Preparador Informática

www.preparadorinformatica.com

MANUAL 1
PROGRAMACIÓN EN JAVA
(Elementos Básicos)

1. Introducción

Java surgió en 1991 cuando un grupo de ingenieros de Sun Microsystems trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Desarrollaron un código "neutro" que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una **"máquina hipotética o virtual" denominada Java Virtual Machine (JVM)**. Era la JVM quien interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadores, Java se introdujo a finales de 1995. La clave fue la incorporación de un intérprete Java en la versión 2.0 del programa Netscape Navigator, produciendo una verdadera revolución en Internet.

Al programar en Java no se parte de cero. Cualquier aplicación que se desarrolle se apoya en un gran número de clases preexistentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje **(el API o Application Programming Interface de Java).** Java incorpora en el propio lenguaje muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de software o fabricantes de ordenadores (threads, ejecución remota, componentes, seguridad, acceso a bases de datos, etc.).

El principal objetivo del lenguaje Java es llegar a ser el "nexo universal" que conecte a los usuarios con la información, esté ésta situada en el ordenador local, en un servidor de Web, en una base de datos o en cualquier otro lugar.

Java es un lenguaje muy completo. En cierta forma casi todo depende de casi todo. Por ello, conviene aprenderlo de modo iterativo: primero una visión muy general, que se va refinando en sucesivas iteraciones.

La compañía Sun describe el lenguaje Java como **"simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico".** Todo ello describe bastante bien el lenguaje Java. Para entender mejor algunas de estas características de Java y la programación orientada a objetos os recomiendo que miréis el tema 27 de Informática que equivale al 29 de SAI denominado "PROGRAMACIÓN ORIENTADA A OBJETOS. OBJETOS. CLASES. HERENCIA. POLIMORFISMO."

2. Motivos para el diseño de Java

Los lenguajes de programación C y Fortran se han utilizado para diseñar algunos de los sistemas más complejos en lenguajes de programación estructurada, creciendo hasta formar complicados procedimientos. De ahí provienen términos como "código de espagueti" o "canguros" referentes a programas con múltiples saltos y un control de flujo difícilmente trazable.

No sólo se necesitaba un lenguaje de programación para tratar esta complejidad, sino un nuevo estilo de programación. Este cambio de paradigma de la programación estructurada a la programación orientada a objetos comenzó con un lenguaje llamado Simula67.

El lenguaje C++ fue un intento de tomar estos principios y emplearlos dentro de las restricciones de C. Todos los compiladores de C++ eran capaces de compilar programas de C sin clases, es decir, un lenguaje capaz de interpretar dos estilos diferentes de programación. Esta compatibilidad que habitualmente se vende como una característica de C++ es precisamente su punto más débil. No es necesario utilizar un diseño orientado a objetos para programar en C++, razón por la que muchas veces las aplicaciones en este lenguaje no son realmente orientadas al objeto, perdiendo así los beneficios que este paradigma aporta.

Así Java utiliza convenciones casi idénticas para declaración de variables, paso de parámetros, y demás, pero sólo considera las partes de C++ que no estaban ya en C. **Las principales características que Java no hereda de C++ son:**

- **Punteros**: Las direcciones de memoria son la característica más poderosa de C++. El inadecuado uso de los punteros provoca la mayoría de los errores de colisión de memoria, errores muy difíciles de detectar. Además, casi todos los virus que se han escrito aprovechan la capacidad de un programa para acceder a la memoria volátil (RAM) utilizando punteros. **En Java, no existen punteros, evitando el acceso directo a la memoria volátil.**
- **Variables globales**: Con ellas cualquier función puede producir efectos laterales, e incluso se pueden producir fallos catastróficos cuando algún otro método cambia el estado de la variable global necesaria para la realización de otros procesos. **En Java lo único global es el nombre de las clases.**
- **goto**: Manera rápida de arreglar un programa sin estructurar el código. **Java no tiene ninguna sentencia goto. Sin embargo, Java tiene las sentencias break y continue que cubren los casos importantes de goto.**
- **Asignación de memoria**: La función malloc de C, asigna un número especificado de bytes de memoria devolviendo la dirección de ese bloque. La función free devuelve un bloque asignado al sistema para que lo utilice. Si se olvida de llamar a free para liberar un bloque de memoria, se están limitando los recursos del sistema, ralentizando progresivamente los programas. Si por el contrario se hace un free sobre un puntero ya liberado, puede ocurrir cualquier cosa. Más tarde C++ añadió new y delete, que se usan de forma similar, siendo todavía el programador, el responsable de liberar el espacio de memoria. **Java no tiene funciones malloc ni free.** Se utiliza el operador new para asignar un espacio de memoria a un objeto en el montículo de memoria. Con new no se obtiene una dirección de memoria sino un descriptor al objeto del montículo. La memoria real asignada a ese objeto se puede mover a la vez que el programa se ejecuta, pero sin tener que preocuparse de ello. **Cuando no tenga ninguna referencia de ningún objeto, la memoria ocupada estará disponible para que la reutilice el resto del sistema sin tener que llamar a free o delete. A esto se le llama recogida de basura.** El recolector de basura se ejecuta siempre que el sistema esté libre, o cuando una asignación solicitada no encuentre asignación suficiente.
- **Conversión de tipos insegura**: Los moldeados de tipo (type casting) son un mecanismo poderoso de C y C++ que permite cambiar el tipo de un puntero. Esto requiere extremada precaución puesto que no hay nada previsto para detectar si la conversión

es correcta en tiempo de ejecución. **En Java se puede hacer una comprobación en tiempo de ejecución de la compatibilidad de tipos y emitir una excepción cuando falla.**

Considerando todo lo anterior podemos decir que los puntos fuertes de Java y los que le hicieron resurgir son, entre otros, los siguientes:

- Java es un **lenguaje orientado a objetos**: Esto es lo que facilita abordar la resolución
- de cualquier tipo de problema.
- Es un lenguaje **sencillo**, aunque sin duda **potente**.
- La ejecución del código Java es **segura y fiable**: Los programas no acceden directamente a la memoria del ordenador, siendo imposible que un programa escrito en Java pueda acceder a los recursos del ordenador sin que esta operación le sea permitida de forma explícita. De este modo, los datos del usuario quedan a salvo de la existencia de virus escritos en Java. La ejecución segura y controlada del código Java es una característica única, que no puede encontrarse en ninguna otra tecnología.
- Es totalmente **multiplataforma**: Es un lenguaje sencillo, por lo que el entorno necesario para su ejecución es de pequeño tamaño y puede adaptarse incluso al interior de un navegador.

3. El entorno de desarrollo de Java

Existen distintos programas comerciales que permiten desarrollar código Java. La compañía Sun, creadora de Java, distribuye gratuitamente el Java(tm) Development Kit (JDK). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en Java. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (con el denominado Debugger).

Existe también una versión reducida del JDK, denominada JRE (Java Runtime Environment) destinada únicamente a ejecutar código Java (no permite compilar).

Los IDEs (Integrated Development Environment), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código Java, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar Debug gráficamente, frente a la versión que incorpora el JDK basada en la utilización de una consola bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa. Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas, y ficheros resultantes de mayor tamaño que los basados en clases estándar. Para las prácticas yo os recomendaría que lo hicierais en un entorno de desarrollo (por ejemplo, yo voy a utilizar Eclipse como IDE) pero siempre luego debemos aprender a compilar y ejecutar en nuestro terminal por si fuera necesario. (Tanto para utilizar el IDE correspondiente como para realizar la compilación y ejecución por el terminal debemos tener instalado en nuestro ordenador el JDK correspondiente).

3.1. Compilar y Ejecutar en Java

Se trata de una de las herramientas de desarrollo incluidas en el JDK. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de Java (con extensión *.java). Si no encuentra

errores en el código genera los ficheros compilados (con extensión *.class). En otro caso muestra la línea o líneas erróneas. En el JDK de Sun dicho compilador se llama javac.exe.

Por tanto, cuando queremos compilar desde línea de comandos un archivo .java usamos:

```
javac nombrearchivo.java
```

Y en este momento si no hay errores se creará el archivo con el mismo nombre pero con extensión .class. Para ejecutarlo bastaría con utilizar el comando java.exe de la siguiente manera: (Muy importante, no se coloca la extensión del archivo (es decir el .class))

```
java nombrearchivo
```

4. Características de Java

Veamos en este apartado más detalladamente las características de Java. Tiene tres elementos claves que diferencian a este lenguaje desde un punto de vista tecnológico:

- Es un lenguaje de programación que ofrece la potencia del diseño orientado a objetos con una sintaxis fácilmente accesible y un entorno robusto y agradable.
- Proporciona un conjunto de clases potente y flexible.
- Pone al alcance de cualquiera la utilización de aplicaciones que se pueden incluir directamente en páginas Web.

