

**G-TeC**

Facultad de Informática  
Universidad Complutense de Madrid



# **Fundamentos de programación en Java**

Editorial EME

ISBN 978-84-96285-36-2

# Contenido

<b>1. Introducción a Java .....</b>	<b>1</b>
<b>Los lenguajes de programación .....</b>	<b>1</b>
<b>Historia de Java .....</b>	<b>2</b>
<b>La plataforma de Java .....</b>	<b>3</b>
<b>Entornos de desarrollo para Java .....</b>	<b>4</b>
<b>El proceso de edición y compilación.....</b>	<b>5</b>
<b>La codificación de programas Java .....</b>	<b>6</b>
<b>El proceso de desarrollo de software .....</b>	<b>6</b>
<b>2. Estructura de un programa Java .....</b>	<b>9</b>
<b>La estructura de un programa Java .....</b>	<b>9</b>
<b>Los elementos de un programa Java.....</b>	<b>13</b>
Comentarios.....	14
Identificadores.....	14
Variables y valores .....	15
Tipos primitivos .....	17
Literales .....	18
Operadores .....	19
Expresiones.....	21
Expresiones aritmético-lógicas .....	22
Conversión de tipos .....	22
Las palabras reservadas de Java .....	23
<b>3. Clases y objetos.....</b>	<b>25</b>
<b>Clases .....</b>	<b>26</b>
<b>Los elementos de una clase .....</b>	<b>27</b>
Atributos.....	27
Métodos y constructores .....	28
Representación de clases y objetos.....	32
<b>Objetos .....</b>	<b>33</b>
La referencia null .....	35
Referencias compartidas por varios objetos.....	36

El ciclo de vida de un objeto .....	39
<b>Atributos .....</b>	<b>40</b>
<b>Métodos .....</b>	<b>41</b>
Declaración de métodos .....	44
Invocación de métodos .....	45
El método main() .....	47
Parámetros y argumentos .....	48
Paso de parámetros.....	50
El valor de retorno .....	52
Las variables locales de un método .....	52
Sobrecarga de métodos .....	53
<b>Constructores .....</b>	<b>54</b>
<b>4. Extensión de clases .....</b>	<b>59</b>
<b>Composición .....</b>	<b>59</b>
<b>Herencia .....</b>	<b>64</b>
Extensión de clases .....	64
Polimorfismo .....	70
<b>Compatibilidad de tipos .....</b>	<b>74</b>
Conversión ascendente de tipos .....	74
Conversión descendente de tipos.....	76
Jerarquía de herencia .....	77
<b>5. Ampliación de clases .....</b>	<b>79</b>
<b>Elementos de clase (Static) .....</b>	<b>79</b>
<b>Derechos de acceso .....</b>	<b>80</b>
<b>Paquetes .....</b>	<b>83</b>
Uso .....	83
Nombres.....	84
<b>Clases predefinidas.....</b>	<b>85</b>
Las clases asociadas a los tipos primitivos .....	85
La clase Math .....	86
La clase String.....	87
<b>6. Estructuras de control .....</b>	<b>89</b>
<b>Estructuras de selección.....</b>	<b>90</b>

Estructura if .....	90
Estructura if else .....	91
Estructura if else if .....	94
Estructura switch .....	95
El operador condicional .....	101
<b>Estructuras de repetición.....</b>	<b>102</b>
Estructura while .....	103
Estructura do-while .....	107
Estructura for .....	109
Uso de las estructuras de repetición.....	111
<b>Estructuras de salto.....</b>	<b>115</b>
Sentencia break.....	115
Sentencia continue.....	115
Uso de break y continue.....	115
<b>7. Estructuras de almacenamiento .....</b>	<b>119</b>
<b>Arrays.....</b>	<b>119</b>
<b>Arrays multidimensionales .....</b>	<b>127</b>
<b>Uso de arrays .....</b>	<b>131</b>
<b>Búsqueda binaria en arrays ordenados.....</b>	<b>141</b>
<b>Ordenación de arrays .....</b>	<b>144</b>
El algoritmo de ordenación "Bubble Sort".....	145
El método sort de la clase Arrays.....	149
<b>Arrays redimensionables .....</b>	<b>153</b>
<b>Uso de arrays redimensionables.....</b>	<b>156</b>
<b>8. Entrada y salida.....</b>	<b>165</b>
<b>Los flujos de Java .....</b>	<b>165</b>
<b>Entrada de datos desde el teclado .....</b>	<b>167</b>
<b>Leer y escribir en ficheros de texto.....</b>	<b>169</b>
<b>Leer y escribir objetos en ficheros.....</b>	<b>173</b>

# Anexos

<b>A. Operadores del lenguaje Java .....</b>	<b>179</b>
Operadores aritméticos .....	179
Operadores unarios y compuestos.....	179
Operadores de relación.....	181
Operadores lógicos .....	181
Orden de precedencia de los operadores .....	182
<b>B. Referencias .....</b>	<b>183</b>
El lenguaje de programación Java .....	183
El API de Java.....	184
<b>C. Glosario .....</b>	<b>185</b>

# 1. Introducción a Java

---

## Los lenguajes de programación

Los lenguajes de programación son idiomas artificiales diseñados para expresar cálculos y procesos que serán llevados a cabo por ordenadores. Un lenguaje de programación está formado por un conjunto de palabras reservadas, símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones. El proceso de programación consiste en la escritura, compilación y verificación del código fuente de un programa.

Antes de diseñar un programa es necesario entender completamente el problema que queremos resolver y conocer las restricciones de operación de la aplicación. La programación es una tarea compleja y es muy importante abordar la solución a un problema específico desde un punto de vista algorítmico. Un algoritmo es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema. Está definido por instrucciones o reglas bien definidas, ordenadas y finitas que permiten realizar una actividad. Dado un estado inicial, una entrada y una secuencia de pasos sucesivos, se llega a un estado final y se obtiene una solución.

Para programar de forma eficaz es necesario aprender a resolver problemas de una forma sistemática y rigurosa. Solo se puede llegar a realizar un buen programa si previamente se ha diseñado un algoritmo. Un algoritmo dará lugar a un programa que puede codificarse en cualquier lenguaje de programación.

Uno de los objetivos del curso de Fundamentos de Informática es que el alumno desarrolle habilidades de análisis y diseño de algoritmos simples que le puedan ser de utilidad en el futuro. Es importante tener nociones básicas de programación porque esto permitirá entender y diseñar procesos básicos en lenguajes de uso general como Java y también en aplicaciones informáticas de uso común en la ingeniería o el diseño. En la actualidad la mayoría de las aplicaciones que utilizamos a diario ofrecen posibilidades de programación. Esto facilita el diseño de pequeñas aplicaciones para automatizar tareas de uso cotidiano.

## Historia de Java

Java es un lenguaje de programación desarrollado por Sun Microsystems. Java fue presentado en la segunda mitad del año 1995 y desde entonces se ha convertido en un lenguaje de programación muy popular. Java es un lenguaje muy valorado porque los programas Java se pueden ejecutar en diversas plataformas con sistemas operativos como Windows, Mac OS, Linux o Solaris. James Gosling, el director del equipo de trabajo encargado de desarrollar Java, hizo realidad la promesa de un lenguaje independiente de la plataforma. Se buscaba diseñar un lenguaje que permitiera programar una aplicación una sola vez que luego pudiera ejecutarse en distintas máquinas y sistemas operativos. Para conseguir la portabilidad de los programas Java se utiliza un entorno de ejecución para los programas compilados. Este entorno se denomina Java Runtime Environment (JRE). Es gratuito y está disponible para los principales sistemas operativos. Esto asegura que el mismo programa Java pueda ejecutarse en Windows, Mac OS, Linux o Solaris.



“Write Once, Run Anywhere”, que podría traducirse como “programar una sola vez y después ejecutar los programas en cualquier sistema operativo”, era el objetivo del equipo de desarrollo de Java. Esta idea resume el concepto de portabilidad. Los programas Java son portables, es decir, independientes de la plataforma, porque pueden ejecutarse en cualquier ordenador o dispositivo móvil, independientemente del sistema operativo que tengan instalado: Un programa Java puede ejecutarse en un ordenador de mesa, un ordenador portátil, una tableta, un teléfono, un reproductor de



música o en cualquier otro dispositivo móvil con cualquier sistema operativo.

### La plataforma de Java

Los programas Java se compilan a un lenguaje intermedio, denominado Bytecode. Este código es interpretado por la máquina virtual de Java del entorno de ejecución (JRE) y así se consigue la portabilidad en distintas plataformas. El JRE es una pieza intermedia entre el código Bytecode y los distintos sistemas operativos existentes en el mercado. Un programa Java compilado en Bytecode se puede ejecutar en sistemas operativos como Windows, Linux, Mac Os, Solaris, BlackBerry OS, iOS o Android utilizando el entorno de ejecución de Java (JRE) apropiado.

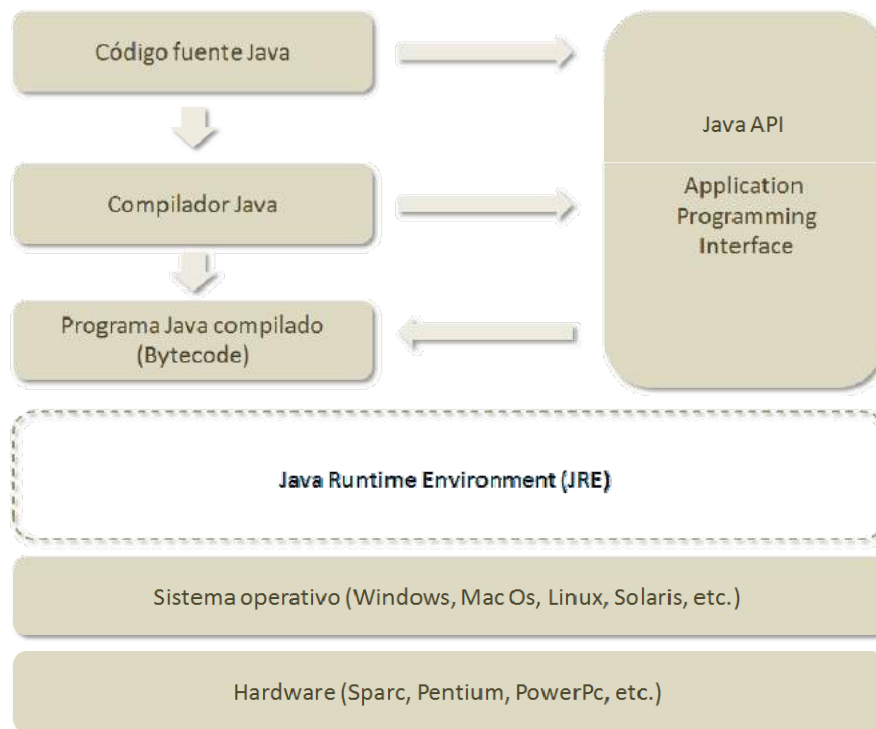
Una de las características más importantes de los lenguajes de programación modernos es la portabilidad. Como se ha comentado antes, un programa es portable cuando es independiente de la plataforma y puede ejecutarse en cualquier sistema operativo y dispositivo físico. Los programas Java son portables porque se ejecutan en cualquier plataforma. Sucede algo parecido con las fotografías o los ficheros PDF. Las fotografías con formato JPEG son portables porque un archivo JPEG lo podemos visualizar con distintos visores de fotos y en dispositivos como ordenadores, tabletas o teléfonos. El formato JPEG es un estándar para almacenar archivos de imagen. Todas las imágenes JPEG tienen el mismo formato y los visores de fotos están diseñados para mostrar las imágenes con este formato. De forma similar, los archivos PDF (Portable Document Format) son portables. El formato PDF fue desarrollado por Adobe Systems con la idea de que estos archivos se puedan ver en cualquier dispositivo que tenga instalado Adobe Acrobat Reader, el software de visualización de documentos PDF.

La portabilidad de Java ha contribuido a que muchas empresas hayan desarrollado sus sistemas de comercio electrónico y sus sistemas de información en Internet con Java. El proceso de desarrollo y de mantenimiento de los sistemas resulta menos costoso y las aplicaciones son compatibles con distintos sistemas operativos.

La evolución del lenguaje de programación Java ha sido muy rápida. La plataforma de desarrollo de Java, denominada Java Development Kit (JDK), se ha ido ampliando y cada vez incorpora a un número mayor de programadores en todo el mundo. En realidad Java no solo es un lenguaje de programación. Java es un lenguaje, una plataforma de desarrollo, un entorno de ejecución y un conjunto de librerías para desarrollo de

programas sofisticados. Las librerías para desarrollo se denominan Java Application Programming Interface (Java API).

El siguiente esquema muestra los elementos de la plataforma Java, desde el código fuente, el compilador, el API de Java, los programas compilados en Bytecode y el entorno de ejecución de Java. Este entorno de ejecución (JRE) y la máquina virtual (JVM) permiten que un programa compilado Java se ejecute en distintos sistemas operativos.



## Entornos de desarrollo para Java

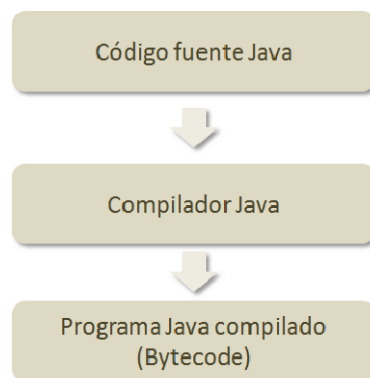
Existen distintos entornos de desarrollo de aplicaciones Java. Este tipo de productos ofrecen al programador un entorno de trabajo integrado para facilitar el proceso completo de desarrollo de aplicaciones, desde el diseño, la programación, la documentación y la verificación de los programas. Estos productos se denominan IDE (Integrated Development Environment).

Existen entornos de distribución libre como: NetBeans, Eclipse o BlueJ. Entre los productos comerciales están JBuilder o JCreatorPro.

Para utilizar un entorno de desarrollo es necesario instalar el Java Runtime Environment (JRE) apropiado para el sistema operativo. El JRE se descarga de la página de Oracle Java.

### El proceso de edición y compilación

En Java, al igual que en otros lenguajes de programación, se sigue el siguiente proceso: edición del código fuente, compilación y ejecución. Los programas Java se desarrollan y se compilan para obtener un código denominado Bytecode que es interpretado por una máquina virtual de Java (Java Virtual Machine).



La edición del programa fuente se realiza escribiendo el programa Java en un editor de texto como el Bloc de notas o utilizando un entorno integrado de desarrollo. El código fuente se almacena en un fichero de tipo `.java`.

La compilación se realiza con el compilador Java `javac` o utilizando un entorno integrado de desarrollo. Durante el proceso de compilación se verifica que el código fuente cumple la definición léxica, sintáctica y semántica de Java. Esto significa que el compilador comprueba que el código fuente se compone de palabras válidas en Java y que los comandos Java tienen la forma sintáctica correcta. Si durante el proceso de compilación el compilador detecta los errores que ha cometido el programador y le informa de los problemas que ha encontrado para que pueda corregirlos. Si durante la compilación no se detectan errores, se genera un fichero de tipo `class` en Bytecode. Una vez finalizado el proceso de compilación se puede ejecutar el programa. Para esto, es necesario que la máquina virtual de Java interprete el código Bytecode y ejecute la aplicación.

# La codificación de programas Java

El estilo de programación o codificación de los programas Java es muy importante. La legibilidad de un programa determina en buena medida que se haya desarrollado correctamente y que el producto final sea eficiente.

Legibilidad > Corrección > Eficiencia

**Legibilidad.** Un programa Java debe ser fácil de leer y entender, incluso para una persona que no ha participado en el desarrollo del programa. La legibilidad es un aspecto muy importante porque facilita el mantenimiento del software, la corrección de errores o la modificación de la funcionalidad de la aplicación con menor coste.

**Corrección.** Un programa debe hacer lo que debe hacer, ni más, ni menos. Esto es lo que se entiende por corrección. Un programa debe cumplir rigurosamente los requisitos funcionales y técnicos de la fase de especificación. Durante la fase de prueba se verifica que el programa funciona correctamente y que cumple los requisitos técnicos y funcionales.

