admiten cuatro tipos de parámetros: parámetros de entrada, parámetros de salida, parámetros por referencia y parámetros de número indefinido.

Parámetros de entrada

Un **parámetro de entrada** recibe una copia del valor que almacenaría una variable del tipo del objeto que se le pase. Por tanto, si el objeto es de un tipo valor se le pasará una copia del objeto y cualquier modificación que se haga al parámetro dentro del cuerpo del la función no afectará al objeto original sino a su copia; mientras que si el objeto es de un tipo referencia entonces se le pasará una copia de la referencia al mismo y cualquier modificación que se haga al parámetro dentro de la función también afectará al objeto original ya que en realidad el parámetro referencia a ese mismo objeto original. Para definir un parámetro de entrada basta indicar cuál el nombre que se le desea dar y el cuál es tipo de dato que podrá almacenar. Para ello se sigue la siguiente sintaxis:

<tipoParámetro> <nombreParámetro>

Por ejemplo, el siguiente código define una función llamada Suma que toma dos parámetros de entrada de tipo **int** llamados par1 y par2 y devuelve un **int** con su suma:

```
int Suma(int par1, int par2)
{
      return par1+par2;
}
```

Como se ve, se usa la instrucción **return** para indicar cuál es el valor que ha de devolver la función. Este valor es el resultado de ejecutar la expresión par1+par2; es decir, es la suma de los valores pasados a sus parámetros par1 y par2 al llamarla.
En las llamadas a funciones se expresan los valores que se deseen dar a este tipo de parámetros indicando simplemente el valor deseado. Por ejemplo, para llamar a la función anterior con los valores 2 y 5 se haría <objeto>.Suma(2,5), lo que devolvería el valor 7.

Arrays

Un **array unidimensional** es un tipo especial de variable que es capaz de almacenar en su interior y de manera ordenada uno o varios datos de un determinado tipo. Para declarar variables de este tipo especial se usa la siguiente sintaxis:

```
<tipoDatos>[] <nombreArray>;
```

Por ejemplo, un array que pueda almacenar objetos de tipo **int** se declara así:

```
int[] array;
```

Con esto el array creado no almacenaría ningún objeto, sino que valdría **null**. Si se desea que verdaderamente almacene objetos hay que indicar cuál es el número de objetos que podrá almacenar, lo que puede hacerse usando la siguiente sintaxis al declararla:

```
<tipoDatos>[] <nombreArray> = new <tipoDatos>[<númeroDatos>];
```

Por ejemplo, un array que pueda almacenar 100 objetos de tipo **int** se declara así:

```
int[] array = new int[100];
```

Aunque también sería posible definir el tamaño del array de forma separada a su declaración de este modo:

```
int[] array;
array = new int[100];
```

Con esta última sintaxis es posible cambiar dinámicamente el número de elementos de una variable array sin más que irle asignando nuevos arrays. Ello no significa que un array se pueda redimensionar conservando los elementos que tuviese antes del cambio de tamaño, sino que ocurre todo lo contrario: cuando a una variable array se le asigna un array de otro tamaño, sus elementos antiguos son sobrescritos por los nuevos. Si se crea un array con la sintaxis hasta ahora explicada todos sus elementos tendrían el valor por defecto de su tipo de dato. Si queremos darles otros valores al declarar el array, hemos de indicarlos entre llaves usando esta sintaxis:

```
<tipoDatos>[] <nombreArray> = {<valores>};
```

Ha de especificarse tantos <valores> como número de elementos se desee que tenga el array, y si son más de uno se han de separar entre sí mediante comas (,) Nótese que ahora no es necesario indicar el número de elementos del array, pues el compilador puede deducirlo del número de valores especificados. Por ejemplo, para declarar un array de cuatro elementos de tipo **int** con valores 5,1,4,0
se podría hacer lo siguiente:

```
int[] array = {5,1,4,0};
```

También podemos crear arrays cuyo tamaño se pueda establecer dinámicamente a partir del valor de cualquier expresión que produzca un valor de tipo entero. Por ejemplo, para crear un array cuyo tamaño sea el valor indicado por una variable de tipo **int** (luego su valor será de tipo entero) se haría:

```
int i = 5;
...
int[] arrayDinamico = new int[i];
```