A. Lenguaje Potente:

i. **Orientación a objetos:** En este aspecto Java fue diseñado partiendo de cero, no siendo derivado de otro lenguaje anterior y no tiene compatibilidad con ninguno de ellos. En Java el concepto de objeto resulta sencillo y fácil de ampliar. Además, se conservan elementos "no objetos", como números, caracteres y otros tipos de datos simples.

ii. **Riqueza semántica:** Pese a su simpleza se ha conseguido un considerable potencial, y aunque cada tarea se puede realizar de un número reducido de formas, se ha conseguido un gran potencial de expresión e innovación desde el punto de vista del programador.

iii. **Robusto:** Java verifica su código al mismo tiempo que lo escribe, y una vez más antes de ejecutarse, de manera que se consigue un alto margen de codificación sin errores. Se realiza un descubrimiento de la mayor parte de los errores durante el tiempo de compilación, ya que Java es estricto en cuanto a tipos y declaraciones, y así lo que es rigidez y falta de flexibilidad se convierte en eficacia. Respecto a la gestión de memoria, Java libera al programador del compromiso de tener que controlar especialmente la asignación que de ésta hace a sus necesidades específicas. Este lenguaje posee una gestión avanzada de memoria llamada gestión de basura, y un manejo de excepciones orientado a objetos integrados. Estos elementos realizarán muchas tareas antes tediosas a la vez que obligadas para el programador.

iv. **Modelo de objeto rico:** Existen varias clases que contienen las abstracciones básicas para facilitar a los programas una gran capacidad de representación. Para ello se contará con un conjunto de clases comunes que pueden crecer para admitir todas las necesidades del programador.

B. Lenguaje Simple:

i. **Fácil aprendizaje:** El único requerimiento para aprender Java es tener una comprensión de los conceptos básicos de la programación orientada a objetos. Así se ha creado un lenguaje simple (aunque eficaz y expresivo) pudiendo mostrarse

cualquier planteamiento por parte del programador sin que las interioridades del sistema subyacente sean desveladas.

ii. **Completado con utilidades:** El paquete de utilidades de Java viene con un conjunto completo de estructuras de datos complejas y sus métodos asociados, que serán de inestimable ayuda para implementar aplicaciones más complejas. Se dispone también de estructuras de datos habituales, como por ejemplos pilas, como clases ya implementadas.

C. Interactivo y Orientado a Red:

i. **Interactivo y animado:** Uno de los requisitos de Java desde sus inicios fue la posibilidad de crear programas en red interactivos, por lo que es capaz de hacer varias cosas a la vez sin perder rastro de lo que debería suceder y cuándo. Para se da soporte a la utilización de múltiples hilos de programación (multithread). Además, las aplicaciones de Java permiten situar figuras animadas en las páginas Web, y éstas pueden concebirse con logotipos animados o con texto que se desplace por la pantalla.

ii. **Arquitectura neutral:** Java está diseñado para que un programa escrito en este lenguaje sea ejecutado correctamente independientemente de la plataforma en la que se esté actuando (Macintosh, PC, UNIX…).

iii. **Trabajo en red:** Java anima las páginas Web y hace posible la incorporación de aplicaciones interactivas y especializadas. Los protocolos básicos para trabajar en Internet están encapsulados en unas cuantas clases simples. Se incluyen implementaciones ampliables de los protocolos FTP, HTTP, NNTP y SMTP junto con conectores de red de bajo nivel e interfaces de nombrado. Este lenguaje está diseñado para cumplir los requisitos de entrega de contenidos interactivos mediante el uso de applets insertados en sus páginas HTML. También, Java proporciona un conjunto de clases para tratar con una abstracción de los conectores de red (sockets originales de la versión UNIX de Berckley, encapsular la noción de una dirección de Internet o conectar sockets con flujos de datos de Entrada/Salida.

D. Seguridad: Existe una preocupación lógica en Internet por el tema de la seguridad: virus, caballos de Troya, y programas similares navegan de forma usual por la red, constituyendo una amenaza palpable. Java ha sido diseñado poniendo un énfasis especial en el tema de la seguridad, y se ha conseguido lograr cierta inmunidad en el aspecto de que un programa realizado en Java no puede realizar llamadas a funciones globales ni acceder a recursos arbitrarios del sistema, por lo que el control sobre los programas ejecutables no es equiparable a otros lenguajes.

E. Gestión de la Entrada/Salida: En lugar de utilizar primitivas como las de C para trabajar con ficheros, se utilizan primitivas mucho más elegantes, que permiten tratar los ficheros, sockets, teclado y monitor como flujos de datos. De este modo se pueden utilizar dichas primitivas para cualquier operación de Entrada/Salida.

F. **Diferentes tipos de aplicaciones:** En Java podemos crear los siguientes tipos de aplicaciones entre otras:

i. Aplicaciones: Se ejecutan sin necesidad de un navegador.

ii. Applets: Se pueden descargar de Internet y se observan en un navegador.

iii. JavaBeans: Componentes software Java, que se puedan incorporar gráficamente a otros componentes.

iv. JavaScript: Conjunto del lenguaje Java que puede codificarse directamente sobre cualquier documento HTML

v. Servlets: Módulos que permiten sustituir o utilizar el lenguaje Java en lugar de programas CGI (Common Gateway Interface) a la hora de dotar de interactividad a las páginas Web.

5. Nomenclatura habitual en la programación en Java

Los nombres de Java son sensibles a las letras mayúsculas y minúsculas. Así, las variables masa, Masa y MASA son consideradas variables completamente diferentes. Las reglas del lenguaje respecto a los nombres de variables son muy amplias y permiten mucha libertad al programador, pero es habitual seguir ciertas normas que facilitan la lectura y el mantenimiento de los programas de ordenador. Se recomienda seguir las siguientes instrucciones:

1. En Java es habitual utilizar nombres con minúsculas, con las excepciones que se indican en los puntos siguientes.
2. Cuando un nombre consta de varias palabras es habitual poner una a continuación de otra, poniendo con mayúscula la primera letra de la palabra que sigue a otra (Ejemplos: elMayor(), ventanaCerrable, rectanguloGrafico, addWindowListener()).
3. Los nombres de clases e interfaces comienzan siempre por mayúscula (Ejemplos: Geometria, Rectangulo, Dibujable, Graphics, ArrayList, Iterator).
4. Los nombres de objetos, los nombres de métodos y variables miembro, y los nombres de las variables locales de los métodos, comienzan siempre por minúscula (Ejemplos: main(), dibujar(), numRectangulos, x, y, r).
5. Los nombres de las variables finales, es decir de las constantes, se definen siempre con mayúsculas (Ejemplo: PI)

6. Estructura general de un programa Java

Normalmente aparece una clase que contiene el programa principal (aquel que contiene la función main()) y algunas clases de usuario (las específicas de la aplicación que se está desarrollando) que son utilizadas por el programa principal. Los ficheros fuente tienen la extensión *.java, mientras que los ficheros compilados tienen la extensión *.class.

Un fichero fuente (*.java) puede contener más de una clase, pero sólo una puede ser public. El nombre del fichero fuente debe coincidir con el de la clase public (con la extensión *.java). Si por ejemplo en un fichero aparece la declaración (public class MiClase {...}) entonces el nombre del fichero deberá ser MiClase.java. Es importante que coincidan mayúsculas y minúsculas ya que MiClase.java y miclase.java serían clases diferentes para Java.

Es habitual que una aplicación está constituida por varios ficheros *.class. Cada clase realiza unas funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. La aplicación se ejecuta por medio del nombre de la clase que contiene la función main() (sin la extensión *.class). Las clases de Java se agrupan en packages, que son librerías de clases.

6.1. Concepto de Clase

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos. A estos datos y funciones pertenecientes a una clase se les denomina variables y métodos o funciones miembro. La programación orientada a objetos se basa en la programación de clases. Un programa se construye a partir de un conjunto de clases.

Una vez definida e implementada una clase, es posible declarar elementos de esta clase de modo similar a como se declaran las variables del lenguaje (de los tipos primitivos int, double, String, …). Los elementos declarados de una clase se denominan objetos de la clase. De una única clase se pueden declarar o crear numerosos objetos. La clase es lo genérico: es el patrón o modelo para crear objetos. Cada objeto tiene sus propias copias de las variables miembro, con

sus propios valores, en general distintos de los demás objetos de la clase. Las clases pueden tener variables static, que son propias de la clase y no de cada objeto.