**Eficiencia.** La eficiencia se refiere al tiempo que un programa tarda en ejecutarse y a los recursos que consume. Cuanto más rápido sea un programa y utilice menos memoria o disco duro, el diseño es mejor. La eficiencia no es un problema que deba atender cuando se aprende a programar. Ahora lo mejor es utilizar los mecanismos de optimización propios de los compiladores. La eficiencia se debe analizar solo cuando un programa funciona correctamente y cumple con los requisitos técnicos definidos.

## El proceso de desarrollo de software

El proceso de desarrollo de los programas Java no difiere de la mayoría de los lenguajes de programación. Es necesario seguir un conjunto de pasos para desarrollar correctamente un producto software.

La Ingeniería del Software estudia los distintos procesos de desarrollo de software. El IEEE define Ingeniería del Software como la aplicación sistemática, disciplinada y cuantificable de un proceso de desarrollo, operación y mantenimiento de un producto software.

El proceso clásico de desarrollo de software es ampliamente utilizado por su sencillez. Este proceso se compone de las siguientes fases: especificación, diseño, codificación, prueba y mantenimiento.

**Especificación.** En esta fase se decide la funcionalidad, las características técnicas de una aplicación y sus condiciones de uso. En esta fase es necesario responder a las siguientes preguntas:

- ¿Para qué se va a utilizar la aplicación?
- ¿Cuáles son las necesidades funcionales de los usuarios?
- ¿Cuál es el perfil de los usuarios de la aplicación?
- ¿En qué plataforma se ejecutará la aplicación?
- ¿Cuáles son sus condiciones de operación?
- ¿Cómo se va a utilizar?

**Diseño.** En esta fase se utiliza toda la información recogida en la fase de especificación y se propone una solución que responda a las necesidades del usuario y se pueda desarrollar. En esta fase se define la arquitectura de la aplicación. Es necesario detallar la estructura y la organización del programa y cómo se relacionan las distintas partes de la aplicación.

**Codificación.** Esta fase consiste en la programación en Java de las especificaciones de diseño de la fase anterior. Durante esta fase de codificación o implementación se aplican normas de programación para facilitar la legibilidad de los programas Java.

**Prueba.** En esta fase se compila y se ejecuta la aplicación para verificar que cumple con los requisitos funcionales y técnicos definidos en la fase de especificación. Si el programa no cumple con todos los requisitos, se puede deber a errores de diseño o de programación. En tal caso, es necesario corregir los errores que se hayan detectado y repetir de nuevo el proceso de diseño y codificación. Durante la fase de prueba se verifica que la aplicación cumple con los criterios de calidad establecidos en el proyecto: facilidad de uso, corrección, eficiencia, integridad, fiabilidad, flexibilidad, facilidad de mantenimiento, facilidad de prueba, portabilidad, capacidad de reutilización e interoperabilidad.

Una vez que la aplicación se ha probado y cumple con los requisitos establecidos, entonces se pone en funcionamiento y comienza la fase de operación para que sea utilizada para el fin con el que ha sido desarrollada.

**Mantenimiento.** Una vez que la aplicación se ha puesto en operación da comienzo la fase de mantenimiento. En esta fase se corrigen errores de funcionamiento de la aplicación, se modifica la funcionalidad o se añaden las nuevas funcionalidades que demandan los usuarios. La fase de mantenimiento es la de mayor duración, pues pueden pasar muchos años desde el inicio de la operación hasta que el producto es retirado.

## 2. Estructura de un programa Java

---

Un programa describe cómo un ordenador debe interpretar las órdenes del programador para que ejecute y realice las instrucciones dadas tal como están escritas. Un programador utiliza los elementos que ofrece un lenguaje de programación para diseñar programas que resuelvan problemas concretos o realicen acciones bien definidas.

El siguiente programa Java muestra un mensaje en la consola con el texto "Hola Mundo".

```
/*
 * Este programa escribe el texto "Hola Mundo" en la consola
 * utilizando el método System.out.println()
 */

public class HolaMundo {

    public static void main (String[] args) {
        System.out.println("Hola Mundo");
    }

}
```

### La estructura de un programa Java

En este programa se pueden identificar los siguientes elementos del lenguaje Java: comentarios, definiciones de clase, definiciones de método y sentencias.

**Comentario.** El programa comienza con un comentario. El delimitador de inicio de un comentario es `/*` y el delimitador de fin de comentario es `*/`. El texto del primer comentario de este ejemplo sería: 'Este programa escribe el texto "Hola Mundo" en la consola utilizando el método `System.out.println()`'. Los comentarios son ignorados por el compilador y solo son útiles para el programador. Los comentarios ayudan a explicar aspectos relevantes de un programa y lo hacen más legible. En un comentario se puede escribir todo lo que se desee, el texto puede ser de una o más líneas.

**Definición de clase.** La primera línea del programa, después del primer comentario. Define una clase que se llama `HolaMundo`. La definición de la clase comienza por el carácter `{` y termina con el carácter `}`. El nombre de la clase lo define el programador.

**Definición de método.** Después de la definición de clase se escribe la definición del método `main()`. Todos los programas Java deben incluir un método `main()`. Este método indica las sentencias a realizar cuando se ejecuta un programa. Un método es una secuencia de sentencias ejecutables. Las sentencias de un método quedan delimitadas por los caracteres `{` y `}` que indican el inicio y el fin del método, respectivamente.

**Sentencia.** Dentro del método `main()` se incluye una sentencia para mostrar un texto por la consola. Los textos siempre se escriben entre comillas dobles para diferenciarlos de otros elementos del lenguaje. Todas las sentencias de un programa Java deben terminar con el símbolo punto y coma. Este símbolo indica al compilador que ha finalizado una sentencia.

Una vez que el programa se ha editado, es necesario compilarlo y ejecutarlo para comprobar si es correcto. Al finalizar el proceso de compilación, el compilador indica si hay errores en el código Java, dónde se encuentran y el tipo de error que ha detectado: léxico, sintáctico o semántico.

```
/* Este programa calcula el perímetro de una circunferencia */

public class PerimetroCircunferencia {
    public static void main (String[] args) {

        // declaración de PI y la variables radio y perimetro

        final double PI = 3.1415926536;
        double radio = 25.0, perimetro;

        perimetro = 2.0*PI*radio;
        System.out.print("El perimetro de la circunferencia de radio ");
        System.out.print(radio);
        System.out.print(" es ");
        System.out.print(perimetro);

    }
}
```



En un programa Java las sentencias se ejecutan secuencialmente, una detrás de otra en el orden en que están escritas.

En este ejemplo se puede ver que dentro del método `main()` se incluye un comentario de una sola línea que comienza con `//`. A continuación se declaran las variables `PI`, `radio` y `perimetro`, todas ellas de tipo `double` porque almacenan números reales. `PI` representa un valor constante, por lo que es necesario utilizar el delimitador `final` y asignarle el valor `3.1415926536` correspondiente al número pi.

Después de las declaraciones, se asigna el valor `25.0` a la variable `radio` y se calcula el perímetro. Finalmente, se muestra el resultado del cálculo del perímetro para una circunferencia de radio `25`.

En este ejemplo se utilizan variables numéricas de tipo `double`. Cada variable almacena un número real. La parte entera del número se separa de los decimales con un punto, no con una coma. Esta es una característica de Java que se debe tener en cuenta, de lo contrario, el compilador no entiende que se trata de un número real.

El valor `25.0` almacenado en la variable `radio` es una magnitud para la que no se indican sus unidades. El programador es responsable de que los cálculos se realicen correctamente y de realizar la conversión de unidades cuando sea necesario.

Para escribir un mensaje por la consola se utilizan los métodos `System.out.print()` y `System.out.println()`. Para escribir un mensaje sin saltar a la línea siguiente se utiliza `System.out.print()`, `System.out.println()` escribe un mensaje y da un salto de línea.

¿Qué hace el siguiente código Java?

```
System.out.print("Hola");  
System.out.print(" ");  
System.out.print("Mundo");
```

En este ejemplo se escribe el texto "Hola Mundo" en la consola.

```
Hola Mundo
```

¿Qué pasaría si se omitiera la segunda línea de código?,

```
System.out.print("Hola");  
System.out.print("Mundo");
```

En este caso se escribiría "HolaMundo" sin el espacio de separación entre las dos palabras.

```
HolaMundo
```

Si en vez de utilizar el método `System.out.print()` se utiliza el método `System.out.println()`, entonces el mensaje se escribe en dos líneas distintas.

```
System.out.println("Hola");  
System.out.println("Mundo");
```

En este ejemplo, se escribiría "Hola" y "Mundo" en dos líneas.

```
Hola  
Mundo
```

Combinando el uso de los métodos `System.out.print()` y `System.out.println()` se puede escribir cualquier mensaje en la consola saltando de línea cuando sea necesario. Además, ambos métodos permiten utilizar el operador `+` para concatenar textos.

El siguiente código Java escribe "Hola Mundo" en la consola.

```
System.out.print("Hola");  
System.out.print(" ");  
System.out.print("Mundo");
```

En este caso se ejecutan tres métodos `System.out.print()`. Este código se puede simplificar utilizando un solo método `System.out.print()` y el operador `+` para concatenar los textos "Hola", " " y "Mundo".

```
System.out.print("Hola" + " " + "Mundo");
```

También se puede escribir directamente el mensaje "Hola Mundo". El resultado es el mismo y el código es más claro.

```
System.out.print("Hola Mundo");
```

Para mostrar por la consola un texto seguido del valor almacenado de una variable se puede ejecutar dos veces el método `System.out.print()`:

```
System.out.print("El perímetro es ");  
System.out.print(perimetro);
```

Este código se puede simplificar utilizando el operador `+` para concatenar el texto "El perímetro es " con el valor almacenado en la variable `perimetro`.

```
System.out.print("El perímetro es " + perimetro);
```

En ambos casos el resultado es el mismo. Por ejemplo, si el valor almacenado en la variable `perimetro` fuera 157.08 la salida por la consola sería:

```
El perímetro es 157.08
```

## Los elementos de un programa Java

A continuación se describe la definición léxica y sintáctica de los elementos de un programa Java: comentarios, identificadores, variables y valores, tipos primitivos, literales, operadores, expresiones y expresiones aritmético-lógicas.

### Comentarios

En un programa Java hay tres tipos de comentarios.

**Comentario de bloque.** Empieza por `/*` y termina por `*/`. El compilador ignora todo el texto contenido dentro del comentario.

```
/*  
 * El programa HolaMundo se utiliza para aplicar los  
 * métodos System.out.print() y System.out.println()  
 */
```

**Comentario de documentación.** Empieza por `/**` y termina por `*/`. Java dispone de la herramienta `javadoc` para documentar automáticamente los programas. En un comentario de documentación normalmente se indica el autor y la versión del software.

```
/**  
 * Programa HolaMundo  
 * @author Fundamentos de Informática  
 * @version 1.0  
 * @see Referencias  
 */
```

**Comentario de línea.** Empieza con `//`. El comentario comienza con estos caracteres y termina al final de la línea.

```
// El método System.out.println() salta la línea
```

El uso de comentarios hace más claro y legible un programa. En los comentarios se debe decir qué se hace, para qué y cuál es el fin de nuestro programa. Conviene utilizar comentarios siempre que merezca la pena hacer una aclaración sobre el programa.

### Identificadores

El programador tiene libertad para elegir el nombre de las variables, los métodos y de otros elementos de un programa. Existen reglas muy estrictas

sobre los nombres que se utilizan como identificadores de clases, de variables o de métodos.

Todo identificador debe empezar con una letra que puede estar seguida de más letras o dígitos. Una letra es cualquier símbolo del alfabeto y el carácter '\_'. Un dígito es cualquier carácter entre '0' y '9'.

Los identificadores `Hola`, `hola`, `numero`, `numeroPar`, `numeroImpar`, `numero_impar`, `numero_par`, `nombre`, `apellido1` y `apellido2` son válidos. El identificador `1numero` no es válido porque empieza con un dígito, no con una letra.

Cualquier identificador que empiece con una letra seguida de más letras o dígitos es válido siempre que no forme parte de las palabras reservadas del lenguaje Java. El lenguaje Java distingue entre letras mayúsculas y minúsculas, esto significa que los identificadores `numeroPar` y `numeropar` son distintos.

Existen unas normas básicas para los identificadores que se deben respetar.

- Los nombres de variables y métodos empiezan con minúsculas. Si se trata de un nombre compuesto cada palabra debe empezar con mayúscula y no se debe utilizar el guión bajo para separar las palabras. Por ejemplo, los identificadores `calcularSueldo`, `setNombre` o `getNombre` son válidos.
- Los nombres de clases empiezan siempre con mayúsculas. En los nombres compuestos, cada palabra comienza con mayúscula y no se debe utilizar el guión bajo para separar las palabras. Por ejemplo, `HolaMundo`, `PerimetroCircunferencia`, `Alumno` o `Profesor` son nombres válidos.
- Los nombres de constantes se escriben en mayúsculas. Para nombres compuestos se utiliza el guión bajo para separar las palabras. Por ejemplo, `PI`, `MINIMO`, `MAXIMO` o `TOTAL_ELEMENTOS` son nombres válidos.

## Variables y valores

Un programa Java utiliza variables para almacenar valores, realizar cálculos, modificar los valores almacenados, mostrarlos por la consola, almacenarlos en disco, enviarlos por la red, etc. Una variable almacena un único valor.

## Estructura de un programa Java

---

Una variable se define por un nombre, un tipo y el rango de valores que puede almacenar.

El nombre de una variable permite hacer referencia a ella. Este nombre debe cumplir las reglas aplicables a los identificadores. El tipo indica el formato de los valores que puede almacenar la variable: cadenas de caracteres, valores lógicos, números enteros, números reales o tipos de datos complejos. El rango indica los valores que puede tomar la variable.

Por ejemplo, una variable de tipo número entero, con nombre `mesNacimiento` puede almacenar valores positivos y negativos, lo que no tiene sentido cuando se trata de meses del año. El rango válido para esta variable sería de 1 a 12.

Para declarar una variable en Java se indica el tipo y su nombre.

```
int mesNacimiento;
```

En este ejemplo se indica que la variable es de tipo entero (`int`) y su nombre es `mesNacimiento`. Una vez declarada una variable, se puede utilizar en cualquier parte del programa referenciándola por su nombre. Para almacenar un valor en una variable se utiliza el operador de asignación y a continuación se indica el valor, por ejemplo 2.

```
mesNacimiento = 2;
```

A partir de este momento la variable `mesNacimiento` almacena el valor 2 y cualquier referencia a ella utiliza este valor. Por ejemplo, si se imprime el valor de la variable por la consola, muestra el valor 2.

```
System.out.print(mesNacimiento);
```

Java permite declarar e inicializar una variable en una sola sentencia.

```
int diaNacimiento = 20;  
int mesNacimiento = 3;
```

En este ejemplo se declaran las variables `diaNacimiento` con el valor 20 y `mesNacimiento` con valor 3.

Si se declara una constante, entonces se debe utilizar el delimitador `final` y es necesario indicar su valor. En la siguiente declaración se define el valor de la constante `pi` de tipo `double` para almacenar un número real.

```
final double PI = 3.1415926536;
```

Es conveniente utilizar nombres apropiados para las variables. El nombre de una variable debe indicar para qué sirve. El nombre de una variable debe explicarse por sí mismo para mejorar la legibilidad del programa.

Si se desea declarar más de una variable a la vez se deben separar las variables en la sentencia de la declaración.

```
int dia=20, mes=2, año=2020;
```

En este ejemplo se declaran las variables enteras `dia`, `mes` y `año`. La variable `día` se inicializa con el valor 20, `mes` con 2 y `año` con 2020. La siguiente declaración es equivalente a la anterior.

```
int dia=20;
int mes=2;
int año=2020;
```

## Tipos primitivos

Las variables de java pueden ser de un tipo primitivo de datos o una referencia a un objeto. Los tipos primitivos permiten representar valores básicos. Estos tipos se clasifican en números enteros, números reales, caracteres y valores booleanos.

**Números enteros.** Representan números enteros positivos y negativos con distintos rangos de valores, desde cientos a trillones. Los tipos enteros de Java son `byte`, `int`, `short` y `long`.