A la hora de acceder a los elementos almacenados en un array basta indicar entre corchetes, y a continuación de la referencia a la misma, la posición que ocupe en el array el elemento al que acceder. Cuando se haga hay que tener en cuenta que en Java los arrays se indexan desde 0, lo que significa que el primer elemento del array ocupará su posición 0, el segundo ocupará la posición 1, y así sucesivamente para el resto de elementos. Por ejemplo, aunque es más ineficiente, el array declarada en el último fragmento de código de ejemplo también podría haberse definido así:

```
int[] array = new int[4];
array[0] = 5;
array[1]++;
// Por defecto se inicializó a 0, luego ahora el valor de array[1]
// pasa a ser 1
array[2] = array[0] - array[1];
// array[2] pasa a valer 4, pues 5-4 = 1
// El contenido de la array será {5,1,4,0}, pues array[3] se
// inicializó por defecto a 0.
```

Hay que tener cuidado a la hora de acceder a los elementos del array ya que si se especifica una posición superior al número de elementos que pueda almacenar el array se producirá una excepción de tipo **java.lang.arrayindexoutofboundsexception.**

Para evitar este tipo de excepciones puede consultar el valor del campo de sólo lectura **length** que está asociado a todo array y contiene el número de elementos del mismo. Por ejemplo, para asignar un 7 al último elemento del array anterior se haría:

```
array[array.length - 1] = 7;
// Se resta 1 porque array.length devuelve 4 pero el último
// elemento del array es array[3]
```

Arrays multidimensionales

Un **array multidimensional** es un array cuyos elementos se encuentran organizando una estructura de varias dimensiones. Para definir este tipo de arrays se usa una sintaxis similar a la usada para declarar arrays unidimensionales, pero colocando cada nueva dimensión entre corchetes. Por ejemplo, un array multidimensional de elementos de tipo **int** que conste de 12 elementos puede tener sus elementos distribuidos en dos dimensiones formando una estructura 3x4 similar a una matriz de la forma:

```
1 2  3  4
5 6  7  8
9 10 11 12
```

Este array se podría declarar así:

```
int[][] arrayMultidimensional = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

Si no queremos indicar explícitamente los elementos del array al declararlo podemos obviarlos, pero aun así indicar el tamaño de cada dimensión del array (a los elementos se les daría el valor por defecto de su tipo de dato) así:

```
int[][] arrayMultidimensional = new int[3][4];
```

También podemos no especificar ni siquiera el número de elementos del array de esta forma (arrayMultidimensional contendría ahora **null**):

```
int[][] arrayMultidimensional;
```

Aunque los ejemplos de arrays multidimensionales hasta ahora mostrados son de arrays de dos dimensiones, en general también es posible crear arrays de cualquier número de dimensiones. Por ejemplo, un array que almacene 24 elementos de tipo **int** y valor 0 en una estructura tridimensional 3x4x2 se declararía así:

```
int[][][] arrayMultidimensional = new int[3][4][2];
```

El acceso a los elementos de un array multidimensional es muy sencillo: sólo hay que indicar los índices de la posición que ocupe en la estructura multidimensional el elemento al que se desee acceder. Por ejemplo, para incrementar en una unidad el elemento que ocupe la posición (1,3,2) del array anterior se haría (se indexa desde 0):

```
arrayMultidimensional[0][2][1]++;
```

Tamaños de un array multidimensional

Obtener el tamaño de las diferentes dimensiones de un array multidimensional es un poco más complicado que en un array unidimensional, puesto que vamos a tener varias longitudes para cada dimensión.
Por ejemplo, si tenemos un array de dos dimensiones como éste:

```
int[][] arrayMultidimensional = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

Para obtener el tamaño de la primera dimensión (que es 3) escribiríamos:

```
int dimension1 = arrayMultidimensional.length;
```

Para obtener el tamaño de la segunda dimensión (que en este caso es 4) tendríamos que escribir:

```
int dimension2 = arrayMultidimensional[0].length;
```

En este caso podemos escribir entre los corchetes cualquiera de las filas del array, porque todas las columnas tienen el mismo tamaño. En el caso en el que estemos usando un array con filas de diferente tamaño, tendríamos que ir leyendo la longitud de cada fila usando **array[fila].length**.