6.2. Herencia

La herencia permite que se pueden definir nuevas clases basadas en clases existentes, lo cual facilita re-utilizar código previamente desarrollado. Si una clase deriva de otra (**extends**) hereda todas sus variables y métodos. La clase derivada puede añadir nuevas variables y métodos y/o redefinir las variables y métodos heredados.

En Java, a diferencia de otros lenguajes orientados a objetos, una clase sólo puede derivar de una única clase, con lo cual no es posible realizar herencia múltiple en base a clases. Sin embargo, es posible "simular" la herencia múltiple en base a las interfaces.

6.3. Concepto de Interface

Una interface es un conjunto de declaraciones de funciones. Si una clase implementa (**implements**) una interface, debe definir todas las funciones especificadas por la interface. Una clase puede implementar más de una interface, representando una forma alternativa de la herencia múltiple. A su vez, una interface puede derivar de otra o incluso de varias interfaces, en cuyo caso incorpora todos los métodos de las interfaces de las que deriva.

6.4. Concepto de Package

Un **package** es una agrupación de clases. Existen una serie de packages incluidos en el lenguaje. Además, el usuario puede crear sus propios packages. Lo habitual es juntar en packages las clases que estén relacionadas. Todas las clases que formen parte de un package deben estar en el mismo directorio.

Es importante distinguir entre lo que significa herencia y package. Un package es una agrupación arbitraria de clases, una forma de organizar las clases. La herencia sin embargo consiste en crear nuevas clases en base a otras ya existentes. Las clases incluidas en un package no derivan por lo general de una única clase.

6.5. La jerarquía de clases de Java (API)

Durante la generación de código en Java, es recomendable y casi necesario tener siempre a la vista la documentación on-line del API de Java. En dicha documentación es posible ver tanto la jerarquía de clases, es decir la relación de herencia entre clases, como la información de los distintos packages que componen las librerías base de Java. Lo podemos encontrar en:

https://docs.oracle.com/javase/7/docs/api/

7. Elementos de Java

Un token es el elemento más pequeño de un programa que es significativo para el compilador. Estos tokens definen la estructura de Java. Cuando se compila un programa Java, el compilador analiza el texto, reconoce y elimina los espacios en blanco y comentarios y extrae tokens individuales. Los tokens resultantes se compilan, traduciéndolos a código de byte Java, que es independiente del sistema e interpretable dentro de un entorno Java. Los códigos de byte se ajustan al sistema de máquina virtual Java, que abstrae las diferencias entre procesadores a un procesador virtual único.

7.1. Identificadores

Los identificadores son tokens que representan nombres asignables a variables, métodos y clases para identificarlos de forma única ante el compilador y darles nombres con sentido para el programador.

Todos los identificadores de Java diferencian entre mayúsculas y minúsculas y deben comenzar con una letra, un subrayado (_) o símbolo de dólar ($). Los caracteres posteriores del identificador pueden incluir las cifras del 0 al 9. Como nombres de identificadores no se pueden usar palabras claves de Java. Por ejemplo:

```
int alturaMedia;
```

7.2. Palabras clave

Las palabras claves son aquellos identificadores reservados por Java para un objetivo determinado y se usan sólo de la forma limitada y específica. Son las siguientes:

abstact	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short
static	super	switch	syncroniced	this
throw	throws	transient	true	try
var	void	volatile	while	

7.3. Variables

Una variable es un nombre que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

- **Variables de tipos primitivos:** Están definidas mediante un valor único que puede ser entero, de punto flotante, carácter o booleano. Java permite distinta precisión y distintos rangos de valores para estos tipos de variables (char, byte, short, int, long, float, double, boolean). Ejemplos de variables de tipos primitivos podrían ser: 123, 3456754, 3.1415, 12e-09, 'A', True, etc.
- **Variables referencia:** Las variables referencia son referencias o nombres de una información más compleja: arrays u objetos de una determinada clase.

Desde el punto de vista del papel o misión en el programa, las variables pueden ser:

- **Variables miembro de una clase:** Se definen en una clase, fuera de cualquier método; pueden ser tipos primitivos o referencias.

- **Variables locales:** Se definen dentro de un método o más en general dentro de cualquier bloque entre llaves {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también tipos primitivos o referencias.

Los nombres de variables en Java se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por Java como operadores o separadores (,.+-*/ etc.). Por supuesto no se podrían utilizar las palabras reservadas comentadas en el apartado anterior.

Si bien es cierto que en Java los nombres de las variables se acostumbra a nombrarlas con todas sus letras en minúsculas. Ejemplos:

```
int soles=3;
char salir='q';
```

7.4. Tipos de datos básicos

En Java los tipos de datos básicos los podemos resumir en la siguiente tabla:

Tipo de Datos	*Alcance o Rango*
int	de -2147483648 a 2147483647 (4bytes)
byte	de -128 a 127 (1Byete)
short	de -32768 a 32767 (2Bytes)
long	de -9223372036854775808 a 9223372036854775807 (8Bytes)
char	de '\u0000' a '\uffff', ambos incluidos que es lo mismo que de 0 a 65535, 1 letra
boolean	0 o 1 (1bit)
float	4Bytes, punto flotante
double	8Bytes, punto flotante
String	No es un tipo de datos básico, es un objeto básico, con propiedades y métodos, pero el lenguaje Java permite definir un nuevo objeto con el delimitador (“), por lo que podemos concatenar (unir) texto utilizando el operador (+) con los nombres de los objetos de tipo String y los trozos de texto delimitados con (“).

7.5. Conversión de tipos de datos

En Java es posible transformar el tipo de una variable u objeto en otro diferente al original con el que fue declarado. Este proceso se denomina "conversión", "moldeado" o "tipado". La conversión se lleva a cabo colocando el tipo destino entre paréntesis, a la izquierda del valor que queremos convertir de la forma siguiente:

```
char c = (char)System.in.read();
```

La función read devuelve un valor int, que se convierte en un char debido a la conversión (char), y el valor resultante se almacena en la variable de tipo carácter c.

El tamaño de los tipos que queremos convertir es muy importante. No todos los tipos se convertirán de forma segura. Por ejemplo, al convertir un long en un int, el compilador corta los 32 bits superiores del long (de 64 bits), de forma que encajen en los 32 bits del int, con lo que, si contienen información útil, esta se perderá. Por ello se establece la norma de que "en las conversiones el tipo destino siempre debe ser igual o mayor que el tipo fuente". La siguiente tabla muestra las posibles conversiones que podemos hacer sin perder información:

Tipo Origen	Tipo Destino
byte	*double, float, long, int, char, short*
short	*double, float, long, int*
char	*double, float, long, int*
int	*double, float, long*
long	*double, float*
float	*double*

7.6. Vectores y Matrices

Una matriz es una construcción que proporciona almacenaje a una lista de elementos del mismo tipo, ya sea simple o compuesto. Si la matriz tiene solo una dimensión, se la denomina vector o array.

En Java los vectores se declaran utilizando corchetes tras la declaración del tipo de datos que contendrá el vector. Por ejemplo, esta sería la declaración de un vector de números enteros (int):

```
int vectorNumeros[ ]; // Vector de números
```

Se observa la ausencia de un número que indique cuántos elementos componen el vector, debido a que Java no deja indicar el tamaño de un vector vacío cuando le declara. La asignación de memoria al vector se realiza de forma explícita en algún momento del programa.

Para ello o se utiliza el operador new:

```
int vectorNumeros = new int[ 5 ]; // Vector de 5 números
```

O se asigna una lista de elementos al vector:

```
int vectorIni = { 2, 5, 8}; // == int vectorIni[3]=new int[3];
```


Se puede observar que los corchetes son opcionales en este tipo de declaración de vector, tanto después del tipo de variable como después del identificador. Si se utiliza la forma de new se establecerá el valor 0 a cada uno de los elementos del vector.

7.7. Literales y constantes

Los literales son sintaxis para asignar valores a las variables. Cada variable es de un tipo de datos concreto, y dichos tipos de datos tienen sus propios literales.

Mediante determinados modificadores (static y final) podremos crear variables constantes, que no modifican su valor durante la ejecución de un programa. Las constantes pueden ser numéricas, booleanas, caracteres (Unicode) o cadenas (String). Las cadenas, que contienen múltiples caracteres, aún se consideran constantes, aunque están implementadas en Java como objetos.