**Números reales.** Existen dos tipos de números reales en Java, `float` y `double`. La diferencia entre ellos está en el número de decimales que tienen capacidad para expresar y en sus rangos de valores.

**Caracteres.** El tipo `char` permite representar cualquier carácter Unicode. Los caracteres Unicode contienen todos los caracteres del alfabeto de la lengua castellana.

**Booleano.** Se utiliza para representar los valores lógicos verdadero y falso. Solo tiene dos valores `true` y `false`.

La siguiente tabla resume los tipos primitivos de Java.

Tipo	Descripción	Valor mínimo y máximo
<code>byte</code>	Entero con signo	-128 a 127
<code>short</code>	Entero con signo	-32768 a 32767
<code>int</code>	Entero con signo	-2147483648 a 2147483647
<code>long</code>	Entero con signo	-922117036854775808 a +922117036854775807
<code>float</code>	Real de precisión simple	$\pm 3.40282347e+38$ a $\pm 1.40239846e-45$
<code>double</code>	Real de precisión doble	$\pm 1.7976931348623157e+309$ a $\pm 4.94065645841246544e-324$
<code>char</code>	Caracteres Unicode	<code>\u0000</code> a <code>\uFFFF</code>
<code>boolean</code>	Valores lógicos	<code>true</code> , <code>false</code>

## Literales

Se denomina literal a la manera en que se escriben los valores para cada uno de los tipos primitivos.

**Números enteros.** Un número entero en Java se puede escribir en decimal, octal o en hexadecimal. Cuando se utiliza el sistema octal es necesario poner el dígito 0 delante del número. Si se utiliza el sistema hexadecimal se debe poner 0x delante del número. Por ejemplo, el número decimal 10 se puede escribir 012 en octal y 0xA en hexadecimal. Los números enteros se supone que pertenecen al tipo `int`.



**Números reales.** Un número real en Java siempre debe tener un punto decimal o un exponente. Por ejemplo, el número 0.25 se puede expresar también como 2.5e-1. Los números reales se supone que pertenecen al tipo `double`.

**Booleanos.** Los valores lógicos solo pueden ser `true` y `false`. Se escriben siempre en minúsculas.

**Caracteres.** Los valores de tipo carácter representan un carácter Unicode. Se escriben siempre entre comillas simples, por ejemplo `'a'`, `'A'`, `'0'`, `'9'`. En Java un carácter se puede expresar por su código de la tabla Unicode en octal o en hexadecimal. Los caracteres que tienen una representación especial se indican en la siguiente tabla.

Carácter	Significado
<code>\b</code>	Retroceso
<code>\t</code>	Tabulador
<code>\n</code>	Salto de línea
<code>\r</code>	Cambio de línea
<code>\"</code>	Carácter comilla doble
<code>'</code>	Carácter comilla simple
<code>\\</code>	Carácter barra hacia atrás

**Textos.** Un texto en Java pertenece a la clase `String` y se expresa como el texto entre comillas dobles. Un texto siempre debe aparecer en una sola línea. Para dividir un texto en varias líneas se debe utilizar el operador `+` para concatenar textos.

Un texto puede estar vacío o contener uno o más caracteres. Por ejemplo, "Hola Mundo" es un texto de 10 caracteres, mientras que "" es un texto vacío y tiene 0 caracteres. El texto "a" es diferente del carácter 'a' de tipo `char`.

## Operadores

Cada tipo puede utilizar determinados operadores para realizar operaciones o cálculos.

**Números enteros.** Al realizar una operación entre dos números enteros, el resultado siempre es un número entero. Con los números enteros se pueden realizar operaciones unarias, aditivas, multiplicativas, de incremento y decremento, relacionales, de igualdad y de asignación.

- Una operación unaria permite poner un signo delante: +5, -2.
- Una operación aditiva se refiere a la suma y la resta: 2+3, 5-2.
- Una operación multiplicativa multiplica o divide dos valores: 5\*2, 5/2. El operador % calcula el resto de la división entera 5%2.
- Un incremento o decremento aumenta o decreta en 1 el valor de una variable: ++numero, numero++, --numero, numero--. Si el operador va antes de la variable, primero se realiza la operación y se modifica el valor de la variable. Si el operador va después de la variable, su valor se modifica al final.
- Un operador relacional permiten comparar dos valores: >, <, >= y <=. El resultado de la comparación es un valor booleano que indica si la relación es verdadera o falsa.
- Un operador de igualdad compara si dos valores son iguales o no. El operador == devuelve verdadero si los dos valores son iguales, el operador != devuelve verdadero si son diferentes. El resultado de la comparación es un valor booleano que indica si la igualdad o desigualdad es verdadera o falsa.
- Un operador de asignación permite asignar un valor o el resultado de una operación a una variable: =, +=, -=, \*=, /=, %=.

**Números reales.** Con los números reales se aplican los mismos operadores que con los números enteros. Si se realizan operaciones unarias, aditivas o multiplicativas, el resultado es un número real. También se pueden aplicar los operadores relacionales para comparar dos números reales.

**Booleanos.** Los operadores que se aplican a los valores lógicos son: negación, Y lógico, O lógico.

- La negación (!) devuelve true si el operando es false.
- El Y lógico (&&) devuelve false si uno de los operandos es false.
- El O lógico (||) devuelve true si uno de los operandos es true.

## Expresiones

Una expresión permite realizar operaciones entre valores utilizando distintos operadores. Las expresiones son útiles para representar las fórmulas matemáticas que se utilizan para realizar cálculos.

En Java se pueden definir expresiones tan complejas como sea necesario. Por ejemplo, para convertir una temperatura de grados Fahrenheit a Centígrados se utiliza la fórmula:

$$C = ((F - 32) * 5) / 9$$

En este ejemplo C representa la temperatura en grados centígrados y F en grados Fahrenheit. La fórmula en Java, utilizando las variables `centigrados` y `fahrenheit` de tipo `double`.

```
centigrados = ((fahrenheit - 32.0) * 5.0) / 9.0;
```

Toda la expresión se evalúa a un valor. El orden de los cálculos depende del orden de prioridad de los operadores: primero los operadores unarios, después los multiplicativos, de izquierda a derecha, después los operadores aditivos, de izquierda a derecha, después los operadores de relación y por último los operadores de asignación.

Por ejemplo, la expresión  $x = -3 + 2 * 4 - 12 / 6 + 5$  se calcula en el orden siguiente:

Primero se aplica el operador unario (-) a 3 y se obtiene -3. A continuación se evalúan los operadores multiplicativos de izquierda a derecha. Se calcula el producto de  $2 * 4$  y se obtiene 8, después se divide  $12 / 6$  y se obtiene 2. Al finalizar estas operaciones la expresión queda  $x = -3 + 8 - 2 + 5$ . Por último, se evalúan los operadores aditivos de izquierda a derecha y se obtiene 8.

Cuando se desea modificar el orden de prioridad de los operadores es necesario utilizar paréntesis para indicar el orden de evaluación. Por ejemplo, al calcular el valor de  $y = -3 + 2 * (14 - 2) / 6 + 5$  se obtiene 6.

### Expresiones aritmético-lógicas

Una expresión aritmético-lógica devuelve un valor lógico verdadero o falso. En este tipo de expresiones se utilizan operadores aritméticos, operadores relacionales y de igualdad. Por ejemplo, una expresión lógica puede ser:

$$(10 - 2) > (5 - 3)$$

En este ejemplo la expresión aritmético-lógica es verdadera porque el lado derecho de la expresión es mayor que el lado izquierdo.

En una expresión aritmético-lógica se pueden combinar varias expresiones con operadores lógicos. La precedencia de los operadores lógicos es menor que la de los operadores relacionales, por lo que primero se evalúan las desigualdades y después los operadores lógicos. El orden de prioridad de los operadores lógicos es el siguiente: primero la negación, después el Y lógico y por último el O lógico. La prioridad de los operadores de asignación es la menor de todas.

Por ejemplo, la expresión  $3 + 5 < 5 * 2 \ || \ 3 > 8 \ \&\& \ 7 > 6 - 2$  se evalúa en el orden siguiente.

Primero se evalúan las expresiones aritméticas y se obtiene la expresión lógica  $8 < 10 \ || \ 3 > 8 \ \&\& \ 7 > 4$ . A continuación se evalúan los operadores relacionales y se obtiene  $true \ || \ false \ \&\& \ true$ . Ahora se evalúa el operador Y lógico con los operandos  $false \ \&\& \ true$  y se obtiene  $false$ . Por último, se evalúa el operador O lógico con los operandos  $true \ || \ false$  y se obtiene  $true$ , el valor final de la expresión.

Los operadores lógicos  $\&\&$  y  $\||$  se evalúan por cortocircuito. Esto significa que al evaluar  $a \ \&\& \ b$ , si  $a$  es falso, no es necesario evaluar  $b$  porque la expresión es falsa. De forma similar, al evaluar  $a \ \|| \ b$ , si  $a$  es verdadero, no es necesario evaluar  $b$  porque la expresión es verdadera.

### Conversión de tipos

Muchas veces es necesario realizar conversiones de tipos cuando se evalúa una expresión aritmética. Por ejemplo, si después de realizar el cálculo de conversión de grados Fahrenheit a Centígrados se quiere almacenar el resultado en la variable de tipo entero `temperatura`, es necesario hacer

una conversión de tipos. La fórmula en Java, utilizando las variables `centigrados` y `fahrenheit` de tipo `double`.

```
centigrados = ((fahrenheit - 32.0) * 5.0) / 9.0;
```

Antes de asignar el valor resultante a la variable `temperatura`, que almacena un valor entero, es necesario convertir el valor de tipo `double` de la variable `centigrados` a `int`.

```
int temperatura = (int) centigrados;
```

## Las palabras reservadas de Java

En todos los lenguajes de programación existen palabras con un significado especial. Estas palabras son reservadas y no se pueden utilizar como nombres de variables.

<code>abstract</code>	<code>final</code>	<code>public</code>
<code>assert</code>	<code>finally</code>	<code>return</code>
<code>boolean</code>	<code>float</code>	<code>short</code>
<code>break</code>	<code>for</code>	<code>static</code>
<code>byte</code>	<code>if</code>	<code>strictfp</code>
<code>case</code>	<code>implements</code>	<code>super</code>
<code>catch</code>	<code>import</code>	<code>switch</code>
<code>char</code>	<code>instanceof</code>	<code>synchronized</code>
<code>class</code>	<code>int</code>	<code>this</code>
<code>continue</code>	<code>interface</code>	<code>throw</code>
<code>default</code>	<code>long</code>	<code>throws</code>
<code>do</code>	<code>native</code>	<code>transient</code>
<code>double</code>	<code>new</code>	<code>true</code>
<code>else</code>	<code>null</code>	<code>try</code>
<code>enum</code>	<code>package</code>	<code>void</code>
<code>extends</code>	<code>private</code>	<code>volatile</code>
<code>false</code>	<code>protected</code>	<code>while</code>

## Estructura de un programa Java

---

En realidad, las palabras `false`, `null` y `true` son literales. No son palabras reservadas del lenguaje, pero no se pueden utilizar como identificadores.

### 3. Clases y objetos

---

La programación orientada a objetos se enfoca en los elementos de un sistema, sus atributos y las interacciones que se producen entre ellos para diseñar aplicaciones informáticas. Los elementos abstractos del modelo orientado a objetos se denominan clases.

Un programa orientado a objetos es, esencialmente, una colección de objetos que se crean, interaccionan entre sí y dejan de existir cuando ya no son útiles durante la ejecución de un programa. Una aplicación informática puede llegar a ser muy compleja. La complejidad es más manejable cuando se descompone y se organiza en partes pequeñas y simples, los objetos.

Un programa Java utiliza clases y objetos. Las clases representan un esquema simplificado de la casuística de una aplicación informática. Una clase es una representación abstracta de un conjunto de objetos que comparten los mismos atributos y comportamiento, es decir, una clase describe un tipo de objetos. Un objeto es una instancia de una clase, tiene una identidad propia y un estado. La identidad de un objeto se define por su identificador. El estado de un objeto se define por el valor de sus atributos. El comportamiento de un objeto queda determinado por el comportamiento la clase a la que pertenece. Los objetos son unidades indivisibles y disponen de mecanismos de interacción llamados métodos.

Para entender el concepto de objeto es necesario saber que existe una relación directa entre los elementos que forman parte de una aplicación informática y los objetos. Normalmente, para identificar los elementos de una aplicación, debemos fijarnos en los sustantivos que utilizamos para describir los objetos reales del sistema. Para diseñar una aplicación orientada a objetos es necesario responder las siguientes preguntas:

- ¿Cuáles son los elementos tangibles de un sistema?
- ¿Cuáles son sus atributos?
- ¿Cuáles son sus responsabilidades?
- ¿Cómo se relacionan los elementos del sistema?
- ¿Qué objeto debe "saber"...?
- ¿Qué objeto debe "hacer"...?

Por ejemplo, si se desea diseñar un programa Java para gestionar las ventas de una tienda, entonces habría que identificar y describir las características de los elementos como: cliente, tipo de cliente, producto, pedido, tipo de entrega, forma de pago, unidades en existencia de los productos, etc. Los procesos de gestión de la tienda incluirían el registro de los clientes, el registro de los productos de la tienda, el proceso de compra del cliente y la realización de los pedidos, la entrada y salida de productos del almacén, etc.

Si en vez de una tienda se trata de una aplicación para dibujar figuras geométricas en dos dimensiones, entonces sería necesario identificar y describir las características de las figuras y su posición. En este caso habría figuras de tipo círculo, rectángulo, triángulo, etc. Las operaciones a realizar con las figuras incluirían dibujar, borrar, mover o rotar.

## Clases

En su forma más simple, una clase se define por la palabra reservada `class` seguida del nombre de la clase. El nombre de la clase debe empezar por mayúscula. Si el nombre es compuesto, entonces cada palabra debe empezar por mayúscula. `Circulo`, `Rectangulo`, `Triangulo` y `FiguraGeometrica` son nombres válidos de clases.

Por ejemplo, la clase `Circulo` se define con tres atributos: el radio y las coordenadas `x`, `y` que definen la posición del centro del círculo.

```
/* Esta clase define los atributos de un círculo */

public class Circulo {

    int x;
    int y;
    int radio;

}
```



Una vez que se ha declarado una clase, se pueden crear objetos a partir de ella. A la creación de un objeto se le denomina instanciación. Es por esto que se dice que un objeto es una instancia de una clase y el término instancia y objeto se utilizan indistintamente.

Para crear objetos, basta con declarar una variable de alguno de los tipos de figuras geométricas:

```
Circulo circulo1;  
Circulo circulo2;
```

Para crear el objeto y asignar un espacio de memoria es necesario realizar la instanciación con el operador `new`.

```
circulo1 = new Circulo();  
circulo2 = new Circulo();
```

Después de crear los objetos, `circulo1` y `circulo2` almacenan los valores predeterminados de la clase `Circulo`. A partir de este momento los objetos ya pueden ser referenciados por su nombre. Los nombres `circulo1` y `circulo2` son las referencias válidas para utilizar ambos objetos.

## Los elementos de una clase

Una clase describe un tipo de objetos con características comunes. Es necesario definir la información que almacena el objeto y su comportamiento.

### Atributos

La información de un objeto se almacena en atributos. Los atributos pueden ser de tipos primitivos de Java o de tipo objeto. Por ejemplo, para el catálogo de vehículos de una empresa de alquiler, es necesario conocer la matrícula del coche, su marca, modelo, color, la tarifa del alquiler y su disponibilidad.

```
public class Vehiculo {  
  
    String matricula;  
    String marca;  
    String modelo;  
    String color;  
    double tarifa;  
    boolean disponible;  
  
}
```

En este ejemplo, los atributos `matricula`, `marca`, `modelo` y `color` son cadenas de caracteres, `tarifa` es un número real y `disponible` es un valor lógico.

## Métodos y constructores

Además de definir los atributos de un objeto, es necesario definir los métodos que determinan su comportamiento. Toda clase debe definir un método especial denominado constructor para instanciar los objetos de la clase. Este método tiene el mismo nombre de la clase. Por ejemplo, para la clase `Vehiculo`, el identificador del método constructor es `Vehiculo`. El método constructor se ejecuta cada vez que se instancia un objeto de la clase. Este método se utiliza para inicializar los atributos del objeto que se instancia.