Cadenas de texto

Una **cadena de texto** no es más que una secuencia de caracteres Unicode. En Java se representan mediante objetos del tipo tipo de dato llamado **String**.
Las cadenas de texto suelen crearse a partir literales de cadena o de otras cadenas previamente creadas. Ejemplos de ambos casos se muestran a continuación:

```
String cadena1 = "José Antonio";
String cadena2 = cadena;
```

En el primer caso se ha creado un objeto **string** que representa a la cadena formada por la secuencia de caracteres `José Antonio` indicada literalmente (nótese que las comillas dobles entre las que se encierran los literales de cadena no forman parte del contenido de la cadena que representan, sino que sólo se usan como delimitadores de la misma). En el segundo caso la variable cadena2 creada se genera a partir de la variable cadena1 ya existente, por lo que ambas variables apuntarán al mismo objeto en memoria.

Al igual que el significado del operador **==** ha sido especialmente modificado para trabajar con cadenas, lo mismo ocurre con el operador binario **+**. En este caso, cuando se aplica entre dos cadenas o una cadena y un carácter lo que hace es devolver una nueva cadena con el resultado de concatenar sus operandos. Así por ejemplo, en el siguiente código las dos variables creadas almacenarán la cadena "Hola Mundo":

```
public static void main(String[] args)
{
      String cadena = "Hola" + " Mundo";
      String cadena2 = "Hola Mund" + 'o';
}
```

Por otro lado, el acceso a las cadenas se hace de manera similar a como si de un array de caracteres se tratase: su función **length** almacenará el número de caracteres que la forman y para acceder a sus elementos se utiliza el método **charAt()**. Por ejemplo, el siguiente código muestra por pantalla cada carácter de la cadena `Hola` en una línea diferente:

```
public static void main(String[] args)
{
      String cadena = "Hola";
      System.out.println(cadena.charAt(0));
      System.out.println(cadena.charAt(1));
      System.out.println(cadena.charAt(2));
      System.out.println(cadena.charAt(3));
}
```

Sin embargo, hay que señalar una diferencia importante respecto a la forma en que se accede a las tablas: las cadenas son inmutables, lo que significa que no es posible modificar los caracteres que las forman.

Aparte de los métodos ya vistos, en la clase **String** se definen muchos otros métodos aplicables a cualquier cadena y que permiten manipularla. Los principales son:

- **int indexOf(String subcadena)**: Indica cuál es el índice de la primera aparición de la subcadena indicada dentro de la cadena sobre la que se aplica. La búsqueda de dicha subcadena se realiza desde el principio de la cadena, pero es posible indicar en un segundo parámetro opcional de tipo **int** cuál es el índice de la misma a partir del que se desea empezar a buscar. Del mismo modo, la búsqueda acaba al llegar al final de la cadena sobre la que se busca, pero pasando un tercer parámetro opcional de tipo **int** es posible indicar algún índice anterior donde terminarla.
  Nótese que es un método muy útil para saber si una cadena contiene o no alguna subcadena determinada, pues sólo si no la encuentra devuelve un **–1**.

- **int lastIndexOf(String subcadena)**: Funciona de forma similar a **indexOf()** sólo que devuelve la posición de la última aparición de la subcadena buscada en lugar de devolver la de la primera.

- **String replace(String aSustituir, String sustituta)**: Devuelve la cadena resultante de sustituir en la cadena sobre la que se aplica toda aparación de la cadena a Sustituir indicada por la cadena sustituta especificada como segundo parámetro.

- **String substring(int posiciónInicio, int posiciónFinal)**: Devuelve la subcadena de la cadena sobre la que se aplica que comienza en la posición inicial (incluida) y llega hasta la posición final (que no va incluida). Si no se indica la posición final se devuelve la subcadena que va desde la posición inicial hasta el final de la cadena.

- **String toUpperCase()** y **String toLowerCase()**: Devuelven, respectivamente, la cadena que resulte de convertir a mayúsculas o minúsculas la cadena sobre la que se aplican.

- **int compareTo(String cadena2)**: Nos compara una nuestra cadena con la cadena2. Si nuestra cadena es más pequeña (en orden alfabético) nos devolverá un número negativo. Si son iguales, devolverá 0. Y si la segunda cadena es más pequeña, devolverá un número positivo.

```
if(cadena1.compareTo(cadena2) == 0)
{
    System.out.println("Las cadenas son iguales");
}
```

La clase StringBuilder

En Java existe también la clase **StringBuilder**. Esta clase es similar a la clase **String** y nos permitirá modificar el contenido de nuestra cadena. Como en la mayoría de los casos el resultado final lo querremos como **String**, lo que haremos si necesitamos modificar una cadena es crear un **StringBuilder** con nuestra cadena, modificarla y después convertirlo a **String** de nuevo para dar el resultado.

```
String s = "Ejemplo";
StringBuilder sb = new StringBuilder(s);
sb.reverse();
s = sb.toString();
```

La clase **StringBuilder** tiene las siguientes funciones que no posee un **String**:

- **StringBuilder append(String s)**: Añade la cadena que le pasamos al final de nuestro **StringBuilder**. Es similar a usar el operador **+** con **Strings**, pero es mucho más eficiente por lo que es preferible usarlo si vamos a hacer muchas operaciones.