Veamos un ejemplo de constante declarada por el usuario:

```
final static int ALTURA_MAXIMA = 200;
```

Se puede observar que utilizamos final static, para que la variable sea total y absolutamente invariable.

Observaciones:

Los literales enteros son básicos en la programación en Java y presentan tres formatos:

- Decimal: Los literales decimales aparecen como números ordinarios sin ninguna notación especial.
- Hexadecimal: Los hexadecimales (base 16) aparecen con un 0x ó 0X inicial, notación similar a la utilizada en C y C++.
- Octal: Los octales aparecen con un 0 inicial delante de los dígitos.

Por ejemplo, un literal entero para el número decimal 12 se representa en Java como 12 en decimal, como 0xC en hexadecimal, y como 014 en octal.

7.8. Operadores

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++. Estos operadores se describen brevemente en los apartados siguientes:

7.8.1. Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: suma (+), resta (-), multiplicación (*), división (/) y resto de la división (%).

7.8.2. Operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el operador igual (=). La forma general de las sentencias de asignación con este operador es:

```
variable = expression;
```

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones "acumulativas" sobre una variable. La tabla siguiente muestra estos operadores y su equivalencia con el uso del operador igual (=).

Operador	Utilización	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

7.8.3. Operadores unarios

Los operadores más (+) y menos (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en Java es el estándar de estos operadores.

7.8.4. Operador instanceof

El operador instanceof permite saber si un objeto pertenece o no a una determinada clase. Es un operador binario cuya forma general es:

```
objectName instanceof ClassName
```

y que devuelve true o false según el objeto pertenezca o no a la clase.

7.8.5. Operador condicional ?:

Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

```
booleanExpression ? res1 : res2
```

donde se evalúa booleanExpression y se devuelve res1 si el resultado es true y res2 si el resultado es false. Es el único operador ternario (tres argumentos) de Java. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo, las sentencias:

```
x=1 ;
y=10;
z = (x<y)?x+3:y+8;
```

asignarían a z el valor 4, es decir x+3.

7.8.6. Operadores incrementales

Java dispone del operador incremento (++) y decremento (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

1. Precediendo a la variable (por ejemplo: ++i). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
2. Siguiendo a la variable (por ejemplo: i++). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles for es una de las aplicaciones más frecuentes de estos operadores.

7.8.7. Operadores relacionales

Los operadores relacionales sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor boolean (true o false) según se cumpla o no la relación considerada. Estos operadores se utilizan con mucha frecuencia en las bifurcaciones y en los bucles. Los operadores relacionales son los siguientes:

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

7.8.8. Operadores lógicos

Los operadores lógicos se utilizan para construir expresiones lógicas, combinando valores lógicos (true y/o false) o los resultados de los operadores relacionales. Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser true y el primero es false, ya se sabe que la condición de que ambos sean true no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre. Los operadores lógicos son los siguientes:

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalúa op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

7.8.9. Operador de concatenación de cadenas de caracteres (+)

El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método println(). La variable numérica result es convertida automáticamente por Java en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

7.8.10. Operadores que actúan a nivel de bits

Java dispone también de un conjunto de operadores que actúan a nivel de bits:

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2 (positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits (1 si sólo uno de los operandos es 1)
~	~op2	Operador complemento (invierte el valor de cada bit)

7.8.11. Prioridad de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de x/y*z depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en una sentencia, de mayor a menor precedencia:

```
postfix operators       [] . (params) expr++ expr--
unary operators         ++expr --expr +expr -expr ~ !
creation or cast        new (type)expr
multiplicative          * / %
additive                + -
shift                   << >> >>>
relational              < > <= >= instanceof
equality                == !=
bitwise                 AND &
bitwise                 exclusive OR ^
bitwise                 inclusive OR |
logical                 AND &&
logical                 OR ||
conditional             ? :
assignment              = += -= *= /= %= &= ^= |= <<= >>= >>>=
```

En Java, todos los operadores binarios, excepto los operadores de asignación, se evalúan de izquierda a derecha. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

7.9. Separadores

Se usan para informar al compilador de Java de cómo están agrupadas las cosas en el código. Los separadores admitidos por Java son: { } , : ;

7.10. Comentarios y espacios en blanco

El compilador de Java reconoce y elimina los espacios en blanco, tabuladores, retornos de carro y comentarios durante el análisis del código fuente. Los comentarios se pueden presentar en tres formatos distintos:

Formato	Uso
/*comentario*/	Se ignoran todos los caracteres entre /* */. Proviene del C
//comentario	Se ignoran todos los caracteres detrás de // hasta el fin de línea. Proviene del C++
/**comentario*/	Lo mismo que /* */ pero se podrán utilizar para documentación automática.

Por ejemplo, la siguiente línea de código presenta un comentario:

```
int alturaMinima = 150; // No menos de 150 centímetros
```

7.11. Expresiones y sentencias

Los operadores, variables y las llamadas a métodos pueden ser combinadas en secuencias conocidas como expresiones. El comportamiento real de un programa Java se logra a través de expresiones, que se agrupan para crear sentencias.

Una expresión es una serie de variables, operadores y llamadas a métodos (construida conforme a la sintaxis del lenguaje) que se evalúa a un único valor. Entre otras cosas, las expresiones son utilizadas para realizar cálculos, para asignar valores a variables, y para ayudar a controlar la ejecución del flujo del programa. La tarea de una expresión se compone de dos partes: realiza el cálculo indicado por los elementos de la expresión y devuelve el valor obtenido como resultado del cálculo. Los operadores devuelven un valor, por lo que el uso de un operador es una expresión. Por ejemplo, la siguiente sentencia es una expresión:

```
int contador=1;
```

7.12. Bloques y ámbito

En Java el código fuente está dividido en partes separadas por llaves, denominas bloques. Cada bloque existe independiente de lo que está fuera de él, agrupando en su interior sentencias (expresiones) relacionadas.

Desde un bloque externo parece que todo lo que está dentro de llaves se ejecuta como una sentencia. Pero, ¿qué es un bloque externo? Esto tiene explicación si entendemos que existe una jerarquía de bloques, y que un bloque puede contener uno o más subbloques anidados.

El concepto de ámbito está estrechamente relacionado con el concepto de bloque y es muy importante cuando se trabaja con variables en Java. El ámbito se refiere a cómo las secciones de un programa (bloques) afectan el tiempo de vida de las variables. Toda variable tiene un ámbito, en el que es usada, que viene determinado por los bloques. Una variable definida en un bloque interno no es visible por el bloque externo.

Las llaves de separación son importantes no sólo en un sentido lógico, ya que son la forma de que el compilador diferencie dónde acaba una sección de código y dónde comienza otra, sino que tienen una connotación estética que facilita la lectura de los programas al ser humano.

Así mismo, para identificar los diferentes bloques se utilizan sangrías. Las sangrías se utilizan para el programador, no para el compilador. Por ejemplo:

```
{
	// Bloque externo
	int x = 1;
	{
		// Bloque interno invisible al exterior
		int y = 2;
	}
	x = y; // Da error porque la "y" está fuera de ámbito
}
```


8. Estructuras de control en Java

El lenguaje Java soporta las siguientes estructuras de control:

Sentencia	Clave
Toma de decisión	if-else, switch-case
Bucle	for, while, do-while
Misceláneo	break, continue, label:, return, goto

Aunque goto es una palabra reservada, actualmente el lenguaje Java no soporta la sentencia goto. Se puede utilizar las sentencias de bifurcación en su lugar.

8.1. Sentencia if-else

La sentencia if-else de Java dota a los programas de la habilidad de ejecutar distintos conjuntos de sentencias según algún criterio. La sintaxis es:

```
if ( condición )
	Bloque de código a ejecutar si la condición es cierta
else
	Bloque de código a ejecutar si la condición es falsa
```

La parte del else es opcional, y un bloque de código puede ser simplemente la sentencia vacía; para representar que en ese caso no se ha de ejecutar nada.

Ejemplo de if-else:

```
// La respuesta es Aceptar o Cancelar
if (respuesta == Aceptar)
{
      // código para realizar la acción Aceptar
      System.out.println( "Su peticion esta siendo atendida" );
}
else
{
      // código para realizar la acción Cancelar
      System.out.println( "Cancelando accion" );
}
```

Se pueden anidar expresiones if-else, para poder implementar aquellos casos con múltiples acciones.