Para diferenciar entre los atributos del objeto y los identificadores de los parámetros del método constructor, se utiliza la palabra `this`. De esta forma, los parámetros del método pueden tener el mismo nombre que los atributos de la clase. Esto permite hacer una asignación como la que se muestra a continuación, donde `this.marca` se refiere al atributo del objeto y `marca` al parámetro del método.

```
    this.marca = marca;
```

En el siguiente ejemplo el método constructor inicializa los atributos matrícula, marca, modelo, color y tarifa.

```
public class Vehiculo {
    String matricula;
    String marca;
    String modelo;
    String color;
    double tarifa;
    boolean disponible;

    // el método constructor de la clase Vehiculo

    public Vehiculo(String matricula,
                    String marca,
                    String modelo,
                    String color,
                    double tarifa) {
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.tarifa = tarifa;
        this.disponible = false;
    }
}
```

La instanciación de un objeto se realiza ejecutando el método constructor de la clase.

```
Vehiculo vehiculo1 = new Vehiculo("4050 ABJ",
                                   "VW", "GTI",
                                   "Blanco",
                                   100.0);
```

En este ejemplo se instancia el objeto vehiculo1. El método constructor crea el objeto vehiculo1 con la matrícula "4050 ABJ", marca "VW", modelo "GTI", color "Blanco" y tarifa 100.0 euros.

```
Vehiculo vehiculo2 = new Vehiculo("2345 JVM",  
                                   "SEAT",  
                                   "León",  
                                   "Negro",  
                                   80.0);
```

En este ejemplo se instancia el objeto `vehiculo2`. El método constructor crea el objeto `vehiculo2` con la matrícula "2345 JVM", marca "SEAT", modelo "León", color "Negro" y tarifa 80.0 euros.

La instanciación de un objeto consiste en asignar un espacio de memoria al que se hace referencia con el nombre del objeto. Los identificadores de los objetos permiten acceder a los valores almacenados en cada objeto. En estos ejemplos, los objetos `vehiculo1` y `vehiculo2` almacenan valores diferentes y ocupan espacios de memoria distintos.

Para acceder a los atributos de los objetos de la clase `Vehiculo` se definen los métodos 'get' y 'set'. Los métodos 'get' se utilizan para consultar el estado de un objeto y los métodos 'set' para modificar su estado. En la clase `Vehiculo` es necesario definir un método 'get' para cada uno de sus atributos: `getMatricula()`, `getMarca()`, `getModelo()`, `getColor()`, `getTarifa()` y `getDisponible()`. Los métodos 'set' solo se definen para los atributos que pueden ser modificados después de que se ha creado el objeto. En este caso es necesario definir `setTarifa(double tarifa)` y `setDisponible(boolean disponible)` para modificar la tarifa del alquiler del vehículo y su disponibilidad, respectivamente.

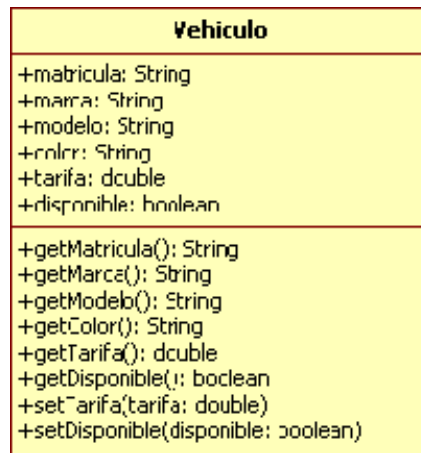
```
public class Vehiculo {
    String matricula;
    String marca;
    String modelo;
    String color;
    double tarifa;
    boolean disponible;

    // los métodos 'get' y 'set' de la clase Vehiculo

    public String getMatricula() {
        return this.matricula;
    }
    public String getMarca() {
        return this.marca;
    }
    public String getModelo() {
        return this.modelo;
    }
    public String getColor() {
        return this.color;
    }
    public double getTarifa() {
        return this.tarifa;
    }
    public boolean getDisponible() {
        return this.disponible;
    }
    public void setTarifa(double tarifa) {
        this.tarifa = tarifa;
    }
    public void setDisponible(boolean disponible) {
        this.disponible = disponible;
    }
}
```

## Representación de clases y objetos

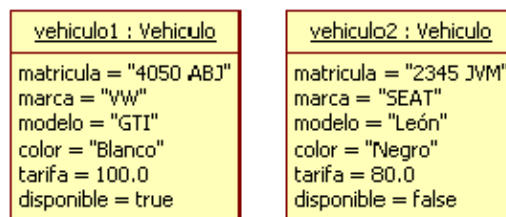
Representación de la clase vehículo utilizando un diagrama de clases.



Una clase se representa como un recuadro dividido en tres partes: el nombre de la clase en la parte superior, la declaración de atributos y la declaración de métodos.

El código Java de una clase se divide en dos partes, la declaración y su definición. La declaración comienza por la palabra `class` y a continuación se indica el nombre de la clase. La definición de una clase queda delimitada por la llave de inicio `{` y la llave de fin `}`. En el bloque de definición de la clase se declaran los atributos de los objetos y los métodos que definen su comportamiento.

Los objetos se representan como cajas que indican el nombre del objeto, la clase a la que pertenecen y el estado del objeto.



En este ejemplo, los objetos `vehiculo1` y `vehiculo2` son instancias de la clase `Vehiculo`. Ambos objetos comparten los mismos atributos, pero almacenan distintos valores. Los valores almacenados en un objeto representan su estado. El estado del objeto `vehiculo1` almacena los valores: matrícula "4050 ABJ", marca "VW", modelo "GTI", color "Blanco", tarifa 100.0 y disponible `true`. El estado del objeto `vehiculo2` almacena

matricula "2345 JVM", marca "SEAT", modelo "León", color "Negro", tarifa 80.0 y disponible false.

El estado de un objeto puede cambiar durante la ejecución de un programa Java. En este ejemplo, se podría modificar la tarifa del alquiler y la disponibilidad de los objetos de la clase Vehiculo, el resto de los atributos no se pueden modificar.

## Objetos

Un objeto se compone de atributos y métodos. Para acceder a los elementos de un objeto se escribe el nombre del objeto, un punto y el nombre del elemento al que se desea acceder.

Por ejemplo, para los objetos `vehiculo1` y `vehiculo2`, se podría acceder directamente a sus atributos como se muestra a continuación:

```
Vehiculo vehiculo1 = new Vehiculo("4050 ABJ",  
                                   "VW", "GTI",  
                                   "Blanco",  
                                   100.0);  
  
System.out.println("Matricula      : " +  
                   vehiculo1.matricula);  
  
System.out.println("Marca y modelo: " +  
                   vehiculo1.marca + " " +  
                   vehiculo1.modelo);  
  
System.out.println("Color          : " +  
                   vehiculo1.color);  
  
System.out.println("Tarifa         : " +  
                   vehiculo1.tarifa);
```

La salida por la consola:

```
Matrícula      : 4050 ABJ
Marca y modelo: VW GTI
Color          : Blanco
Tarifa         : 100.0
```

Para acceder a un método, además de su nombre hay que indicar la lista de argumentos requeridos por el método. Cuando la declaración del método no incluye parámetros no es necesario pasar argumentos.

El siguiente código Java crea el objeto `vehiculo1` y muestra su matrícula y su tarifa. A continuación modifica la tarifa a 90.0 euros y la muestra de nuevo.

```
Vehiculo vehiculo1 = new Vehiculo("4050 ABJ",
                                   "VW",
                                   "GTI",
                                   "Blanco",
                                   100.0);

System.out.println("Matricula      : " +
                  vehiculo1.getMatricula());
System.out.println("Tarifa         : " +
                  vehiculo1.getTarifa());

vehiculo1.setTarifa(90.0);

System.out.println("Matricula      : " +
                  vehiculo1.getMatricula());
System.out.println("Tarifa         : " +
                  vehiculo1.getTarifa());
```



La salida por la consola:

```
Matrícula      : 4050 ABJ
Tarifa         : 100.0
Matrícula      : 4050 ABJ
Tarifa         : 90.0
```

Para mostrar la tarifa del objeto `vehiculo1` se puede acceder directamente al atributo `tarifa` del objeto o se puede ejecutar el método `getTarifa()`. Esto se debe a que los atributos de clase `Vehiculo` son de acceso público porque se han declarado `public` en vez de `private`. Los atributos de la clase se deben declarar `private` y para acceder a ellos se debe utilizar un método 'get'.

### La referencia null

Una referencia a un objeto puede no tener asignada una instancia. Esto puede ocurrir porque se ha declarado el objeto pero no se ha instanciado, es decir no se ha creado un objeto con el operador `new`. Existe un valor especial, llamado `null` que indica que un objeto no se ha instanciado. A continuación se declara el objeto `vehiculo2`, pero no se crea una instancia para él.

```
Vehiculo vehiculo2;
```

Mientras no se instancie el objeto `vehiculo2` su referencia vale `null`. En un programa Java no se deben dejar referencias de objetos sin instanciar. Es necesario asegurarse que los objetos existen para evitar referencias `null`.

El objeto se puede instanciar en la misma declaración o más adelante, como se muestra en el siguiente ejemplo.

```
Vehiculo vehiculo2;

// el objeto vehiculo2 se declara pero no se instancia
// la instancia se crea utilizando el operador new
```

```
Vehiculo vehiculo2 = new Vehiculo("2345 JVM",
                                   "SEAT",
                                   "León",
                                   "Negro",
                                   80.0);
```

Para saber si una referencia está instanciada o no, se puede comparar con null.

```
if (vehiculo2 == null) {
    System.out.print("vehiculo2 es una referencia null")
}

if (vehiculo2 != null) {
    System.out.print("vehiculo2 está instanciado")
}
```

Si vehiculo2 es null, la primera expresión es true y la segunda false. En ese caso, el programa muestra por la consola el mensaje:

```
vehiculo2 es una referencia null
```

## Referencias compartidas por varios objetos

Un objeto puede tener varias referencias o nombres. Un alias es otro nombre que se referencia al mismo objeto. Un alias es una referencia más al mismo espacio de memoria del objeto original. Por ejemplo, si se crea el objeto vehiculo1 y después se declara otro objeto vehiculo3 y a continuación se asigna la referencia de vehiculo1 a vehiculo3, entonces vehiculo3 es un alias de vehiculo1. Esto significa que el espacio de memoria de vehiculo1 y vehiculo3 es el mismo.

```
Vehiculo vehiculo1;
Vehiculo vehiculo3;

vehiculo1 = new Vehiculo("4050 ABJ",
                        "VW",
                        "GTI",
                        "Blanco",
                        100.0);

// el objeto vehiculo1 se instancia, vehiculo3 solo
// está declarado y es una referencia null

vehiculo3 = vehiculo1;

// al asignar la referencia de vehiculo1 a vehiculo3,
// éste se convierte en alias de vehiculo1 y referencia
// el mismo espacio de memoria

System.out.println("Matricula      : " +
                  vehiculo1.getMatricula());
System.out.println("Tarifa         : " +
                  vehiculo1.getTarifa());

// se muestra la matricula y la tarifa de vehiculo1

System.out.println("Matricula      : " +
                  vehiculo3.getMatricula());
System.out.println("Tarifa         : " +
                  vehiculo3.getTarifa());
```

```
// se muestra la matricula y la tarifa de vehiculo3,  
// un alias de vehiculo1, por lo que muestra de nuevo  
// la información correspondiente a vehiculo1
```

La salida por la consola muestra dos veces los valores asignados al objeto vehiculo1.

```
Matrícula      : 4050 ABJ  
Tarifa         : 100.0  
Matrícula      : 4050 ABJ  
Tarifa         : 100.0
```

Un alias se puede utilizar para mostrar el estado de un objeto y también para modificarlo. Si se ejecuta `setTarifa(90.0)` con el objeto `vehiculo3` en realidad se modifica la tarifa de `vehiculo1`.

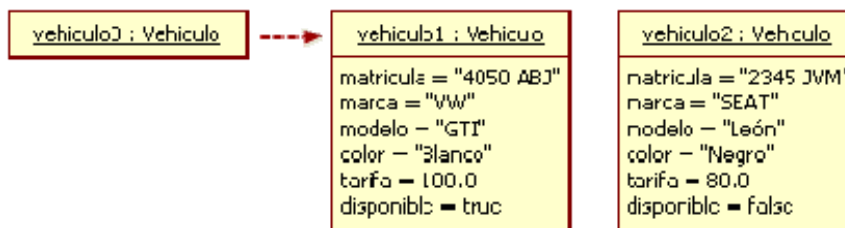
```
vehiculo3.setTarifa(90.0);  
  
// al modificar la tarifa de vehiculo3 en realidad se  
// modifica la tarifa de vehiculo1  
  
System.out.println("Matricula      : " +  
                   vehiculo1.getMatricula());  
System.out.println("Tarifa         : " +  
                   vehiculo1.getTarifa());  
  
System.out.println("Matricula      : " +  
                   vehiculo3.getMatricula());  
System.out.println("Tarifa         : " +  
                   vehiculo3.getTarifa());
```

Se muestra la matrícula y la tarifa de `vehiculo1` y `vehiculo3`.

```
Matrícula      : 4050 ABJ
Tarifa         : 90.0
Matrícula      : 4050 ABJ
Tarifa         : 90.0
```

El objeto `vehiculo3` es un alias de `vehiculo1` y no tiene un espacio de memoria propio. Utiliza el mismo espacio de memoria que `vehiculo1`. Es decir, `vehiculo1` comparte con sus alias el mismo espacio de memoria. Si se modifica el valor de la tarifa almacenada en `vehiculo3`, en realidad se modifica el valor de `vehiculo1`.

Esquemáticamente, un alias se puede ver como un objeto que apunta al espacio de memoria de otro objeto, como se muestra a continuación:



En este esquema, el objeto `vehiculo3` es un alias de `vehiculo1` y ambos objetos comparten el mismo espacio de memoria. El objeto `vehiculo2`, tiene su espacio de memoria propio y no lo comparte con otro objeto.

## El ciclo de vida de un objeto

El ciclo de vida de un objeto empieza por su declaración, su instanciación y su uso en un programa Java hasta que finalmente desaparece.

Cuando un objeto deja de ser utilizado, Java libera la memoria asignada al objeto y la reutiliza. El entorno de ejecución de Java decide cuándo puede reutilizar la memoria de un objeto que ha dejado de ser útil en un programa. El programador no debe preocuparse de liberar la memoria

utilizada por los objetos. A este proceso se le conoce como recolección de basura. Java cuenta con un sistema recolector de basura que se encarga de liberar los objetos y los espacios de memoria que ocupan cuando éstos dejan de ser utilizados en un programa.

## Atributos

Los atributos son los elementos que almacenan el estado de un objeto. Se definen de la misma forma que las variables, pero dentro del bloque de la clase.

Existen dos tipos de atributos: los atributos de clase y los atributos de objeto. Los atributos de clase existen siempre, son independientes de que existan objetos instanciados. Los atributos de clase se declaran utilizando `static`. Los atributos de objeto existen durante el ciclo de vida de un objeto, es decir, se crean cuando se instancia el objeto y se pueden utilizar mientras el objeto exista.

Un atributo se declara con la siguiente sintaxis:

```
tipo-de-acceso tipo nombre [ = valor-inicial ];
```

El tipo de acceso puede ser `private`, `protected` o `public`. Los atributos con acceso `private` solo se pueden acceder desde la propia clase que los define, mientras que los atributos `public` se pueden acceder libremente desde otras clases. Los atributos `protected` se pueden acceder desde la propia clase que los define y desde sus subclases. El concepto de subclase se explica más adelante, en el apartado de extensión de clases.

El tipo puede ser un tipo primitivo de Java o el identificador de una clase. El nombre del atributo debe cumplir las normas de los identificadores y se recomienda utilizar un sustantivo que sea representativo de la información que almacena.