- **StringBuilder insert(int posición, String s)**: Inserta la cadena que le pasamos en la posición indicada dentro de nuestro **StringBuilder**. Es similar a **append** pero en lugar de añadir la cadena al final la puede poner al principio (si especificamos la posición 0) o en la posición intermedia que especifiquemos.

- **StringBuilder delete(int posiciónInicio, int posiciónFinal)**: Elimina los caracteres de nuestro **StringBuilder** que se encuentran entre la posición de inicio (incluida) y la posición final (no incluida).

Listas

Una **lista** es una colección que nos permite guardar elementos del mismo tipo, de la misma forma que hace un **array**. Al contrario que en los **arrays**, las listas no tienen un tamaño definido, sino que van creciendo a medida que se les añade elementos nuevos.

Para definir una lista, usaremos la sintaxis:

```
List<<tipoDatos>> <nombreLista>;

Ej.: List<Integer> lista;
```

Además de definirla, deberemos crearla para poderla usar. En Java, existen varias implementaciones diferentes de una lista, que se diferencian en la forma en la que se almacenan internamente los datos. Las dos principales implementaciones son **ArrayList** (donde los elementos se almacenan en un array) y **LinkedList** (donde los elementos se almacenan en una lista enlazada). La diferencia entre ambas implementaciones es que algunas operaciones son más rápidas en una y más lentas en la otra.

```
<nombreLista> = new ArrayList<<tipoDatos>>();

<nombreLista> = new LinkedList<<tipoDatos>>();

Ej.: lista = new ArrayList<Integer>();
```

Normalmente haremos ambas cosas en una sola línea:

```
List<Integer> lista = new ArrayList<Integer>();
```

Habréis visto que en lugar de **int** estamos escribiendo **Integer** en el tipo de datos que metemos en la lista. Esto es porque en Java no se pueden crear listas de los tipos primitivos (int, double, etc.). En lugar de éstos, habrá que usar otros tipos prácticamente equivalentes (pero que nos darán algún que otro problema al convertir de uno al otro) que sí podremos meter dentro de nuestras listas.

A continuación, tenéis una tabla con los tipos que tendremos que usar:

Tipo primitivo	Tipo que tenemos que usar en una lista
int	Integer
long	Long
double	Double
char	Character
boolean	Boolean

Las **listas** se usan de manera similar a los **arrays**, con unas cuantas diferencias:

- Las **listas** no tienen un tamaño definido, por lo que a la hora de crearlas no hay que especificar el tamaño.

  Para añadir un nuevo dato a la **lista**, usaremos la función **add()**, que nos añade un elemento en la última posición. Así, iremos añadiendo todos los valores que necesitemos.

  ```
  lista.Add(25);
  ```

  En un **array**, tendríamos que crear un array más grande, copiar el contenido y poner el nuevo valor al final.

- En Java no podemos acceder a los elementos de una lista usando el operador []. Para acceder a cada elemento, usaremos **get(posición)**.

  ```
  int valor = lista.get(0);
  ```

  Y para modificar cada elemento, deberemos usar la función **set(posición, elemento)**. Por ejemplo, para poner el valor 100 en la posición 2 (machacando lo que haya):

  ```
  lista.set(2, 100);
  ```

- Para saber cuántos elementos hay en una **lista**, usaremos la función **size()**. En un **array** usábamos **length()** que nos devolvía el tamaño que habíamos reservado.

  ```
  System.out.println(lista.size());
  ```

Salvo estas diferencias, una **lista** se puede usar básicamente de la misma forma que se usa un **array**. No obstante, las **listas** disponen de funciones específicas que nos permiten realizar las operaciones más comunes. A continuación, se enumeran las más importantes:

- **void add(<tipo dato> valor)**: Añade un valor a la lista en la última posición. El valor tiene que ser del mismo tipo del que hayamos definido la lista (esto es, en una lista de enteros sólo podemos almacenar enteros).