8.2. Sentencia switch

Mediante la sentencia switch se puede seleccionar entre varias sentencias según el valor de cierta expresión. La forma general de switch es la siguiente:

```
switch (expresionMultivalor)
{
      case valor1:
            conjuntoDeSentencias;
            break;
      case valor2:
            conjuntoDeSentencias;
            break;
      ...
      case valorN:
            conjuntoDeSentencias;
            break;
      default:
      conjuntoDeSentencias;
      break;
}
```


Cada sentencia case debe ser única y el valor que evalúa debe ser del mismo tipo que el devuelto por la expresiónMultivalor de la sentencia switch.

Finalmente, se puede usar la sentencia default para manejar los valores que no son explícitamente contemplados por alguna de las sentencias case. Su uso es altamente recomendado. Por ejemplo:

```
int meses;
switch ( meses )
{
      case 1:
            System.out.println( "Enero" );
            break;
      case 2:
            System.out.println( "Febrero" );
            break;
```

```
        case 3:
              System.out.println( "Marzo" );
              break;
        // . . .
        case 12:
              System.out.println( "Diciembre" );
              break;
        default:
              System.out.println( "Mes no valido" );
              break;
}
```

8.3. Bucle while

El bucle while es el bucle básico de iteración. Sirve para realizar una acción sucesivamente mientras se cumpla una determinada condición. La sintaxis es:

```
while ( expresiónBooleana )
{
      sentencias;
}
```

Las sentencias se ejecutan mientras la expresiónBooleana tenga un valor de verdadero. Por ejemplo: multiplicar un número por 2 hasta que sea mayor que 100:

```
int i = 1;
while ( i <= 100 )
{
      i = i * 2;
}
```

8.4. Bucle do-while

El bucle do-while es similar al bucle while, pero en el bucle while la expresión se evalúa al principio del bucle y en el bucle do-while la evaluación se realiza al final. La sintaxis es:


```
do
{
      sentencias;
} while ( expresiónBooleana );
```

La sentencia do-while es el constructor de bucles menos utilizado en la programación, pero tiene sus usos, cuando el bucle deba ser ejecutado por lo menos una vez. Por ejemplo, cuando se lee información de un archivo, se sabe que siempre se debe leer por lo menos un carácter:

```
int c;
do
{
      c = System.in.read( );
      // Sentencias para tratar el carácter c
} while ( c != -1 ); // No se puede leer más (Fin fichero)
```

8.5. Bucle for

Mediante la sentencia for se resume un bucle do-while con una iniciación previa. Es muy común que en los bucles while y do-while se inicien las variables de control de número de pasadas por el bucle, inmediatamente antes de comenzar los bucles. Por eso el bucle for está tan extendido. Su forma general sería:

```
for ( iniciación ; terminación ; incremento )
      sentencias;
```

La iniciación es una sentencia que se ejecuta una vez antes de entrar en el bucle. La terminación es una expresión que determina cuándo se debe terminar el bucle. Esta expresión se evalúa al final de cada iteración del bucle. Cuando la expresión se evalúa a falso, el bucle termina. El incremento es una expresión que es invocada en cada iteración del bucle. En realidad, puede ser una acción cualquiera, aunque se suele utilizar para incrementar una variable contador:

```
for ( int i = 0 ; i < 10 ; i++ )
```

Algunos (o todos) estos componentes pueden omitirse, pero los puntos y coma siempre deben aparecer (aunque sea sin nada entre sí).

Se debe utilizar el bucle for cuando se conozcan las restricciones del bucle (su instrucción de iniciación, criterio de terminación e instrucción de incremento). Por ejemplo, los bucles for son utilizados comúnmente para iterar sobre los elementos de una matriz, o los caracteres de una cadena:

```
// cad es una cadena (String)
for ( int i = 0; i < cad.length() ; i++)
{
      // hacer algo con el elemento i-ésimo de cad
}
```

8.6. Sentencia break

La sentencia break provoca que el flujo de control salte a la sentencia inmediatamente posterior al bloque en curso. Ya se ha visto anteriormente la sentencia break dentro de la sentencia switch. El uso de la sentencia break con sentencias etiquetadas es una alternativa al uso de la sentencia goto, que no es soportada por el lenguaje Java. Pero se desaconseja esta forma de programación, basada en goto, y con saltos de flujo no controlados.

8.7. Sentencia continue

Del mismo modo que en un bucle se puede desear romper la iteración, también se puede desear continuar con el bucle, pero dejando pasar una determinada iteración. Se puede usar la sentencia continue dentro de los bucles para saltar a otra sentencia, aunque no puede ser llamada fuera de un bucle.

Tras la invocación a una sentencia continue se transfiere el control a la condición de terminación del bucle, que vuelve a ser evaluada en ese momento, y el bucle continúa o no dependiendo del resultado de la evaluación. En los bucles for además en ese momento se ejecuta la cláusula de incremento (antes de la evaluación). Por ejemplo, el siguiente fragmento de código imprime los números del 0 al 9 no divisibles por 3:

```
for ( int i = 0 ; i < 10 ; i++ )
{
      if ( ( i % 3 ) == 0 )
            continue;
      System.out.print( " " + i );
}
```

8.8. Sentencia return

La última de las sentencias de salto es la sentencia return, que se puede usar para salir del método en curso y retornar a la sentencia dentro de la cual se realizó la llamada. Para devolver un valor, simplemente se debe poner el valor (o una expresión que calcule el valor) a continuación de la palabra return. El valor devuelto por return debe coincidir con el tipo declarado como valor de retorno del método.

Cuando un método se declara como void se debe usar la forma de return sin indicarle ningún valor. Esto se hace para no ejecutar todo el código del programa.

```
int contador;
boolean condicion;
int devuelveContadorIncrementado()
{
      return ++contador;
}

void metodoReturn()
{
      //Sentencias
      if ( condicion == true )
            return;
      //Más sentencias a ejecutar si condición no vale true
}
```

8.9. Bloque try {...} catch {...} finally {...}

Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo, prácticamente sólo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje Java, una Exception es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas excepciones son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras excepciones, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (definiendo por ejemplo un nuevo path del fichero no encontrado).

Los errores se representan mediante clases derivadas de la clase Throwable, pero los que tiene que chequear un programador derivan de Exception (java.lang.Exception que a su vez deriva de Throwable). Existen algunos tipos de excepciones que Java obliga a tener en cuenta. Esto se hace mediante el uso de bloques try, catch y finally.

El código dentro del bloque try está “vigilado”. Si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque catch, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques catch como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque finally, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error. En el caso en que el código de un método pueda generar una Exception y no se desee incluir en dicho método la gestión del error (es decir los bucles try/catch correspondientes), es necesario que el método pase la Exception al método desde el que ha sido

llamado. Esto se consigue mediante la adición de la palabra throws seguida del nombre de la Exception concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques try/catch o volver a pasar la Exception. De esta forma se puede ir pasando la Exception de un método a otro hasta llegar al último método del programa, el método main().

En el siguiente ejemplo se presentan dos métodos que deben "controlar" una IOException relacionada con la lectura ficheros y una MyException propia. El primero de ellos (metodo1) realiza la gestión de las excepciones y el segundo (metodo2) las pasa al siguiente método.

```
void metodo1()
{
	...
	try {
		... // Código que puede lanzar las excepciones
		IOException y MyException
	}
	catch (IOException e1)
	{
		// Se ocupa de IOException simplemente dando aviso
		System.out.println(e1.getMessage());

	}
	catch (MyException e2)
	{
		// Se ocupa de MyException dando un aviso y
		//finalizando la función
		System.out.println(e2.getMessage());
		return;
	}
	finally
	{
		// Sentencias que se ejecutarán, en cualquier caso
		...
	}
	...
} // Fin del metodo1


void metodo2() throws IOException, MyException
{
	...
	// Código que puede lanzar las excepciones IOException y
	//MyException
	...
} // Fin del metodo2
```


9. Clases y objetos

9.1. Clases

El elemento básico de la programación orientada a objetos en Java es la clase. Una clase define la forma y comportamiento de un objeto. Para crear una clase sólo se necesita un archivo fuente que contenga la palabra clave reservada class seguida de un identificador legal y un bloque delimitado por dos llaves para el cuerpo de la clase.

Un archivo de Java debe tener el mismo nombre que la clase que contiene, y se les suele asignar la extensión ".java". Por ejemplo, la clase HolaMundo se guardaría en un fichero que se llamase

HolaMundo.java. Hay que tener presente que en Java se diferencia entre mayúsculas y minúsculas; el nombre de la clase y el de archivo fuente han de ser exactamente iguales.