La inicialización del objeto es opcional. Se puede declarar un objeto que será instanciado después o se puede instanciar al momento de su declaración.

```
public class Vehiculo {
    String matricula;
    String marca;
    String modelo;
    String color;
    double tarifa = 0.0;
    boolean disponible = false;
}
```

En el ejemplo anterior, el atributo `tarifa` se inicializa a cero y `disponible` a `false`. Al resto de atributos no se les asigna un valor inicial. Con esta declaración, cuando se instancia un objeto de tipo `Vehiculo` se inicializan los valores de los atributos `tarifa` y `disponible`. Si no se define el tipo de acceso, entonces el atributo tiene acceso de tipo `public`.

La clase `Vehiculo` se debe declarar con atributos privados. Se utiliza el tipo de acceso `private` para que solo los métodos `'get'` y `'set'` de la clase puedan acceder a ellos.

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa = 0.0;
    private boolean disponible = false;
}
```

En esta nueva declaración, todos los atributos tienen acceso `private` y solo es posible acceder a ellos desde los métodos de la clase.

## Métodos

Los métodos son funciones que determinan el comportamiento de los objetos. Un objeto se comporta de una u otra forma dependiendo de los métodos de la clase a la que pertenece. Todos los objetos de una misma clase tienen los mismos métodos y el mismo comportamiento.

Existen tres tipos de métodos: métodos de consulta, métodos modificadores y operaciones. Los métodos de consulta sirven para extraer información de los objetos, los métodos modificadores sirven para modificar el valor de los atributos del objeto y las operaciones definen el comportamiento de un objeto.

Los métodos 'get' son métodos de consulta, mientras que los métodos 'set' son métodos modificadores.

Los métodos 'get' se utilizan para extraer el valor de un atributo del objeto y los métodos 'set' para modificarlo. En la clase Vehiculo es necesario definir un método 'get' para cada uno de sus atributos: `getMatricula()`, `getMarca()`, `getModelo()`, `getColor()`, `getTarifa()` y `getDisponible()`. Los métodos 'set' solo se definen para los atributos que pueden ser modificados después de que se ha creado el objeto. En este caso es necesario definir los métodos `setTarifa(double tarifa)` y `setDisponible(boolean disponible)` para modificar la tarifa del alquiler del vehículo y su disponibilidad, respectivamente.

Un método 'get' se declara `public` y a continuación se indica el tipo que devuelve. Es un método de consulta. Por ejemplo, el método `getTarifa()` devuelve `double` porque el atributo `tarifa` es de tipo `double`. La lista de parámetros de un método 'get' queda vacía. En el cuerpo del método se utiliza `return` para devolver el valor correspondiente al atributo `tarifa` del objeto, al que se hace referencia como `this.tarifa`.

Un método 'get' se declara `public`

El valor de retorno es `double`, igual que el atributo `tarifa`

La lista de parámetros de un método 'get' queda vacía

```
public double getTarifa() {  
    return this.tarifa;  
}
```

Un método 'get' utiliza `return` para devolver el valor del atributo. En este caso el identificador del atributo es `tarifa` y se refiere a él como `this.tarifa`



El método `getAtributos()` es un caso particular de método 'get' que devuelve los valores concatenados de los atributos del objeto.

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa = 0.0;
    private boolean disponible = false;

    public String getAtributos() {
        return "Matrícula: " + this.matricula +
            " Modelo: " + this.marca + " " + this.modelo +
            " Color: " + this.color +
            " Tarifa: " + this.tarifa +
            " Disponible: " + this.disponible;
    }
}
```

Un método 'set' se declara `public` y devuelve `void`. La lista de parámetros de un método 'set' incluye el tipo y el valor a modificar. Es un método modificador. Por ejemplo, el método `setTarifa(double tarifa)` debe modificar el valor de la tarifa del alquiler almacenado en el objeto. El cuerpo de un método 'set' asigna al atributo del objeto el parámetro de la declaración.

Un método 'set' se declara `public`      El valor de retorno es `void`      La lista de parámetros de un método 'set' incluye el tipo y el nombre del parámetro

```
public void setTarifa(double tarifa) {
    this.tarifa = tarifa;
}
```

Un método 'set' modifica el valor de un atributo del objeto. En este caso el identificador del atributo es `tarifa` y se refiere a él como `this.tarifa` para asignarle el valor del parámetro

## Clases y objetos

---

Un método de tipo operación es aquel que realiza un cálculo o modifica el estado de un objeto. Este tipo de métodos pueden incluir una lista de parámetros y puede devolver un valor o no. Si el método no devuelve un valor, se declara `void`.

Por ejemplo, la clase `Circulo` define dos métodos de tipo operación, uno para calcular el perímetro y otro para calcular el área.

```
public class Circulo {
    public static final double PI = 3.1415926536;
    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }

    public double getRadio() {
        return this.radio;
    }

    public double calcularPerimetro() {
        return 2 * PI * this.radio;
    }

    public double calcularArea() {
        return PI * this.radio * this.radio;
    }
}
```

En este ejemplo, los métodos `calcularPerimetro()` y `calcularArea()` devuelven un valor `double` y ninguno de ellos recibe parámetros.

## Declaración de métodos

La declaración de un método indica si el método necesita o no argumentos. Los métodos 'get' no tienen argumentos y devuelven un valor, los métodos 'set' necesitan un argumento para indicar el valor del atributo que van a modificar.

El método `setTarifa(double tarifa)` tiene un argumento. El nombre de este parámetro es `tarifa` y su tipo es `double`.

Un método se declara con la siguiente sintaxis:

```
tipo-de-acceso tipo nombre (lista-parametros);
```

El tipo de acceso puede ser `private` o `public`. Si el método devuelve un valor se debe indicar su tipo. Este valor puede ser de un tipo primitivo de Java o el identificador de una clase. Si el método no devuelve un valor entonces el tipo es `void`. El nombre del atributo debe cumplir las normas de los identificadores y se recomienda utilizar un verbo que sea representativo de la acción que realiza el método. La lista de parámetros indica los valores que requiere el método para su ejecución.

La lista de parámetros se declara con la siguiente sintaxis:

```
tipo nombre [,tipo nombre ]
```

La lista de parámetros puede declarar una o más variables separadas por una coma. El tipo puede ser un tipo primitivo de Java o el identificador de una clase. El nombre del parámetro debe cumplir las normas de los identificadores y se recomienda utilizar un sustantivo que sea representativo de la información que almacena.

Dentro de una clase los métodos se identifican unívocamente por su nombre y su lista de parámetros.

## Invocación de métodos

Un método se puede invocar dentro o fuera de la clase donde se ha declarado. Si el método se invoca dentro de la clase, basta con indicar su nombre. Si el método se invoca fuera de la clase entonces se debe indicar el nombre del objeto y el nombre del método.

Por ejemplo, el método `getAtributos()` de la clase `Vehiculo` se podría codificar invocando a los métodos `getMatricula()`, `getMarca()`, `getModelo()`, `getTarifa()` y `getDisponible()`. En este caso, el

método `getAtributos()` utiliza los métodos 'get' de la clase en vez de hacer referencia directa a los atributos del objeto.

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa = 0.0;
    private boolean disponible = false;

    public String getAtributos() {
        return "Matrícula: " + getMatricula() + " " +
            " Modelo: " + getMarca() + " " + getModelo() +
            " Color: " + getColor() +
            " Tarifa: " + getTarifa() +
            " Disponible: " + getDisponible;
    }
}
```

Si el método `getAtributos()` se va a invocar desde fuera de la clase, entonces es necesario indicar el nombre del objeto y el nombre del método.

En este ejemplo, el método `getAtributos()` se utiliza para mostrar los valores almacenados en el objeto `vehiculo1`.

```
Vehiculo vehiculo1 = new Vehiculo("4050 ABJ",
                                    "VW",
                                    "GTI",
                                    "Blanco",
                                    100.0);

// invocación del método getAtributos para mostrar
// los datos del objeto vehiculo1

System.out.print("Datos del vehículo " +
                 vehiculo1.getAtributos());
```

Si el método es estático, entonces es necesario indicar el nombre de la clase y el nombre del método. Por ejemplo, la clase `Math` incluye el método `sqr` para calcular el cuadrado de un número. En este caso, es necesario indicar el nombre de la clase y el nombre del método que se invoca.

```
int numero = 4;

// invocación del método sqr de la clase Math

System.out.print("El cuadrado del número es: " +
                 Math.sqr(numero));
```

Cuando se invoca a un método ocurre lo siguiente:

- En la línea de código del programa donde se invoca al método se calculan los valores de los argumentos.
- Los parámetros se inicializan con los valores de los argumentos.
- Se ejecuta el bloque código del método hasta que se alcanza `return` o se llega al final del bloque.
- Si el método devuelve un valor, se sustituye la invocación por el valor devuelto.
- La ejecución del programa continúa en la siguiente instrucción donde se invocó el método.

## El método `main()`

Existe un método especial, llamado `main()`. Este método se invoca cuando se ejecuta un programa Java. Todo programa Java debe tener una clase con el método `main()`. Este método se debe declarar `public static void`. Es un método estático, público y no devuelve un valor de retorno. Los parámetros `String[] args` se refieren a la línea de comandos de la aplicación.

Cuando la máquina virtual de Java (JVM) ejecuta un programa Java invoca al método `main()`. Es este método quien a su vez ejecuta los métodos de la aplicación.

```
public class MisVehiculos {

    public static void main(String args[] ) {

        // este programa crea un objeto de la clase vehiculo y
        // muestra sus atributos

        // instanciación del objeto vehiculo1

        Vehiculo vehiculo1 = new Vehiculo("4050 ABJ",
                                           "VW",
                                           "GTI",
                                           "Blanco",
                                           100.0);

        // invocación del método getAtributos() para mostrar los datos
        // del objeto vehiculo1

        System.out.print("Datos del vehículo " +
                        vehiculo1.getAtributos());

    }
}
```

## Parámetros y argumentos

Los parámetros de un método definen la cantidad y el tipo de dato de los valores que recibe un método para su ejecución. Los argumentos son los valores que se pasan a un método durante su invocación. El método recibe los argumentos correspondientes a los parámetros con los que ha sido declarado.

Un método puede tener tantos parámetros como sea necesario. La lista de parámetros de la cabecera de un método se define con la siguiente sintaxis:

```
tipo nombre [,tipo nombre ]
```

Por ejemplo, el método constructor de la clase Vehiculo tiene cinco parámetros, la matrícula, la marca, el modelo, el color del vehículo y su

tarifa. Los parámetros matrícula, marca, modelo y color son de tipo String, tarifa es de tipo double.

```
public Vehiculo(String matricula,  
                String marca,  
                String modelo,  
                String color,  
                double tarifa) {  
  
}
```

El método `setTarifa(double tarifa)` de la clase Vehiculo tiene un parámetro tarifa de tipo double.

```
public void setTarifa(double tarifa) {  
  
}
```

Durante la invocación de un método es necesario que el número y el tipo de argumentos coincidan con el número y el tipo de parámetros declarados en la cabecera del método.

Por ejemplo, es correcto invocar al método `setTarifa(double tarifa)` del objeto `vehiculo1` pasando un argumento de tipo double.

```
vehiculo1.setTarifa(100.0); // invocación correcta  
vehiculo1.setTarifa(90.0); // invocación correcta
```

La invocación del método no es correcta si se pasan dos argumentos de tipo double o un argumento de tipo String porque la cabecera del método solo incluye un parámetro double. Las invocaciones del método `setTarifa(double tarifa)` son incorrectas.

```
vehiculo1.setTarifa(100.0, 20.0); // no es correcto  
vehiculo1.setTarifa("100.0"); // no es correcto
```

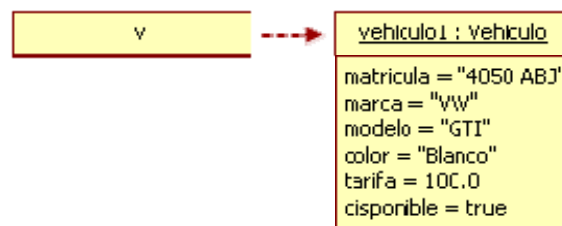
Los dos ejemplos anteriores no son válidos porque la lista de argumentos del método `setTarifa(double tarifa)` no coincide con la lista de parámetros de la declaración de este método.

Durante el proceso de compilación se comprueba que durante la invocación de un método se pasan tantos argumentos como parámetros tiene declarados y que además coinciden los tipos. Esta es una característica de los lenguajes que se denominan "strongly typed" o "fuertemente tipados".

### Paso de parámetros

Cuando se invoca un método se hace una copia de los valores de los argumentos en los parámetros. Esto quiere decir que si el método modifica el valor de un parámetro, nunca se modifica el valor original del argumento.

Cuando se pasa una referencia a un objeto se crea un nuevo alias sobre el objeto, de manera que esta nueva referencia utiliza el mismo espacio de memoria del objeto original y esto permite acceder al objeto original.



Por ejemplo, el método `recibirVehiculoAlquilado(Vehiculo v)` recibe el parámetro `v` de tipo `Vehiculo`. Si el método modifica el estado del objeto `v`, en realidad modifica el estado del objeto original `vehiculo1` que recibe como argumento.

```
public void recibirVehiculoAlquilado (Vehiculo v) {  
    v.setDisponible(true);  
}
```



En el siguiente programa Java, en el método `main()` se crea un objeto `vehiculo1` de la clase `Vehiculo`. Al instanciar el objeto, el método constructor asigna el valor `false` al atributo `disponible`, pero al invocar al método `recibirVehiculoAlquilado(Vehiculo v)` con el objeto `vehiculo1`, se modifica su disponibilidad.

```
public class MisVehiculos {

    public static void recibirVehiculoAlquilado(Vehiculo v) {
        v.setDisponible(true);
    }

    public static void main(String args[]) {

        Vehiculo vehiculo1 = new Vehiculo("4050 ABJ",
                                           "VW",
                                           "GTI",
                                           "Blanco",
                                           100.0);

        // el valor del atributo disponible de vehiculo1 es false

        System.out.println("El objeto vehiculo1 está disponible: " +
                           vehiculo1.getDisponible());

        recibirVehiculoAlquilado (vehiculo1);

        // el valor del atributo disponible de vehiculo1 es true

        System.out.println("El objeto vehiculo1 está disponible: " +
                           vehiculo1.getDisponible());

    }
}
```

La salida por la consola:

```
El objeto vehiculo1 está disponible: false
```

```
El objeto vehiculo1 está disponible: true
```

### El valor de retorno

Un método puede devolver un valor. Los métodos que no devuelven un valor se declaran `void`, mientras que los métodos que devuelven un valor indican el tipo que devuelven: `int`, `double`, `char`, `String` o un tipo de objeto.

Los métodos `'set'` devuelven `void`, mientras que los métodos `'get'` devuelven el tipo correspondiente al atributo al que hacen referencia. Los métodos `'set'` devuelven `void` porque son métodos modificadores, realizan operaciones y cálculos para modificar el estado de los objetos. Los métodos `'get'`, en cambio, son métodos de consulta y devuelven los valores almacenados en los atributos de un objeto.

Por ejemplo, el método `setTarifa(double tarifa)` recibe el parámetro `tarifa` de tipo `double` y devuelve `void`.

```
public void setTarifa(double tarifa) {
    this.tarifa = tarifa;
}
```

El método `getTarifa()` devuelve el tipo `double` correspondiente al atributo `tarifa`. El valor del atributo se devuelve con `return`.

```
public double getTarifa() {
    return this.tarifa;
}
```

### Las variables locales de un método

Las variables locales de un método son útiles para almacenar valores temporales cuyo tiempo de vida coincide con el método.

Por ejemplo, el método `getAtributos()` no utiliza variables locales. El valor de retorno se calcula al momento de hacer `return`.

```
public String getAtributos() {
    return "Matrícula: " + getMatricula() + " " +
        " Modelo: " + getMarca() + " " + getModelo() +
        " Color: " + getColor() +
        " Tarifa: " + getTarifa() +
        " Disponible: " + getDisponible;
}
```

Este método se podría codificar declarando la variable local atributos de tipo `String` para almacenar el valor de retorno. Esta variable se declara dentro del cuerpo del método.