- **void add(int indice, <tipo dato> valor)**: Inserta un valor en la lista en la posición especificada. La posición **0** se corresponde con el inicio de la lista (se introduciría el elemento al principio), mientras que la posición **lista.size()** se corresponde a la última posición.

- **int indexOf(<tipo dato> valor)**: Nos devuelve la posición del elemento, si existe. Si el elemento no existe, nos devolverá -1.

- **bool contains(<tipo dato> valor)**: Nos devuelve **true** si el valor existe dentro de la lista y **false** en caso contrario.

- **void clear()**: Elimina todos los elementos de la lista y nos la deja preparada para volver a introducir datos.

- **void remove(int posicion)**: Elimina el elemento que se encuentra en la posición especificada. Recordemos que la primera posición es la **0** y la última será la **size()-1**.

- **bool remove(<tipo dato> valor)**: Elimina la primera vez que aparece el valor dentro de la lista. Nos devolverá **true** si ha encontrado y borrado el valor y **false** si el valor no se ha encontrado dentro de la lista.

  Si lo que queremos borrar es un entero, tendremos el problema de que Java usará la función anterior en lugar de ésta (porque las dos se llaman igual). Para obligarle a usar esta función, tendremos que usar el siguiente truco:

  ```
  lista.remove(Integer.valueOf(2));
  ```

- **void addAll(<colección> colección)**: Copia dentro de la lista todos los elementos de la colección. Esta colección puede ser otra **lista**, un **array**, etc. En el caso de que queramos meter un array en una lista podemos tener el problema de que el array sea de *int* y la lista de *Integer* (que es lo mismo pero que no es lo mismo). En ese caso, no podremos usar esta función y tendremos que meter los valores uno a uno con un bucle.

- Otras funciones interesantes las encontramos dentro de la clase **Collections**. Las dos más interesantes nos permiten ordenar (**sort**) y darle la vuelta (**reverse**) a una lista.

  ```
  Collections.sort(lista);
  Collections.reverse(lista);
  ```

Ficheros

Los ficheros nos permiten guardar y recuperar información desde los dispositivos de almacenamiento. Básicamente, hay dos tipos de ficheros: **ficheros de texto** y **ficheros binarios**. Ambos funcionan de manera similar, pero en los ficheros de texto sólo podemos guardar cadenas de caracteres mientras que en los ficheros binarios podemos guardar todo tipo de datos.

El proceso para utilizar un archivo consta de tres pasos: **apertura del fichero**, **lectura y escritura de datos**, y **cerrado del fichero**. En Java, utilizaremos diferentes instrucciones para los ficheros de texto y los ficheros binarios, y también diferentes opciones según queramos leer o escribir en el fichero.

Excepciones

Antes de explicar los ficheros en Java, necesitamos hablar de las **excepciones**. Una excepción se produce cuando hay algún tipo de error a la hora de ejecutar el programa (por ejemplo, cuando nos salimos de los límites de un array). En el caso de los ficheros, se pueden producir muchos tipos de excepciones (que el fichero no exista, que no se pueda abrir, que no tenga los datos que creíamos que tenía…) por lo que Java nos obliga a gestionar las excepciones que se puedan producir.

Esto se hace mediante la estructura **try-catch**. Básicamente, todo lo que metamos dentro de un **try** estará protegido si se producen excepciones. Si se produce alguna excepción, se ejecutará el código que hay en el **catch** para mostrar o corregir el error.

```
try
{
    // ... código ...
}
catch(Exception e)
{
    System.out.println("ERROR: " + e.getMessage();
}
```

En el ejemplo, pondremos una instrucción en el **catch** que nos mostrará el error por pantalla.

Se desaconseja usar la estructura **try-catch** cuando no sea imprescindible. Esconder los mensajes de error cuando algo no funciona no es la solución más adecuada. Sólo se debe usar un **try-catch** en los casos en los que sea imposible evitar que se produzcan excepciones (porque no dependan de nuestro código).

Ficheros de texto

Los **ficheros de texto** contienen única y exclusivamente cadenas de texto, organizadas en líneas.