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

Posteriormente necesitamos compilar el programa para obtener el binario con extensión class:

```
javac HolaMundo.java
```

Y después procedemos a ejecutar el programa:

```
java HolaMundo
```

Los programas en Java completos constarán por lo general de varias clases de Java en distintos archivos fuente. Una clase es una plantilla para un objeto. Por lo tanto, define la estructura de un objeto y su interfaz funcional, en forma de métodos. Cuando se ejecuta un programa en Java, el sistema utiliza definiciones de clase para crear instancias de las clases, que son los objetos reales. Los términos instancia y objeto se utilizan de manera indistinta. La forma general de una definición de clase es:

```
class Nombre_De_Clase
{
	tipo_de_variable nombre_de_atributo1;
	tipo_de_variable nombre_de_atributo2;
	// . . .

	tipo_devuelto nombre_de_método1( lista_de_parámetros )
	{
		cuerpo_del_método1;
	}
	tipo_devuelto nombre_de_método2( lista_de_parámetros )
	{
		cuerpo_del_método2;
	}
	// . . .
}
```


Los tipos tipo_de_variable y tipo_devuelto han de ser tipos simples Java o nombres de otras clases ya definidas. Tanto Nombre_De_Clase, como los nombres_de_atributo y nombres_de_metodo han de ser identificadores Java válidos.

9.2. Atributos

Los datos se encapsulan dentro de una clase declarando variables dentro de las llaves de apertura y cierre de la declaración de la clase, variables que se conocen como atributos. Se declaran igual que las variables locales de un método en concreto. Por ejemplo, este es un programa que declara una clase MiPunto, con dos atributos enteros llamados x e y.

```
class MiPunto
{
	int x, y;
}
```

Los atributos se pueden declarar con dos clases de tipos: un tipo simple Java, o el nombre de una clase (será una referencia a objeto). Cuando se realiza una instancia de una clase (creación de un objeto) se reservará en la memoria un espacio para un conjunto de datos como el que definen los atributos de una clase. A este conjunto de variables se le denomina variables de instancia.

9.3. Métodos

Los métodos son subrutinas que definen la interfaz de una clase, sus capacidades y comportamiento. Un método ha de tener por nombre cualquier identificador legal distinto de los ya utilizados por los nombres de la clase en que está definido. Los métodos se declaran al mismo nivel que las variables de instancia dentro de una definición de clase.

En la declaración de los métodos se define el tipo de valor que devuelven y a una lista formal de parámetros de entrada, de sintaxis tipo identificador separadas por comas. La forma general de una declaración de método es:

```
tipo_devuelto nombre_de_método( lista-formal-de-parámetros )
{
      cuerpo_del_método;
}
```

Por ejemplo, el siguiente método devuelve la suma de dos enteros:

```
int metodoSuma( int paramX, int paramY )
{
      return ( paramX + paramY );
}
```

En el caso de que no se desee devolver ningún valor se deberá indicar como tipo la palabra reservada void. Así mismo, si no se desean parámetros, la declaración del método debería incluir un par de paréntesis vacíos ():

```
void metodoVacio( ) { ..... };
```


Los métodos son llamados indicando una instancia individual de la clase, que tendrá su propio conjunto único de variables de instancia, por lo que los métodos se pueden referir directamente a ellas. Por ejemplo, el método inicia() para establecer valores a las dos variables de instancia sería el siguiente:

```
void inicia( int paramX, int paramY )
{
      x = paramX;
      y = paramY;
}
```

9.4. La instanciación de las clases: los objetos

9.4.1. Referencias a objetos e instancias

Los tipos simples de Java describían el tamaño y los valores de las variables. Cada vez que se crea una clase se añade otro tipo de dato que se puede utilizar igual que uno de los tipos simples. Por ello al declarar una nueva variable, se puede utilizar un nombre de clase como tipo. A estas variables se las conoce como referencias a objeto. Todas las referencias a objeto son compatibles también con las instancias de subclases de su tipo:

```
MiPunto p;
```

Esta es una declaración de una variable p que es una referencia a un objeto de la clase MiPunto, de momento con un valor por defecto de null.

9.4.2. Constructores

Las clases pueden implementar un método especial llamado constructor. Un constructor es un método que inicia un objeto inmediatamente después de su creación. De esta forma nos evitamos el tener que iniciar las variables explícitamente para su iniciación.

El constructor tiene exactamente el mismo nombre de la clase que lo implementa; no puede haber ningún otro método que comparta su nombre con el de su clase. Una vez definido, se llamará automáticamente al constructor al crear un objeto de esa clase (al utilizar el operador new).

El constructor no devuelve ningún tipo, ni siquiera void. Su misión es iniciar todo estado interno de un objeto (sus atributos), haciendo que el objeto sea utilizable inmediatamente; reservando memoria para sus atributos, iniciando sus valores...

Por ejemplo:

```
MiPunto( )
{
      inicia( -1, -1 );
}
```

Este constructor denominado constructor por defecto, por no tener parámetros, establece el valor -1 a las variables de instancia x e y de los objetos que construya. El compilador, por defecto, llamará al constructor de la superclase Object() si no se especifican parámetros en el constructor.

Este otro constructor, sin embargo, recibe dos parámetros:


```
MiPunto( int paraX, int paraY )
{
      inicia( paramX, paramY );
}
```

La lista de parámetros especificada después del nombre de una clase en una sentencia new se utiliza para pasar parámetros al constructor.

Se llama al método constructor justo después de crear la instancia y antes de que new devuelva el control al punto de la llamada. Así, cuando ejecutamos el siguiente programa:

```
MiPunto p1 = new MiPunto(10, 20);
System.out.println( "p1.- x = " + p1.x + " y = " + p1.y );
```

Se muestra en la pantalla:

```
p1.- x = 10 y = 20
```

Para crear un programa Java que contenga ese código, se debe de crear una clase que contenga un método main(). El intérprete java se ejecutará el método main de la clase que se le indique como parámetro.

9.4.3. El operador new

El operador new crea una instancia de una clase (objetos) y devuelve una referencia a ese objeto. Por ejemplo:

```
MiPunto p2 = new MiPunto(2,3);
```

Este es un ejemplo de la creación de una instancia de MiPunto, que es controlador por la referencia a objeto p2.

Hay una distinción crítica entre la forma de manipular los tipos simples y las clases en Java: Las referencias a objetos realmente no contienen a los objetos a los que referencian. De esta forma se pueden crear múltiples referencias al mismo objeto, como, por ejemplo:

```
MiPunto p3 = p2;
```

Aunque tan sólo se creó un objeto MiPunto, hay dos variables (p2 y p3) que lo referencian. Cualquier cambio realizado en el objeto referenciado por p2 afectará al objeto referenciado por p3. La asignación de p2 a p3 no reserva memoria ni modifica el objeto.

De hecho, las asignaciones posteriores de p2 simplemente desengancharán p2 del objeto, sin afectarlo:

```
p2 = null; // p3 todavía apunta al objeto creado con new
```

Aunque se haya asignado null a p2, p3 todavía apunta al objeto creado por el operador new.

Cuando ya no haya ninguna variable que haga referencia a un objeto, Java reclama automáticamente la memoria utilizada por ese objeto, a lo que se denomina recogida de basura.

Cuando se realiza una instancia de una clase (mediante new) se reserva en la memoria un espacio para un conjunto de datos como el que definen los atributos de la clase que se indica en la instanciación. A este conjunto de variables se le denomina variables de instancia.

La potencia de las variables de instancia es que se obtiene un conjunto distinto de ellas cada vez que se crea un objeto nuevo. Es importante el comprender que cada objeto tiene su propia copia de las variables de instancia de su clase, por lo que los cambios sobre las variables de instancia de un objeto no tienen efecto sobre las variables de instancia de otro.

El siguiente programa crea dos objetos MiPunto y establece los valores de x e y de cada uno de ellos de manera independiente.

```
MiPunto p4 = new MiPunto( 10, 20 );
MiPunto p5 = new MiPunto( 42, 99 );
```

9.4.4. El operador .

El operador punto (.) se utiliza para acceder a las variables de instancia y los métodos contenidos en un objeto, mediante su referencia a objeto:

```
referencia_a_objeto.nombre_de_variable_de_instancia
referencia_a_objeto.nombre_de_método( lista-de-parámetros );
```

Hemos creado un ejemplo completo que combina los operadores new y punto para crear un objeto MiPunto, almacenar algunos valores en él e imprimir sus valores finales:

```
MiPunto p6 = new MiPunto( 10, 20 );
System.out.println ("p6.- 1. X=" + p6.x + " , Y=" + p6.y);
```

```
p6.inicia( 30, 40 );
System.out.println ("p6.- 2. X=" + p6.x + " , Y=" + p6.y);
```

Durante las impresiones (método println()) se accede al valor de las variables mediante p6.x y p6.y, y entre una impresión y otra se llama al método inicia(), cambiando los valores de las variables de instancia. Este es uno de los aspectos más importantes de la diferencia entre la programación orientada a objetos y la programación estructurada. Cuando se llama al método p6.inicia(), lo primero que se hace en el método es sustituir los nombres de los atributos de la clase por las correspondientes variables de instancia del objeto con que se ha llamado. Así por ejemplo x se convertirá en p6.x.