```
public String getAtributos() {
    String atributos;

    atributos = "Matrícula: " + getMatricula() + " " +
        " Modelo: " + getMarca() + " " + getModelo() +
        " Color: " + getColor() +
        " Tarifa: " + getTarifa() +
        " Disponible: " + getDisponible;

    return atributos;
}
```

Los dos métodos son equivalentes, pero el primero es más claro porque evita el uso de una variable local que no es necesaria.

## Sobrecarga de métodos

La sobrecarga de métodos es útil para que el mismo método opere con parámetros de distinto tipo o que un mismo método reciba una lista de parámetros diferente. Esto quiere decir que puede haber dos métodos con el mismo nombre que realicen dos funciones distintas. La diferencia entre los métodos sobrecargados está en su declaración.

Por ejemplo, el método `getAtributos()` se puede sobrecargar para devolver los atributos de un vehículo y para mostrar la tarifa reducida al aplicar el porcentaje de descuento recibido como argumento.

```
public String getAtributos() {
    return "Matrícula: " + getMatricula() + " " +
        " Modelo: " + getMarca() + " " + getModelo() +
        " Color: " + getColor() +
        " Tarifa: " + getTarifa() +
        " Disponible: " + getDisponible;
}





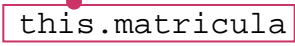

public String getAtributos(double porcentajeDescuento) {
    return "Matrícula: " + getMatricula() + " " +
        " Modelo: " + getMarca() + " " + getModelo() +
        " Color: " + getColor() + " Tarifa: " +
        (100.0 - porcentajeDescuento)/100*tarifa +
        " Disponible: " + getDisponible;
}
```

En este ejemplo los dos métodos se diferencian por la declaración de sus parámetros y ambos métodos realizan operaciones distintas.

## Constructores

Para crear un objeto se utiliza el operador `new`. Si no se ha definido un método constructor para la clase, entonces el objeto se instancia indicando el nombre de la clase y a continuación un paréntesis abierto y otro cerrado. Si ya se ha definido un método constructor, entonces no es posible instanciar un objeto utilizando el constructor por defecto. Cuando se invoca al constructor por defecto se asigna un espacio de memoria para el nuevo objeto y sus atributos se inicializan a los valores por defecto correspondientes a su tipo. Los números enteros se inicializan a cero, los números reales a 0.0, los valores lógicos a `false`, los caracteres se inicializan a `\u0000` y las referencias a `null`.

En una clase se pueden definir uno o más métodos constructores para inicializar los atributos de un objeto con valores distintos de los valores por defecto de Java. Para instanciar un objeto es necesario indicar los valores iniciales de sus atributos cuando se ejecuta el método constructor. En la clase Vehiculo se ha definido un método constructor que inicializa los atributos matricula, marca, modelo, color y tarifa.

```
public class Vehiculo {  
    String matricula;  this.matricula se refiere al  
    String marca;  atributo del objeto  
    String modelo;  
    String color;  
    double tarifa;  
    boolean disponible;  
  
    // El método constructor de la clase Vehiculo  
  
    public Vehiculo(String matricula,  matricula se refiere al  
        String marca,  parámetro del método  
        String modelo,  
        String color,  
        double tarifa) {  
         this.matricula =  matricula;  
        this.marca = marca;  
        this.modelo = modelo;  
        this.color = color;  
        this.tarifa = tarifa;  
        this.disponible = false;  
    }  
}
```

A veces es necesario contar con diferentes métodos constructores con distintos parámetros. Por ejemplo, se podría crear un objeto de la clase Vehiculo sin conocer la tarifa de alquiler. El método constructor debería inicializar la tarifa a cero.

```
public Vehiculo(String matricula,
                String marca,
                String modelo,
                String color) {
    this.matricula = matricula;
    this.marca = marca;
    this.modelo = modelo;
    this.color = color;
    this.tarifa = 0.0;
    this.disponible = false;
}
```

Cuando se definen dos o más métodos constructores para la clase Vehiculo, se dice que el método constructor de la clase está sobrecargado. En este ejemplo la diferencia entre los dos métodos es que el primero recibe cuatro parámetros e inicializa la tarifa a cero, el segundo recibe cinco parámetros, uno de ellos para inicializar la tarifa del vehículo.

```
public Vehiculo(String matricula,
                String marca,
                String modelo,
                String color)

public Vehiculo(String matricula,
                String marca,
                String modelo,
                String color,
                double tarifa)
```

La clase Vehiculo con dos métodos constructores.

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa;
    private boolean disponible;

    public Vehiculo(String matricula,
                    String marca,
                    String modelo,
                    String color) {
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.tarifa = 0.0;
        this.disponible = false;
    }
    public Vehiculo(String matricula,
                    String marca,
                    String modelo,
                    String color,
                    double tarifa) {
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.tarifa = tarifa;
        this.disponible = false;
    }
}
```

Java diferencia los métodos sobrecargados por el número y el tipo de los argumentos que tiene el método. En la clase Vehiculo el número de parámetros de los dos métodos constructores es diferente.

Cuando se invoca al método constructor de la clase con el operador new, Java selecciona el método que debe ejecutar por el número y el tipo de argumentos que recibe.

## Clases y objetos

---

Creación de objetos de la clase Vehiculo utilizando ambos métodos constructores.

```
Vehiculo vehiculo1 = new Vehiculo("4050 ABJ",  
                                   "VW",  
                                   "GTI",  
                                   "Blanco",  
                                   100.0);
```

```
Vehiculo vehiculo2 = new Vehiculo("2345 JVM",  
                                   "SEAT",  
                                   "León",  
                                   "Negro");
```

El objeto vehiculo1 se instancia ejecutando constructor de cinco parámetros, mientras que vehiculo2 se instancia ejecutando el constructor de cuatro parámetros.



## 4. Extensión de clases

---

### Composición

La composición consiste en crear una clase nueva agrupando objetos de clases que ya existen. Una composición agrupa uno o más objetos para construir una clase, de manera que las instancias de esta nueva clase contienen uno o más objetos de otras clases. Normalmente los objetos contenidos se declaran con acceso `private` y se inicializan en el constructor de la clase.

La clase `Vehiculo` está compuesta de objetos de tipo `String`. Los atributos `matricula`, `marca`, `modelo` y `color` en realidad son objetos de la clase `String`.

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa;
    private boolean disponible;

    // se omiten los métodos 'get' y 'set' de la clase

    public Vehiculo(String matricula,
                    String marca,
                    String modelo,
                    String color,
                    double tarifa) {
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.tarifa = tarifa;
        this.disponible = false;
    }
}
```

## Extensión de clases

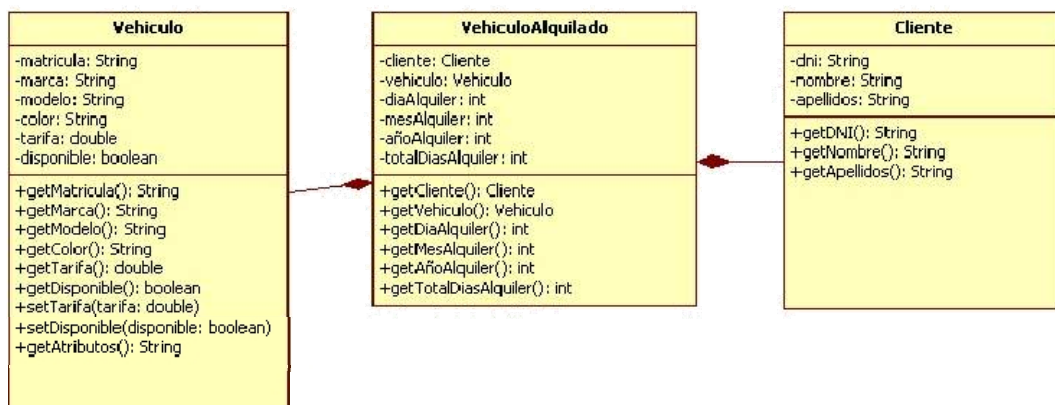
Para hacer una composición utilizando objetos de una clase diferente de String, lo primero es definir una nueva clase. La clase Cliente formará junto con Vehiculo la clase VehiculoAlquilado utilizando la composición.

```
public class Cliente {
    private String nif;
    private String nombre;
    private String apellidos;

    // se omiten los métodos 'get' y 'set' de la clase

    public Cliente(String nif, String nombre, String apellidos) {
        this.nif = nif;
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

Ahora se define una composición que declara un objeto de la clase Vehiculo y un objeto de la clase Cliente. La nueva clase VehiculoAlquilado relaciona una instancia de la clase Vehiculo con una instancia de la clase Cliente y crea objetos que almacenan relaciones entre clientes y vehículos de alquiler.



Esto significa que para instanciar un objeto de la clase VehiculoAlquilado es necesario tener referencias a objetos de las clases Cliente y Vehiculo.

```
public class VehiculoAlquilado {
    private Cliente cliente;
    private Vehiculo vehiculo;
    private int diaAlquiler;
    private int mesAlquiler;
    private int añoAlquiler;
    private int totalDiasAlquiler;

    public VehiculoAlquilado(Cliente cliente,
                              Vehiculo vehiculo,
                              int diaAlquiler,
                              int mesAlquiler,
                              int añoAlquiler,
                              int totalDiasAlquiler) {
        this.cliente = cliente;
        this.vehiculo = vehiculo;
        this.diaAlquiler = diaAlquiler;
        this.mesAlquiler = mesAlquiler;
        this.añoAlquiler = añoAlquiler;
        this.totalDiasAlquiler = totalDiasAlquiler;
    }

    // los métodos 'get' de los atributos de tipo objeto
    // Cliente y Vehiculo

    public Cliente getCliente() {
        return this.cliente;
    }

    public Vehiculo getVehiculo() {
        return this.vehiculo;
    }
}
```

La clase VehiculoAlquilado contiene un objeto de la clase Cliente, un objeto de la clase Vehiculo y atributos de tipo int para almacenar el día, el mes y el año de la fecha del alquiler del vehículo y el total de días de alquiler. La clase contenedora es VehiculoAlquilado y las clases contenidas son Cliente y Vehiculo.

## Extensión de clases

---

El programa principal donde se crean los objetos de las clases Cliente, Vehiculo y VehiculoAlquilado.

```
public class MisVehiculos {

    public static void main(String args[]) {

        // se crean dos instancias de la clase Vehiculo

        Vehiculo vehiculo1 = new Vehiculo("4050 ABJ",
                                           "VW",
                                           "GTI",
                                           "Blanco",
                                           100.0);

        Vehiculo vehiculo2 = new Vehiculo("2345 JVM",
                                           "SEAT",
                                           "León",
                                           "Negro",
                                           80.0);

        // se crea una instancia de la clase Cliente

        Cliente cliente1 = new Cliente("30435624X", "Juan", "Pérez");

        // se crea una instancia de la clase VehiculoAlquilado que
        // relaciona al cliente1 con el vehiculo1, el vehículo se
        // alquila con fecha 11/11/2011 durante 2 días

        VehiculoAlquilado alquiler1 = new VehiculoAlquilado(cliente1,
                                                              vehiculo1,
                                                              11,
                                                              11,
                                                              2011,
                                                              2);

    }
}
```

En una relación de composición, hay atributos de la clase contenedora que son objetos que pertenecen a la clase contenida. Un objeto de la clase contenedora puede acceder a los métodos públicos de las clases contenidas. En la declaración de la clase `VehiculoAlquilado` se han definido dos métodos 'get' para los atributos de tipo objeto. El método `getCliente()` devuelve un objeto de tipo `Cliente` y el método `getVehiculo()` devuelve un objeto de tipo `Vehiculo`.

Por ejemplo, el objeto `alquiler1` de la clase `VehiculoAlquilado` puede acceder a los métodos públicos de su propia clase y de las clases `Cliente` y `Vehiculo`. Un objeto de la clase `VehiculoAlquilado` puede ejecutar métodos 'get' para mostrar la información de los objetos que contiene.

```
alquiler1.getCliente().getNIF();
alquiler1.getVehiculo().getMatricula();
```

Los datos del cliente y del vehículo alquilado:

```
System.out.println("Vehículo alquilado");
System.out.println("Cliente : " +
    alquiler1.getCliente().getNIF() + " " +
    alquiler1.getCliente().getNombre() + " " +
    alquiler1.getCliente().getApellidos());
System.out.println("Vehículo: " +
    alquiler1.getVehiculo().getMatricula());
```

La salida por la consola:

```
Vehículo alquilado
Cliente : 30435624X Juan Pérez
Vehículo: 4050 ABJ
```

# Herencia

La herencia es la capacidad que tienen los lenguajes orientados a objetos para extender clases. Esto produce una nueva clase que hereda el comportamiento y los atributos de la clase que ha sido extendida. La clase original se denomina clase base o superclase, la nueva clase se denomina clase derivada o subclase.

## Extensión de clases

La capacidad para extender clases se llama herencia porque la nueva clase hereda todos los atributos y los métodos de la superclase a la que extiende. Una subclase es una especialización de la superclase. Normalmente una subclase añade nuevos atributos y métodos que le dan un comportamiento diferente al de la superclase. La herencia es un mecanismo muy importante porque permite la reutilización del código.

Suponga que se desea diseñar una aplicación para gestionar una empresa de alquiler de vehículos de tipo turismo, deportivo y furgonetas. La clase `Vehiculo` define los atributos y los métodos de todos los vehículos de la empresa de alquiler. Esto no es suficiente porque hay distintos tipos de vehículos, de manera que es necesario definir subclases para cada tipo de vehículo: turismo, deportivo y furgoneta. En este ejemplo, la superclase es `Vehiculo` y las subclases son `Turismo`, `Deportivo` y `Furgoneta`. Todas las subclases son vehículos, un turismo, un deportivo y una furgoneta son vehículos, pero cada uno de ellos tiene características propias que le hacen diferente del resto. Para un turismo interesa saber el número de puertas y el tipo de cambio de marcha, para un deportivo interesa saber su cilindrada y para una furgoneta su capacidad de carga en kilos y el volumen en metros cúbicos.

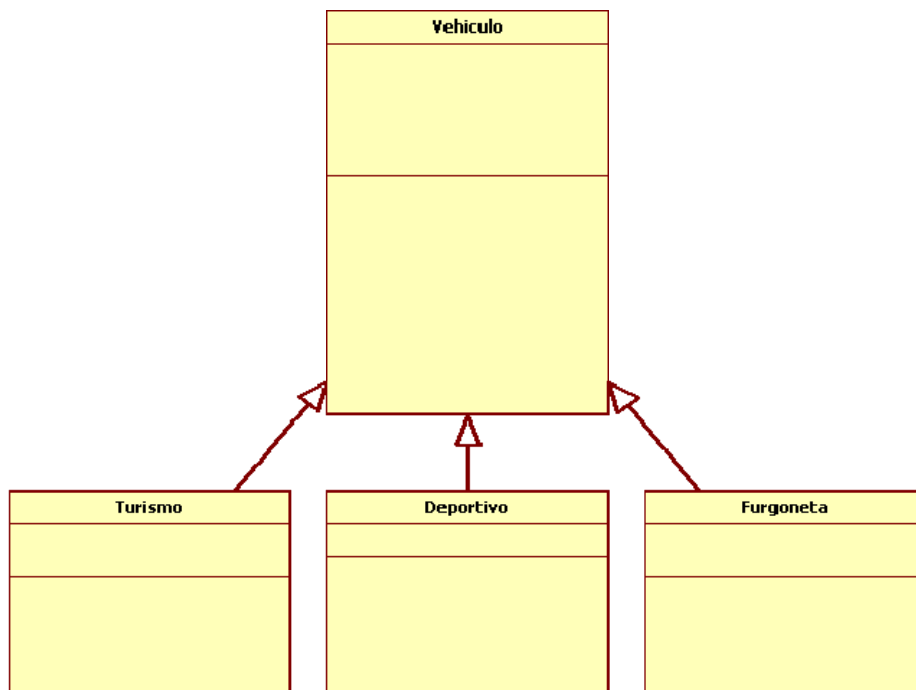
La extensión de una clase tiene la siguiente sintaxis:

```
public class nombre-subclase extends nombre-superclase {  
    }  
}
```

La declaración de la clase Turismo como subclase de Vehiculo:

```
public class Turismo extends Vehiculo {  
}
```

El esquema muestra la relación de herencia que existe entre la superclase Vehiculo y las subclases Turismo, Deportivo y Furgoneta.