Escritura en ficheros de texto

Para escribir texto en un fichero, primero abriremos el fichero en **modo escritura**:

```
FileWriter fw = new FileWriter(<nombreFichero>);
```

Una vez tengamos el fichero abierto, usaremos un *BufferedWriter*, que nos permitirá escribir texto en el fichero.

```
BufferedWriter bw = new BufferedWriter(fw);
```

Una vez abierto, podremos escribir texto en él. La función a utilizar será *write*. Esta función sólo nos permitirá escribir cadenas. Si queremos escribir cualquier otro valor siempre podremos convertirlo a cadena con *String.valueOf*.

```
int n = 90;
bw.write(String.valueOf(n));
```

Si queremos escribir en diferentes líneas usaremos '**\n**' para para saltar a la línea siguiente.

```
bw.write("Hola\n");
```

Cuando hayamos terminado de escribir todo el fichero, deberemos cerrarlo para que se guarden completamente los datos en el disco. Deberemos cerrar tanto el *BufferedWriter* como el *FileWriter*.

```
bw.close();
fw.close();
```

Lectura en ficheros de texto

Para leer texto desde un fichero, primero abriremos el fichero en **modo lectura**:

```
FileReader fr = new FileReader(<nombreFichero>);
```

Y luego abriremos el *BufferedReader*:

```
BufferedReader br = new BufferedReader(fr);
```

Para leer datos del fichero, normalmente lo haremos por líneas mediante la función *readLine*.

```
String s = br.readLine();
```

En principio sólo podemos leer cadenas, pero si contienen otro tipo de datos siempre podemos convertirlo después de leerlo

```
int n = Integer.valueOf(s);
```

Si conocemos el número de líneas que contiene el fichero, podemos hacer un bucle **for** para irlas leyendo, pero si no lo conocemos, tendremos que usar un método especial que nos controla si hemos llegado al final del fichero. Aquí tenéis un ejemplo de cómo se leería:

```
while((s = br.readLine()) != null)
{
    System.out.println(s);
}
```

Otra versión más fácil de entender:

```
s = br.readLine();
while(s != null)
{
    System.out.println(s);
    s = br.readLine();
}
```

No se os olvide cerrar el fichero después de haberlo utilizado.

```
br.close();
fr.close();
```

Ficheros binarios

Aunque en un **fichero de texto** se pueden guardar datos como números enteros, números reales, etc. (se guardan como texto, se leen con *readLine* y se convierten con al tipo de datoscorrespondiente), es mucho mejor usar para ello los **ficheros binarios**.

Escritura en ficheros binarios

Para escribir en un fichero binario, usaremos dos cosas: un *FileStream* para abrir el fichero y un *DataStream* para poder leer y escribir diferentes tipos de datos:

```
FileOutputStream fos = new FileOutputStream(nombreFichero);
DataOutputStream dos = new DataOutputStream(fos);
```

Una vez abierto, podremos escribir datos en él. En un DataOutputStream tenemos diferentes funciones para los diferentes tipos de datos que tenemos que escribir:

```
dos.writeInt(7);
dos.writeDouble(2.6);
dos.writeBoolean(true);
dos.writeChar('a');
dos.writeUTF("cadena");
```

Cuando hayamos terminado de escribir todo el fichero, deberemos cerrarlo para que se guarden completamente los datos en el disco.

```
dos.close();
fos.close();
```

Lectura en ficheros binarios

Para leer datos desde un fichero binario escribiremos:

```
FileInputStream fis = new FileInputStream(nombreFichero);
DataInputStream dis = new DataInputStream(fis);
```

Para leer datos desde un fichero binario también tendremos diferentes funciones dependiendo del tipo de datos que queramos leer. Deberemos leer el tipo de datos que se escribió originalmente en el fichero (o sea, si es un fichero de *double*, podremos leer *doubles* pero no *Strings*).

```
int n = dis.readInt();
double d = dis.readDouble();
boolean b = dis.readBoolean();
char c = dis.readChar();
String s = dis.readUTF();
```

Si conocemos el número de datos que contiene el fichero, podemos hacer un bucle **for** para irlos leyendo, pero si no lo conocemos, tendremos que usar una función especial que nos avisa cuando hemos llegado al final del fichero:

```
while(dis.available() > 0)
{

}
```

Como de costumbre, no os olvidéis de cerrar el fichero.

```
dis.close();
fis.close();
```

Funciones de Ficheros y Directorios

Cuando trabajamos con ficheros, muchas veces necesitamos, además de leer o escribir datos en el mismo, obtener información sobre el fichero. También nos suele hacer falta obtener el contenido de un directorio. Para ello, Java cuenta con la clase **File** que contiene varias funciones para trabajar con ficheros y directorios.