9.4.5. This

Java incluye un valor de referencia especial llamado this, que se utiliza dentro de cualquier método para referirse al objeto actual. El valor this se refiere al objeto sobre el que ha sido llamado el método actual. Se puede utilizar this siempre que se requiera una referencia a un objeto del tipo de una clase actual. Si hay dos objetos que utilicen el mismo código, seleccionados a través de otras instancias, cada uno tiene su propio valor único de this.

Un refinamiento habitual es que un constructor llame a otro para construir la instancia correctamente. El siguiente constructor llama al constructor parametrizado MiPunto(x,y) para terminar de iniciar la instancia:

```
MiPunto()
{
      this( -1, -1 ); // Llama al constructor parametrizado
}
```

En Java se permite declarar variables locales, incluyendo parámetros formales de métodos, que se solapen con los nombres de las variables de instancia.

No se utilizan x e y como nombres de parámetro para el método inicia, porque ocultarían las variables de instancia x e y reales del ámbito del método. Si lo hubiésemos hecho, entonces x se hubiera referido al parámetro formal, ocultando la variable de instancia x:

```
void inicia2( int x, int y )
{
      x = x; // Ojo, no modificamos la variable de instancia!!!
      this.y = y; // Modificamos la variable de instancia!!!
}
```

9.4.6. La destrucción del objeto

Cuando un objeto no va a ser utilizado, el espacio de memoria dinámica que utiliza ha de ser liberado, así como los recursos que poseía, permitiendo al programa disponer de todos los recursos posibles. A esta acción se la da el nombre de destrucción del objeto.

En Java la destrucción se puede realizar de forma automática o de forma personalizada, en función de las características del objeto.

Destrucción por defecto, recogida de basura: El intérprete de Java posee un sistema de recogida de basura, que por lo general permite que no nos preocupemos de liberar la memoria asignada explícitamente. El recolector de basura será el encargado de liberar una zona de memoria dinámica que había sido reservada mediante el operador new, cuando el objeto ya no va a ser

utilizado más durante el programa (por ejemplo, sale del ámbito de utilización, o no es referenciado nuevamente). El sistema de recogida de basura se ejecuta periódicamente, buscando objetos que ya no estén referenciados.

Destrucción personalizada, finalize: A veces una clase mantiene un recurso que no es de Java como un descriptor de archivo o un tipo de letra del sistema de ventanas. En este caso sería acertado el utilizar la finalización explícita, para asegurar que dicho recurso se libera. Esto se hace mediante la destrucción personalizada. Para especificar una destrucción personalizada se añade un método a la clase con el nombre finalize:

```
class ClaseFinalizada
{
      ClaseFinalizada()
      {     // Constructor
            // Reserva del recurso no Java o recurso compartido
      }
      protected void finalize()
      {
      // Liberación del recurso no Java o recurso compartido
      }
}
```

El intérprete de Java llama al método finalize(), si existe cuando vaya a reclamar el espacio de ese objeto, mediante la recogida de basura. Debe observarse que el método finalize() es de tipo protected void y por lo tanto deberá de sobreescribirse con este mismo tipo.

10. Entrada/Salida

Normalmente, cuando se codifica un programa, se hace con la intención de que ese programa pueda interactuar con los usuarios del mismo, es decir, que el usuario pueda pedirle que realice cosas y pueda suministrarle datos con los que se quiere que haga algo. Una vez introducidos los datos y las órdenes, se espera que el programa manipule de alguna forma esos datos para proporcionarnos una respuesta a lo solicitado.

Además, en muchas ocasiones interesa que el programa guarde los datos que se le han introducido, de forma que si el programa termina los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma más normal de hacer esto es mediante la utilización de ficheros que se guardarán en un dispositivo de memoria no volátil (normalmente un disco).

A todas estas operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como Entrada/Salida (E/S). Existen dos tipos de E/S; la E/S estándar que se realiza con el terminal del usuario y la E/S a través de fichero, en la que se trabaja con ficheros de disco.

Todas las operaciones de E/S en Java vienen proporcionadas por el paquete estándar de la API de Java denominado **java.io** que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros. Veamos lo más importante sobre esto:

10.1. Entrada/Salida estándar

Aquí sólo trataremos la entrada/salida que se comunica con el usuario a través de la pantalla o de la ventana del terminal. El acceso a la entrada y salida estándar es controlado por tres objetos que se crean automáticamente al iniciar la aplicación: **System.in, System.out y System.err.**

10.1.1. System.in

Este objeto implementa la entrada estándar (normalmente el teclado). Los métodos que nos proporciona para controlar la entrada son entre otros:

- **read():** Devuelve el carácter que se ha introducido por el teclado leyéndolo del buffer de entrada y lo elimina del buffer para que en la siguiente lectura sea leído el siguiente carácter. Si no se ha introducido ningún carácter por el teclado devuelve el valor -1.
- **skip(n):** Ignora los Q caracteres siguientes de la entrada.

10.1.2. System.out

Este objeto implementa la salida estándar. Los métodos que nos proporciona para controlar la salida son entre otros:

- **print(a):** Imprime a en la salida, donde a puede ser cualquier tipo básico Java ya que Java hace su conversión automática a cadena.
- **println(a):** Es idéntico a print(a) salvo que con println(a) se imprime un salto de línea al final de la impresión de a.

10.1.3. System.err

Este objeto implementa la salida en caso de error. Normalmente esta salida es la pantalla o la ventana del terminal como con System.out, pero puede ser interesante redirigirlo, por ejemplo, hacia un fichero, para diferenciar claramente ambos tipos de salidas. Las funciones que ofrece este objeto son idénticas a las proporcionadas por System.out.

EJEMPLO: A continuación, vemos un ejemplo del uso de estas funciones que acepta texto hasta que se pulsa el retorno de carro e informa del número de caracteres introducidos.

```
import java.io.*;

class CuentaCaracteres
{
	public static void main(String args[]) throws IOException
	{
		int contador=0;
		while(System.in.read()!='\n')
			contador++;
		System.out.println("Tecleados "+contador+" caracteres.");
	}
}
```


10.2. Entrada/Salida por fichero

10.2.1. Tipos de ficheros

En Java es posible utilizar dos tipos de ficheros (de texto o binarios) y dos tipos de acceso a los ficheros (secuencial o aleatorio).

Los ficheros de texto están compuestos de caracteres legibles, mientras que los binarios pueden almacenar cualquier tipo de datos (int, float, boolean…).

Una lectura secuencial implica tener que acceder a un elemento antes de acceder al siguiente, es decir, de una manera lineal (sin saltos). Sin embargo, los ficheros de acceso aleatorio permiten

acceder a sus datos de una forma aleatoria, esto es indicando una determinada posición desde la que leer/escribir.

10.2.2. Clases

En el paquete java.io existen varias clases de las cuales podemos crear instancias de clases para tratar todo tipo de ficheros. Veamos las tres principales:

- **FileOutputStream**: Fichero de salida de texto. Representa ficheros de texto para escritura a los que se accede de forma secuencial.
- **FileInputStream**: Fichero de entrada de texto. Representa ficheros de texto de sólo lectura a los que se accede de forma secuencial.
- **RandomAccessFile**: Fichero de entrada o salida binario con acceso aleatorio. Es la base para crear los objetos de tipo fichero de acceso aleatorio. Estos ficheros permiten multitud de operaciones; saltar hacia delante y hacia atrás para leer la información que necesitemos en cada momento, e incluso leer o escribir partes del fichero sin necesidad de cerrarlo y volverlo a abrir en un modo distinto.

10.2.3. Uso de ficheros

Para tratar con un fichero siempre hay que actuar de la misma manera:

1) **Se abre el fichero:** Para ello hay que crear un objeto de la clase correspondiente al tipo de fichero que vamos a manejar, y el tipo de acceso que vamos a utilizar:

   ```
   TipoDeFichero obj = new TipoDeFichero( ruta );
   ```

   Donde ruta es la ruta de disco en que se encuentra el fichero.