Las subclases Turismo, Deportivo y Furgoneta son especializaciones de la clase Vehiculo. En una relación de herencia, las subclases heredan los atributos y los métodos de la superclase. En la declaración de las subclases se indica la clase a la que extienden, en este caso, Vehiculo.

```
public class Turismo extends Vehiculo {  
}  
public class Deportivo extends Vehiculo {  
}  
public class Furgoneta extends Vehiculo {  
}
```

La declaración de la superclase Vehiculo.

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa = 0.0;
    private boolean disponible;

    // se omiten los métodos 'get' y 'set' de la superclase Vehiculo,
    // excepto el método getAtributos()

    public Vehiculo(String matricula,
                    String marca,
                    String modelo,
                    String color,
                    double tarifa) {
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.tarifa = tarifa;
        this.disponible = false;
    }

    public String getAtributos() {
        return "Matrícula: " + this.matricula +
            " Modelo: " + this.marca + " " + this.modelo +
            " Color: " + this.color +
            " Tarifa: " + this.tarifa +
            " Disponible: " + this.disponible;
    }
}
```



La declaración de la subclase Turismo.

```
public class Turismo extends Vehiculo {
    private int puertas;
    private boolean marchaAutomatica;

    public Turismo(String matricula,
                   String marca,
                   String modelo,
                   String color,
                   double tarifa,
                   int puertas,
                   boolean marchaAutomatica) {
        super(matricula, marca, modelo, color, tarifa);
        this.puertas = puertas;
        this.marchaAutomatica = marchaAutomatica;
    }

    // métodos 'get' de la subclase Turismo

    public int getPuertas() {
        return this.puertas;
    }

    public boolean getMarchaAutomatica() {
        return this.marchaAutomatica;
    }

    public String getAtributos() {
        return super.getAtributos() +
            " Puertas: " + this.puertas +
            " Marcha automática: " + this.marchaAutomatica;
    }
}
```

La declaración de la subclase Deportivo.

```
public class Deportivo extends Vehiculo {
    private int cilindrada;

    public Deportivo(String matricula,
                     String marca,
                     String modelo,
                     String color,
                     double tarifa,
                     int cilindrada) {
        super(matricula, marca, modelo, color, tarifa);
        this.cilindrada = cilindrada;
    }

    // métodos 'get' de la subclase Deportivo

    public int getCilindrada() {
        return this.cilindrada;
    }

    public String getAtributos() {
        return super.getAtributos() +
            " Cilindrada (cm3): " + this.cilindrada;
    }
}
```

La declaración de la subclase Furgoneta.

```
public class Furgoneta extends Vehiculo {
    private int carga;
    private int volumen;

    public Furgoneta(String matricula,
                     String marca,
                     String modelo,
                     String color,
                     double tarifa,
                     int carga,
                     int volumen) {
        super(matricula, marca, modelo, color, tarifa);
        this.carga = carga;
        this.volumen = volumen;
    }

    // métodos 'get' de la subclase Furgoneta

    public int getCarga() {
        return this.carga;
    }

    public int getVolumen() {
        return this.volumen;
    }

    public String getAtributos() {
        return super.getAtributos() +
            " Carga (kg): " + this.carga +
            " Volumen (m3): " + this.volumen;
    }
}
```

### Polimorfismo

Las clases Turismo, Deportivo y Furgoneta extienden a la clase Vehiculo. Estas clases heredan los atributos de Vehiculo y cada subclase añade atributos y métodos propios.

La clase Turismo añade los atributos puertas, marchaAutomatica y los métodos getPuertas() y getMarchaAutomatica().

La clase Deportivo añade el atributo cilindrada y el método getCilindrada().

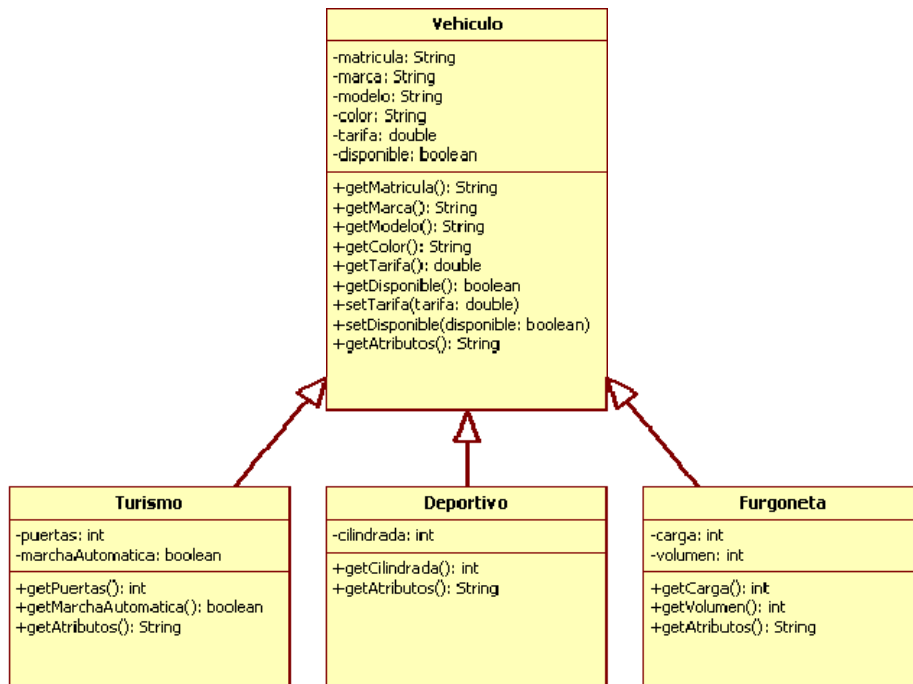
La clase Furgoneta añade los atributos carga, volumen y los métodos getCarga() y getVolumen().

Además, cada subclase declara un método getAtributos(). Este método también se ha declarado en la superclase. Esto significa que el método getAtributos() de las subclases sobrescribe al método de la superclase. Dependiendo del tipo de objeto que invoque el método, se ejecuta el método correspondiente a la clase del objeto. Por ejemplo, si el método es invocado por un objeto de la clase Turismo, entonces se ejecuta el código del método getAtributos() de la clase Turismo.

Los métodos getAtributos() de las subclases modifican el comportamiento del método getAtributos() de la superclase. En cada método se invoca a super.getAtributos() para que muestre los atributos de un vehículo y después se muestran los atributos propios de la subclase. Los métodos getAtributos() de las subclases sobrescriben el método getAtributos() de la superclase. Esta característica de los lenguajes de programación orientados a objetos se conoce como polimorfismo.

Un objeto de las subclases Turismo, Deportivo o Furgoneta puede invocar los métodos getMatricula(), getMarca(), getModelo(), getColor(), getTarifa(), getDisponible(), setTarifa() y setDisponible() de la superclase Vehiculo.

El esquema muestra la superclase Vehiculo y las subclases Turismo, Deportivo y Furgoneta con sus atributos y métodos.



El siguiente ejemplo muestra el comportamiento de los métodos sobrescritos en las subclases.

```
// creación de instancias de la superclase Vehiculo y
// de las subclases Turismo, Deportivo y Furgoneta
```

```
Vehiculo miVehiculo = new Vehiculo("4050 ABJ",
                                     "VW", "GTI",
                                     "Blanco",
                                     100.0);
```

```
Turismo miTurismo = new Turismo("4060 TUR",
                                  "Skoda", "Fabia",
                                  "Blanco",
                                  90.0,
                                  2,
                                  false);
```

```
Deportivo miDeportivo = new Deportivo("4070 DEP",
                                       "Ford", "Mustang",
                                       "Rojo",
                                       150.0,
                                       2000);
```

```
Furgoneta miFurgoneta = new Furgoneta("4080 FUR",
                                       "Fiat", "Ducato",
                                       "Azul",
                                       80.0,
                                       1200,
                                       8);
```

```
// invocación del método getAtributos() de cada objeto
```

```
System.out.print("Vehículo : " +
                 miVehiculo.getAtributos());
```

```
// miVehiculo es una instancia de la clase Vehiculo, se
```

```
// invoca el método getAtributos() de Vehiculo
```

```
Vehículo : Matrícula: 4050 ABJ Modelo: VW GTI
Color: Blanco Tarifa: 100.0 Disponible: false
```

```
System.out.print("Turismo " + miTurismo.getAtributos());
```

```
// miTurismo es una instancia de la clase Turismo, se
```

```
// invoca el método getAtributos() de Turismo
```

```
Turismo Matrícula: 4060 TUR Modelo: Skoda Fabia
Color: Blanco Tarifa: 90.0 Disponible: false Puertas: 2
Marcha automática: false
```

```
System.out.print("Deportivo " +
                 miDeportivo.getAtributos());
```

```
// miDeportivo es una instancia de la clase Deportivo,
// se invoca el método getAtributos() de Deportivo
```

```
Deportivo Matrícula: 4070 DEP Modelo: Ford Mustang
Color: Rojo Tarifa: 150.0 Disponible: false
Cilindrada (cm3): 2000
```

```
System.out.print("Furgoneta " +
                 miFurgoneta.getAtributos());
```

```
// miFurgoneta es una instancia de la clase Furgoneta,
// se invoca el método getAtributos() de Furgoneta
```

```
Furgoneta Matrícula: 4080 FUR Modelo: Fiat Ducato
Color: Azul Tarifa: 80.0 Disponible: false
Carga (kg): 1200 Volumen (m3): 8
```

```
// el objeto miTurismo pertenece a la subclase Turismo,
// es un vehículo y puede invocar a los métodos de la
// superclase Vehiculo: getMatricula(),
// getMarca() y getModelo()
```

```
System.out.print("Turismo  : " +
                 miTurismo.getMatricula() + " " +
                 miTurismo.getMarca() + " " +
                 miTurismo.getModelo());
```

```
Turismo  : 4060 TUR Skoda Fabia
```

## Compatibilidad de tipos

En una relación de tipo herencia, un objeto de la superclase puede almacenar un objeto de cualquiera de sus subclases. Por ejemplo, un objeto de la clase `Vehiculo` puede almacenar un objeto de la clase `Turismo`, `Deportivo` o `Furgoneta`. Dicho de otro modo, cualquier referencia de la clase `Vehiculo` puede contener una instancia de la clase `Vehiculo` o bien una instancia de las subclases `Turismo`, `Deportivo` o `Furgoneta`.

Esto significa que la clase base o superclase es compatible con los tipos que derivan de ella, pero no al revés. Una referencia de la clase `Turismo` solo puede almacenar una instancia de `Turismo`, nunca una instancia de la superclase `Vehiculo`.

## Conversión ascendente de tipos

Cuando un objeto se asigna a una referencia distinta de la clase a la que pertenece, se hace una conversión de tipos. Java permite asignar un objeto a una referencia de la clase base.

Por ejemplo, si un objeto de la clase `Turismo` se asigna a una referencia de la clase `Vehiculo`, se hace una conversión ascendente de tipos, denominada "upcasting". La conversión ascendente de tipos siempre se puede realizar.



```
Vehiculo miVehiculo = new Turismo("4090 TUR",
                                   "Skoda", "Fabia",
                                   "Negro",
                                   90.0,
                                   2,
                                   true);

System.out.println("Vehículo " +
                  miVehiculo.getAtributos());
```

En este ejemplo se crea un objeto de la clase base Vehiculo utilizando el constructor de la clase derivada Turismo.

Dado que la instancia es de tipo Turismo, al invocar al método getAtributos() muestra los atributos de un turismo.

```
Vehículo Matrícula: 4090 TUR Modelo: Skoda Fabia Color:
Negro Tarifa: 90.0 Disponible: false Puertas: 2
Marcha automática: true
```

A la referencia miVehiculo también se le puede asignar la referencia de una instancia existente de la clase Turismo.

```
Turismo miTurismo = new Turismo("4100 TUR",
                                 "VW", "Polo",
                                 "Rojo",
                                 80.0,
                                 2,
                                 false);
```

```
Vehiculo miVehiculo = miTurismo;  
  
System.out.println("Vehículo " +  
                    miVehiculo.getAtributos());
```

De nuevo, el método `getAtributos()` muestra los atributos de un turismo:

```
Vehículo Matrícula: 4100 TUR Modelo: VW Polo Color:  
Rojo Tarifa: 80.0 Disponible: false Puertas: 2 Marcha  
automática: false
```

## Conversión descendente de tipos

Si una instancia de la clase base `Vehiculo` almacena una referencia a un objeto de una de sus clases derivadas, entonces es posible hacer una conversión descendente de tipos, denominada "downcasting".

El objeto `miVehiculo` de la clase base `Vehiculo` almacena una referencia a un objeto de la clase derivada `Turismo`. En este caso, está permitido hacer una conversión descendente de tipos. La conversión se debe hacer de forma explícita, indicando el nombre de la clase a la que se desea convertir.

Conversión descendente de tipos:

```
Vehiculo miVehiculo = new Turismo("4090 TUR",  
                                   "Skoda", "Fabia",  
                                   "Negro",  
                                   90.0,  
                                   2,  
                                   true);  
  
Turismo miNuevoTurismo = (Turismo) miVehiculo;
```

En este ejemplo, el objeto de la clase `Vehiculo` almacena un objeto de la clase derivada `Turismo`. El objeto `miVehiculo` se convierte de forma explícita a un objeto de tipo `Turismo` utilizando el "casting" (`Turismo`). Solo así es posible realizar la asignación a una referencia que ha sido declarada de tipo `Turismo`.

Si no se utiliza el "casting", entonces el compilador de Java da un mensaje de error que indica que se produce un conflicto de tipos y no puede convertir automáticamente una referencia `Vehiculo` en una referencia `Turismo`.

Es importante señalar que el "downcasting" no siempre es legal y puede producir un error durante la ejecución del programa Java.

### Jerarquía de herencia

Cualquier clase Java puede ser utilizada como una clase base para extender sus atributos y comportamiento. La clase derivada que se obtenga, puede a su vez, ser extendida de nuevo. La relación de herencia es transitiva y define una jerarquía.

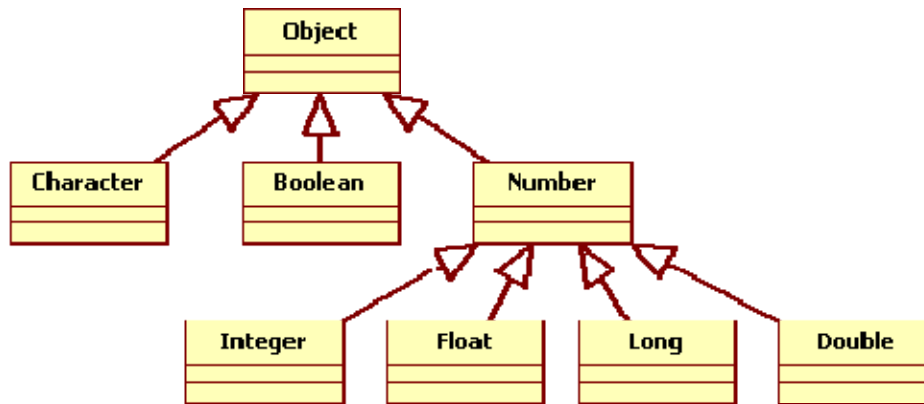
En Java todas las clases están relacionadas en una única jerarquía de herencia puesto que toda clase hereda explícitamente de otra o bien implícitamente de `Object`.

La clase `Vehiculo` no extiende explícitamente otra clase, por lo que se puede decir que es una extensión de la clase `Object` de Java. Esto quiere decir que cualquier objeto de un programa Java se puede ver como una instancia de la clase `Object`.

## Extensión de clases

---

El esquema muestra la jerarquía de herencia de las clases asociadas a los tipos primitivos de Java.



## 5. Ampliación de clases

---

### Elementos de clase (Static)

Los atributos y métodos de una clase precedidos con la palabra `static` se denominan elementos de clase. Solo existe un elemento estático para todos los objetos de una misma clase. Esto significa que los elementos de clase son compartidos por todas las instancias de la clase. Cuando se modifica un elemento de clase todas las instancias de la clase ven dicha modificación. Los atributos de clase deben tener un valor inicial aunque no exista ninguna instancia de la clase. Si el elemento de clase es un valor constante, entonces se debe indicar la palabra `final`.

Por ejemplo, se puede definir la constante `PI` para calcular el perímetro y el área de la clase `Circulo`.