Comprobar si existe un fichero

Para comprobar si existe o no un fichero, primero tendremos que crear un objeto de la clase **File** usando la ruta donde creemos que se encuentra el fichero.

```
File fichero = new File("prueba.txt");
```

Una vez creado este objeto "fichero" tenemos varias funciones que nos permiten obtener información sobre él. Para ver si un fichero existe usaremos:

```
if(fichero.exists())
```

Sin embargo, puede que un fichero exista pero no sea un fichero realmente porque Java llama ficheros tanto a ficheros como directorios (y otras cosas más raras que nos encontramos en los sistemas de archivos). Afortunadamente, la clase File tiene también funciones para ver si nuestro objeto es un fichero o un directorio:

```
if(fichero.isFile())
if(fichero.isDirectory())
```

Así que para comprobar si un fichero existe y es un fichero realmente, pondremos:

```
if(fichero.exists() && fichero.isFile())
```

Una manera alternativa que existe en versiones de Java más modernas (desde 2011, pero a la gente le cuesta actualizarse) es usar la clase **Files** (no confundir con **File**). La sintaxis es similar:

```
if(Files.exists(Path.of("prueba.txt")))
if(Files.isRegularFile(Path.of("prueba.txt")));
if(Files.isDirectory(Path.of("prueba.txt")));
```

Realizar operaciones con ficheros

Una vez que disponemos de un objeto File, podemos realizar varias operaciones con ficheros. Las más normales son renombrar un fichero o borrarlo.

```
fichero.delete();
fichero.renameTo(new File("otronombre.txt"));
```

Sin embargo, la "nueva" clase **Files** nos da más opciones, lo que nos permite además copiar o mover un fichero:

```
Files.delete(Path.of("prueba.txt"));
Files.move(Path.of("prueba.txt"), Path.of("otronombre.txt"));
Files.copy(Path.of("prueba.txt"), Path.of("prueba2.txt"));
```

Obtener el listado de un directorio

Para obtener los contenidos de un directorio también usaremos **File** ya que es la misma clase para ficheros y directorios.

```
File directorio = new File(".");
File[] ficheros = directorio.listFiles();
```

Con esto obtendremos un array con todos los ficheros del directorio actual (donde se ha ejecutado el programa). Si queremos los de cualquier otra carpeta, se la pasamos por parámetro:

```
File directorio = new File("C:\\Windows");
```

Este array tendremos que irlo recorriendo con un **for** y obteniendo la información que necesitemos de los ficheros. Hay muchas funciones para ello, como **getName()** que nos da el nombre del fichero, **getPath()** que nos da la ruta completa, **length()** que nos da el tamaño, etc.
Recordad que para Java tanto los ficheros como los directorios son **File**, así que si habrá que comprobar si lo que hay dentro es un fichero o un directorio. Ejemplo:

```
for(int i = 0; i < ficheros.length; i++)
{
    if(ficheros[i].isFile())
    {
        System.out.println(ficheros[i].getName());
    }
}
```

Programación Orientada a Objetos

Java es un lenguaje imperativo que contiene muchos elementos de los lenguajes orientados a objetos. Un objeto es un agregado de datos y de métodos que permiten manipular dichos datos.

Clases y Objetos

Una **clase** es la definición de las características concretas de un determinado tipo de objetos. Es decir, de cuáles son los datos y los métodos de los que van a disponer todos los objetos de ese tipo. Por esta razón, se suele decir que el **tipo de dato** de un objeto es la clase que define las características del mismo.

Un **objeto**, por su parte, es el resultado de instanciar una clase. En este sentido, es similar al concepto de **variable**.

Las **clases** y **objetos** se componen de diferentes **miembros**, que podrán ser:

- **Atributos**: se corresponden con los datos que guarda un objeto en un determinado momento. Son similares a las **variables,** pero se encuentran dentro de un objeto.

Un atributo puede ser de cualquier tipo de datos, incluyendo otros objetos (incluso de la misma clase).

- **Métodos**: son un conjunto de instrucciones a las que se les asocia un nombre de modo que si se desea ejecutarlas basta referenciarlas a través de dicho nombre. Son equivalentes a las **funciones,** pero dentro de un objeto. Dentro de estos métodos es posible acceder con total libertad a la información almacenada en los atributos pertenecientes a la clase dentro de la que el método se ha definido.
- **Constructores**: son un tipo de métodos especial que nos permiten inicializar cada objeto.
- **Propiedades**: son similares a los **métodos** en el sentido de que contienen instrucciones que se van a ejecutar, pero a la hora de usarlas en un programa se comportan de manera parecida a los **atributos**, es decir, podemos leer datos y escribir datos de ellas como si fueran variables. Sirven para encapsular la información contenida en los atributos del objeto.