   Este formato es válido, excepto para los objetos de la clase RandomAccessFile (acceso aleatorio), para los que se ha de instanciar de la siguiente forma:

   ```
   RandomAccessFile obj = new RandomAccessFile( ruta, modo );
   ```

   Donde modo es una cadena de texto que indica el modo en que se desea abrir el fichero; "r" para sólo lectura o "rw" para lectura y escritura.

2) **Se utiliza el fichero:** Para ello cada clase presenta diferentes métodos de acceso para escribir o leer en el fichero.
3) **Gestión de excepciones** (opcional, pero recomendada): Se puede observar que todos los métodos que utilicen clases de este paquete deben tener en su definición una cláusula throws IOException. Los métodos de estas clases pueden lanzar excepciones de esta clase (o sus hijas) en el transcurso de su ejecución, y dichas excepciones deben de ser capturadas y debidamente gestionadas para evitar problemas.
4) **Se cierra el fichero y se destruye el objeto:** Para cerrar un fichero lo que hay que hacer es destruir el objeto. Esto se puede realizar de dos formas, dejando que sea el recolector de basura de Java el que lo destruya cuando no lo necesite o destruyendo el objeto explícitamente mediante el uso del procedimiento close() del objeto:

   ```
   obj.close()
   ```

10.2.4. La clase FileOutputStream

Mediante los objetos de esta clase escribimos en ficheros de texto de forma secuencial. Presenta el método write() para la escritura en el fichero. Presenta varios formatos:

- **int write (int c):** Escribe el carácter en el fichero.
- **int write (byte a[]):** Escribe el contenido del vector en el fichero.
- **int write (byte a[], int off, int len):** Escribe len caracteres del vector a en el fichero, comenzando desde la posición off.

El siguiente **ejemplo** crea el fichero de texto “carta.txt" a partir de un texto que se le introduce por teclado:

```
import java.io.*;

class CreaCarta
{
	public static void main(String args[]) throws IOException
	{
		int c;
		FileOutputStream f=new FileOutputStream("carta.txt");
		while( ( c=System.in.read() ) != -1 )
			f.write( (char)c );
		f.close();
	}
}
```

10.2.5. La clase FileInputStream

Mediante los objetos de esta clase leemos de ficheros de texto de forma secuencial. Presenta el método read() para la lectura del fichero. Este método se puede invocar de varias formas.

- **int read():** Devuelve el siguiente carácter del fichero.
- **int read(byte a[]):** Llena el vector a con los caracteres leídos del fichero. Devuelve la longitud del vector que se ha llenado si se realizó con éxito o –1 si no había suficientes caracteres en el fichero para llenar el vector.
- **int read(byte a[], int off, int len):** Lee len caracteres del fichero, insertándolos en el vector a.

Todos ellos devuelven -1 si se ha llegado al final del fichero (momento de cerrarle).

El siguiente **ejemplo** muestra el fichero de texto “carta.txt" en pantalla:

```
import java.io.*;
class MuestraCarta
{
	public static void main(String args[]) throws IOException
	{
		int c;
		FileInputStream f=new FileInputStream("carta.txt");
		while( ( c=f.read() ) != -1 )
			System.out.print( (char)c );
		f.close();
	}
}
```

10.2.6. La clase RandomAccessFile

Mediante los objetos de esta clase utilizamos ficheros binarios mediante un acceso aleatorio, tanto para lectura como para escritura. En estos ficheros hay un índice que nos dice en qué posición del fichero nos encontramos, y con el que se puede trabajar para posicionarse en el fichero.

Métodos de desplazamiento: Cuenta con una serie de funciones para realizar el desplazamiento del puntero del fichero. Hay que tener en cuenta que cualquier lectura o escritura de datos se realizará a partir de la posición actual del puntero del fichero.

- **long getFilePointer():** Devuelve la posición actual del puntero del fichero.
- **void seek (long l):** Coloca el puntero del fichero en la posición indicada por l. Un fichero siempre empieza en la posición 0.
- **int skipBytes (int n):** Intenta saltar n bytes desde la posición actual.
- **long length():** Devuelve la longitud del fichero.
- **void setLength(long l):** Establece a l el tamaño de este fichero.
- **fileDescriptor getFD():** Devuelve el descriptor de este fichero.

Métodos de escritura: La escritura del fichero se realiza con una función que depende el tipo de datos que se desee escribir.

- **void write (byte b[], int ini, int len**): Escribe len caracteres del vector b.
- **void write (int i):** Escribe i (un byte) en el flujo.
- **void writeBoolean (boolean b):** Escribe el boolean b como un byte.
- **void writeByte (int i):** Escribe i como un byte.
- **void writeBytes (String s):** Escribe la cadena s tratada como bytes, no caracteres.
- **void writeChar (int i):** Escribe i como 1 byte.
- **void writeChars (String s):** Escribe la cadena s.
- **void writeDouble (double d):** Convierte d a long y le escribe como 8 bytes.
- **void writeFloat (float f):** Convierte f a entero y le escribe como 4 bytes.
- **void writeShort (int i):** Escribe i como 2 bytes.
- **void writeInt (int i):** Escribe i como 4 bytes.
- **void writeLong (long v):** Escribe v como 8 bytes.
- **void writeUTF (String s):** Escribe la cadena s utilizando la codificación UTF-8.

Métodos de lectura: La lectura del fichero se realiza con una función que depende del tipo de datos que queremos leer. Si no es posible la lectura devuelven -1.

- **boolean readBoolean():** Lee un byte y devuelve false si vale 0 o true si no.
- **byte readByte():** Lee y devuelve un byte.
- **char readChar():** Lee y devuelve un caracter.
- **double readDouble():** Lee 8 bytes y devuelve un double.
- **float readFloat():** Lee 4 bytes, y devuelve un float.
- **void readFully (byte b[]):** Lee bytes del fichero y los almacena en un vector b.
- **void readFully (byte b[], int ini, int len):** Lee len bytes del fichero y los almacena
- en un vector b.
- **int readInt():** Lee 4 bytes, y devuelve un int.
- **long readLong():** Lee 8 bytes, y devuelve un long.
- **short readShort():** Lee 2 bytes, y devuelve un short.
- **int readUnsignedByte():** Lee 1 byte y devuelve un valor de 0 a 255.
- **int readUnsignedShort():** Lee 2 bytes, y devuelve un valor de 0 a 65535.

- **string readUTF():** Lee una cadena codificada con el formato UTF-8.
- **int skipBytes (int n):** Salta n bytes del fichero.

Ejemplo: Vamos a crear un pequeño programa que cree y acceda a un fichero binario, mediante acceso aleatorio. El siguiente ejemplo crea un fichero binario que contiene los 100 primeros números (en orden):

```
// Crea un fichero binario con los 100 primeros números
static void creaFichBin( String ruta ) throws IOException
{
	RandomAccessFile f=new RandomAccessFile(ruta,"rw"); // E/S
	for ( int i=1; i <= 100 ; i++ )
	{
		try{
			f.writeByte( i );
		}
		catch( IOException e)
		{
			// Gestion de excepcion de ejemplo
			break; // No se puede seguir escribiendo
		}
	f.close();
}
```

El siguiente método accede al elemento cual de un fichero binario, imprimiendo la longitud del fichero, el elemento cual y su 10 veces siguiente elemento:

```
static void imprimeEltoN(String ruta, long cual) throws IOException
{
	RandomAccessFile f=new RandomAccessFile(ruta,"r");
	System.out.print( "El fichero " + ruta );
	System.out.println( " ocupa " + f.length() + " bytes." );

	f.seek( cual-1 ); // Me posiciono (-1 porque empieza en 0)
	System.out.print(" En la posicion " + f.getFilePointer() );
	System.out.println(" esta el numero " + f.readByte() );

	f.skipBytes( 9 ); // Salto 9 => Elemento 10 mas alla
	System.out.print(" 10 elementos más allá, esta el ");
	System.out.println( f.readByte() );

	f.close();
}
```

Si incluimos ambos métodos en una clase, y les llamamos con el siguiente programa principal (main()):

```
public static void main(String args[]) throws IOException
{
	String ruta="numeros.dat"; // Fichero
	creaFichBin( ruta ); // Se crea
	imprimeEltoN( ruta, 14 ); // Accedo al elemento 14.
}
```

Obtendremos la siguiente salida:

```
El fichero numeros.dat ocupa 100 bytes.
En la posicion 13 esta el numero 14
10 elementos más allá, esta el 24
```