  En Java no existen las propiedades como tales (al contrario que otros lenguajes), pero solemos llamar propiedad al conjunto de dos funciones que nos permiten acceder a un atributo:
  - La función que nos permite leer el valor de un atributo se llama *get+nombre del atributo* (ej.: getNombre). No recibirá parámetros y nos devolverá un dato del mismo tipo que el atributo.
  - La función que nos permite escribir un valor en un atributo se llama *set+nombre del atributo* (ej: setNombre). No devuelve nada (es de tipo *void*) y le pasamos un parámetro que será el valor que queremos almacenar en el atributo.

Conceptos fundamentales

La programación orientada a objetos introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- **Encapsulamiento**: Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema.
- **Principio de ocultación**: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.
- **Polimorfismo**: básicamente, consiste en que pueden existir dos o más métodos con el mismo nombre dentro de una clase siempre y cuando tengan:
  a) Diferente número de parámetros.
  b) El mismo número de parámetros, pero de pertenecientes a diferentes tipos de datos.

  A la hora de ejecutar el método, el compilador elegirá el más apropiado de entre los que tienen el mismo nombre.

- **Herencia**: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.

Control de Acceso a métodos y atributos

Para proteger el acceso a los diferentes atributos y también a los métodos, se definen varios niveles de acceso que se asignaran a dichos atributos y métodos. Los niveles de acceso son los siguientes:

- **public**: el acceso no está restringido. Todo el mundo puede acceder a los métodos y atributos con total libertad.
- **private**: el acceso está restringido a la propia clase donde se define el método o atributo, es decir, sólo se podrá acceder a un atributo o a un método desde los propios métodos de la misma clase.
- **protected**: el acceso está limitado a la propia clase y a las clases que heredan de ésta. Se usa combinado con la herencia.

Definición de una Clase en Java

La sintaxis básica para definir una clase es la que a continuación se muestra:

```
public class <nombreClase>
{
      <miembros>
}
```

Dentro de la clase, definiremos los diferentes atributos, métodos y propiedades. Para definir un atributo, usaremos la sintaxis:

```
<nivel_de_acceso> <tipo_de_dato> <nombre_del_atributo>;
```

Por ejemplo, si queremos definir un atributo privado de tipo entero y otro público de tipo cadena, escribiremos:

```
private int entero;
public String cadena;
```

Si no especificamos el nivel de acceso, en Java se sobreentiende que el atributo o método será público siempre y cuando estemos dentro del mismo package.
Para definir un método:

```
<nivel_de_acceso> <tipoDevuelto> <nombreMétodo> (<parametros>)
{
      <instrucciones>
}
```

Por ejemplo:

```
public void insertaValor(int dato)
{
    ...
}
```

Para acceder a un atributo o a un método de un objeto, usaremos el operador ".":

```
objeto.cadena = "hola";
objeto.insertaValor(3);
```

Constructores

Los constructores inicializan el estado de un objeto a la hora de crearlo. Se definen de manera similar a los métodos, sólo que no devuelven ningún valor (en teoría devuelven un objeto de la misma clase, pero no hace falta indicarlo ni devolverlo con un *return*) y tienen que tener el mismo nombre que la clase. Así, para definir un constructor para la clase "prueba", escribiremos:

```
public Prueba(int dato)
{
    ...
}
```

Gracias al polimorfismo, podemos definir varios constructores con diferentes parámetros. Para crear un nuevo objeto, deberemos llamar al constructor con el comando especial **new**.

```
Prueba p = new Prueba(6);
```

Los constructores se utilizan, generalmente, para darle valores iniciales a los atributos del objeto. Al menos se debe crear un constructor por defecto (sin parámetros).

Propiedades

Las propiedades nos permiten acceder de manera controlada a los atributos de una clase. En Java no existen las propiedades como tales, así que se simulan usando una pareja de métodos *getAtributo* y *setAtributo*.
Aquí tenéis un ejemplo de cómo serían las propiedades básicas correspondientes al atributo edad.:

```
public class Alumno
{
    private int edad;

    public int getEdad()
    {
        return edad;
    }

    public void setEdad(int edad)
    {
        this.edad = edad;
```

```
        }
}
```