```
public class Circulo {
    public static final double PI = 3.1415926536;
    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }

    public double getRadio() {
        return this.radio;
    }

    public double calcularPerimetro() {
        return 2 * PI * this.radio;
    }

    public double calcularArea() {
        return PI * this.radio * this.radio;
    }
}
```

## Ampliación de clases

---

El acceso al elemento estático PI, la instanciación del objeto miCirculo y las invocaciones a los métodos calcularPerimetro() y calcularArea().

```
// Este programa calcula el perímetro y el área de una circunferencia

public class PerimetroAreaCircunferencia {

    public static void main (String[] args) {

        System.out.println("El valor de PI es " + Circulo.PI);

        Circulo miCirculo = new Circulo(10.0);

        System.out.println("El radio del circulo es " +
            miCirculo.getRadio() +
            " su perimetro es " +
            miCirculo.calcularPerimetro() +
            " y su área es " +
            miCirculo.calcularArea());

    }
}
```

## Derechos de acceso

El estado de un objeto está dado por el conjunto de valores de sus atributos. Una modificación arbitraria, intencionada o no, puede provocar inconsistencias o comportamientos no deseados de un objeto. Es por este motivo que se debe controlar el acceso a los atributos de los objetos. Java proporciona mecanismos de acceso a los elementos de una clase, de forma que se puede determinar el derecho de acceso de cada elemento según las necesidades de los objetos.

**Acceso privado.** Los elementos privados solo se pueden utilizar dentro de la clase que los define. Para indicar el acceso privado se utiliza `private`.

**Acceso de paquete.** El acceso a estos componentes es libre dentro del paquete en el que se define la clase. El acceso de paquete no se indica expresamente.

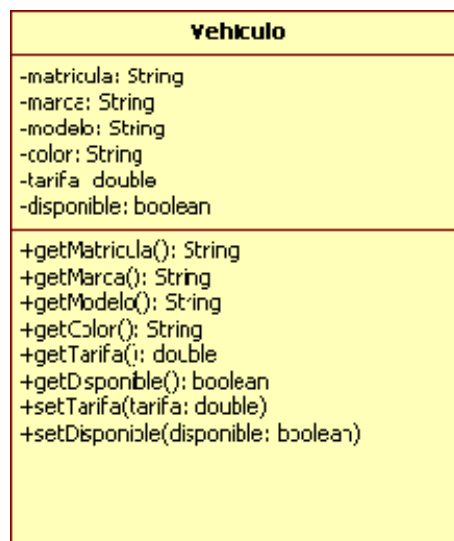
**Acceso protegido.** Los elementos protegidos solo se pueden utilizar dentro de la clase que los define, aquellas clases que la extiendan y cualquier clase definida en el mismo paquete. Para indicar el acceso protegido se utiliza `protected`.

**Acceso público.** Los elementos públicos se pueden utilizar libremente. Para indicar expresamente el acceso público se utiliza `public`. No es necesario, el acceso público se utiliza como valor por defecto mientras no se indique `private` o `protected`.

Para limitar el acceso a los atributos de la clase `Vehiculo` se utiliza `private`. Al utilizar este tipo de acceso, solo los métodos 'get' y 'set' de la clase pueden acceder a ellos.

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa;
    private boolean disponible;
}
```

Con esta declaración, todos los atributos de la clase tienen acceso `private` y el diagrama de clases muestra un signo menos delante del identificador del atributo para indicar que es privado.



La clase Vehiculo con sus métodos 'get y 'set'.

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa;
    private boolean disponible;

    public String getMatricula() {
        return this.matricula;
    }
    public String getMarca() {
        return this.marca;
    }
    public String getModelo() {
        return this.modelo;
    }
    public String getColor() {
        return this.color;
    }
    public double getTarifa() {
        return this.tarifa;
    }
    public boolean getDisponible() {
        return this.disponible;
    }
    public void setTarifa(double tarifa) {
        this.tarifa = tarifa;
    }
    public void setDisponible(boolean disponible) {
        this.disponible = disponible;
    }
}
```

La clase Vehiculo define métodos 'get' para los atributos matrícula, marca, modelo, color, tarifa y disponible. Los métodos 'set' solo son aplicables a los atributos tarifa y disponible porque se considera que el resto de atributos de la clase no pueden modificar su valor una vez que se ha creado el objeto.



La responsabilidad de modificar los atributos de los objetos es de los métodos 'set'. Estos métodos deben verificar que el valor que se desea asignar a un atributo es válido y cumple con las restricciones de diseño de la clase.

## Paquetes

Los paquetes son grupos de clases, interfaces y otros paquetes que están relacionados entre sí. Los paquetes aportan una forma de encapsulación de un nivel superior al de las clases. Permiten unificar un conjunto de clases e interfaces que se relacionan funcionalmente. Por ejemplo, el paquete `java` engloba un conjunto de paquetes con utilidades de soporte para desarrollo y ejecución de aplicaciones como `util` o `lang`.

Un paquete se declara con la siguiente sintaxis:

```
package nombre-del-paquete;
```

Por ejemplo, se podría definir el paquete `vehiculos` para la aplicación de la empresa de alquiler de vehículos:

```
package vehiculos;
```

## Uso

Para utilizar componentes que están en otro paquete diferente se debe añadir una declaración de importación.

El uso de un paquete se declara con la siguiente sintaxis:

```
import nombre-del-paquete;
```

Se puede importar un paquete entero o un componente del paquete. Por ejemplo, si se desea importar las librerías para cálculos matemáticos de Java.

```
import java.math.*;
```

## Ampliación de clases

---

Si solo se desea importar una librería, entonces se debe indicar el nombre del paquete y del componente. En este ejemplo se importa el componente Calendar de la librería de utilidades de Java.

```
import java.util.Calendar;
```

La declaración de importación se incluye antes de la declaración de la clase. En el siguiente ejemplo se incluye el componente Calendar de util y se utiliza el método `getInstance()` para obtener el día, el mes y el año de la fecha actual.

```
import java.util.Calendar;

public class CalcularFechaHoy {

    public static void main (String[] args) {
        int edad, diaHoy, mesHoy, añoHoy;

        diaHoy = Calendar.getInstance().get(Calendar.DAY_OF_MONTH);
        mesHoy = Calendar.getInstance().get(Calendar.MONTH) + 1;
        añoHoy = Calendar.getInstance().get(Calendar.YEAR);

        System.out.println("La fecha de hoy es " +
            diaHoy + "/" +
            mesHoy + "/" +
            añoHoy);

    }
}
```

## Nombres

El nombre de un paquete debe ser representativo de su contenido. El nombre puede contener la declaración de subpaquete. Se puede incluir el nombre de la empresa que ha desarrollado el paquete para facilitar su identificación.

```
package nombre-de-la-empresa.nombre-del-paquete;
```

Por ejemplo, el paquete `vehiculos` de la empresa "Mi Empresa" se podría identificar:

```
package miEmpresa.vehiculos;
```

## Clases predefinidas

Una característica importante de Java es que aporta gran cantidad de clases predefinidas. Estas clases están especializadas en comunicaciones, web, interfaz de usuario, matemáticas y muchas otras aplicaciones.

A continuación se describen las clases asociadas a los tipos primitivos de Java, la clase `Math` y la clase `String`.

## Las clases asociadas a los tipos primitivos

Los tipos predefinidos `boolean`, `char`, `int`, `long`, `float` y `double` son tipos simples, no son clases. Para facilitar la programación en Java se han creado clases asociadas a los tipos predefinidos. Estas clases proporcionan métodos útiles para convertir cadenas de texto a otros tipos, para imprimir los números con diversos formatos y para describir los tipos simples.

Estas clases generan automáticamente una instancia cuando se usan tipos simples en contextos en los que se espera un objeto. Además, pueden utilizarse en expresiones en donde se espera un tipo simple.

Las clases asociadas a los tipos primitivos son:

Clase	Tipo primitivo asociado
<code>Boolean</code>	<code>boolean</code>
<code>Character</code>	<code>char</code>
<code>Integer</code>	<code>int</code>
<code>Long</code>	<code>long</code>
<code>Float</code>	<code>float</code>
<code>Double</code>	<code>double</code>

Estas clases tienen los siguientes métodos:

- Método constructor a partir de un valor de tipo simple

```
Character letra = new Character('A');  
Integer numero = new Integer(10);
```

- Método constructor que recibe una cadena de texto y la traduce al tipo simple

```
Integer numero = new Integer("120");
```

- Método `toString()` que transforma el valor almacenado en una cadena

```
Integer numero = new Integer("100");  
System.out.println(numero.toString());
```

- Método `equals()` para comparar el valor almacenado

```
Integer numero1 = new Integer("100");  
Integer numero2 = new Integer("101");  
System.out.println(numero2.equals(numero1));
```

## La clase Math

La clase `Math` contiene constantes y métodos de uso común en matemáticas. Todas las operaciones que se realizan en esta clase utilizan el tipo `double`. Contiene la constante pi (`Math.PI`) y el número de Euler (`Math.E`). En las funciones trigonométricas, los ángulos se expresan en radianes y los métodos devuelven valores de tipo `double`. La clase `Math` incluye funciones como potenciación, redondeo, cuadrado, raíz cuadrada y muchas más.

Para más información sobre los métodos de la clase `Math`, consulte el API de Java.

<http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Math.html>

## La clase String

La clase `String` se usa para manejar cadenas de caracteres de cualquier longitud. Un objeto `String` se puede crear a partir de una secuencia de caracteres delimitados por comillas dobles.

```
String nombre = "Juan";  
String apellidos = "González López";
```

Un objeto `String` también se puede crear utilizando el constructor de la clase.

```
String mensaje = new String("Hola Mundo");
```

La clase `String` tiene un tratamiento particular en Java. Además de la construcción de objetos a partir de literales entre comillas, se pueden aplicar los operadores `+` y `+=` para concatenar objetos de tipo `String`.

```
String hola = new String("Hola");  
String espacio = new String (" ");  
String mundo = new String("Mundo");  
String holaMundo = hola + espacio + mundo;  
System.out.println(holaMundo);
```

Para conocer la longitud de un objeto `String` se utiliza el método `length()`. Por ejemplo, el objeto `holaMundo` tiene una longitud de 10 caracteres.

```
System.out.println("El texto " + holaMundo + " tiene " +  
                    holaMundo.length() + " letras");
```

## Ampliación de clases

---

Para comparar cada letra de dos objetos de tipo `String` se utiliza el método `contentEquals()`.

```
String nombre1 = "Angel";
String nombre2 = "Carlos";
System.out.println(nombre1.contentEquals(nombre2));
```

El método `String.valueOf()` devuelve una cadena correspondiente al valor de su parámetro. Este método está sobrecargado y acepta tipos `boolean`, `char`, `int`, `long`, `float` y `double`.

```
String año = String.valueOf(2011); // año = "2011"
```

El método `charAt(int posicion)` de la clase `String` devuelve el carácter almacenado en la posición indicada de una cadena de caracteres. El primer carácter de una cadena se almacena en la posición cero y el último en la posición correspondiente a la longitud de la cadena - 1.

```
String holaMundo = "Hola Mundo";
System.out.println("La primera letra de 'Hola Mundo' " +
                  " es " + holaMundo.charAt(0));
```

Para más información sobre los métodos de la clase `String`, consulte el API de Java.

<http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html>

## 6. Estructuras de control

---

El cuerpo de un programa se compone de un conjunto de sentencias que especifican las acciones que se realizan durante su ejecución. Dentro de cualquier programa, se escriben sentencias que definen la secuencia de acciones a ejecutar. Estas sentencias incluyen acciones de cálculo, entrada y salida de datos, almacenamiento de datos, etc. Las sentencias se ejecutan una a una en el orden en el que han sido escritas.

Se denomina flujo de un programa al orden de ejecución de las sentencias que forman parte del cuerpo de un programa. Las estructuras de control son una característica básica de los lenguajes que se utiliza para modificar el flujo de un programa.

Hay casos en los que el flujo de un programa debe ejecutar determinadas instrucciones solo cuando se cumple una condición. En otras ocasiones, debe repetir un conjunto de sentencias un número determinado de veces. Las estructuras de control permiten condicionar el flujo de ejecución dependiendo del estado de las variables de un programa.

Las estructuras de control básicas se pueden clasificar en estructuras de selección, de repetición y de salto.

- Selección. Permiten decidir si se ejecuta un bloque de sentencias o no.
- Repetición. Permiten ejecutar un bloque de sentencias muchas veces.
- Salto. Permiten dar un salto y continuar la ejecución de un programa en un punto distinto de la siguiente sentencia en el orden natural de ejecución.

Las estructuras de control se pueden combinar sin ningún tipo de limitación. Cualquier nuevo bloque de sentencias puede incluir estructuras de control a continuación de otras. Cuando se incluyen varias estructuras seguidas unas de otras, se dice que son estructuras de control apiladas.

Por otra parte, dentro de un bloque de una estructura de control se puede incluir otra estructura de control y dentro de este nuevo bloque se puede incluir otra estructura de control y así sucesivamente. Cuando una estructura contiene otra estructura, se dice que son estructuras de control anidadas.

Es importante destacar que no existe limitación en cuanto al número de estructuras de control apiladas o anidadas que se pueden utilizar en un

programa Java. La única restricción a tener en cuenta es la claridad y la legibilidad del programa.

## Estructuras de selección

Las estructuras de selección permiten modificar el flujo de un programa. La decisión de ejecutar un bloque de sentencias queda condicionada por el valor de una expresión lógica definida utilizando variables del programa.

### Estructura `if`

La estructura `if` se denomina estructura de selección única porque ejecuta un bloque de sentencias solo cuando se cumple la condición del `if`. Si la condición es verdadera se ejecuta el bloque de sentencias. Si la condición es falsa, el flujo del programa continúa en la sentencia inmediatamente posterior al `if`.

Una sentencia `if` tiene la siguiente sintaxis:

```
if (condicion) {  
    bloque-de-sentencias  
}
```

La condición es una expresión que evalúa un valor lógico, por lo que el resultado solo puede ser `true` o `false`. La condición siempre se escribe entre paréntesis. La selección se produce sobre el bloque de sentencias delimitado por llaves. Si el bloque de sentencias solo tiene una sentencia, entonces se puede escribir sin las llaves, como se muestra a continuación.

```
if (condicion)  
    sentencia;
```

Cuando el flujo de un programa llega a una estructura `if`, se evalúa la condición y el bloque de instrucciones se ejecuta si el valor de la condición es `true`. Si la condición es `false`, entonces se ejecuta la sentencia inmediatamente posterior al `if`.



Por ejemplo, si la calificación de un alumno es 10, entonces se debe mostrar por la consola un mensaje indicando que tiene una Matrícula de Honor.

La sentencia `if` considerando que calificación es una variable de tipo `int`:

```
if (calificacion == 10) {  
    System.out.println("Matrícula de Honor");  
}
```

En este ejemplo el mensaje "Matrícula de Honor" solo se muestra cuando el valor de la calificación es igual a 10.

### Estructura `if else`

La estructura `if-else` se denomina de selección doble porque selecciona entre dos bloques de sentencias mutuamente excluyentes. Si se cumple la condición, se ejecuta el bloque de sentencias asociado al `if`. Si la condición no se cumple, entonces se ejecuta el bloque de sentencias asociado al `else`.

Una sentencia `if-else` tiene la siguiente sintaxis:

```
if (condicion) {  
    bloque-de-sentencias-if  
}  
else {  
    bloque-de-sentencias-else  
}
```

Al igual que en el `if`, la condición se debe escribir entre paréntesis. La selección depende del resultado de evaluar la condición. Si el resultado es `true`, se ejecuta el bloque de sentencias del `if`, en cualquier otro caso se ejecuta el bloque de instrucciones del `else`. Después de ejecutar el bloque de sentencias se ejecuta la sentencia inmediatamente posterior al `if-else`.

## Estructuras de control

---

Por ejemplo, si se desea mostrar un mensaje por la consola para indicar si un número es par o impar, basta con calcular el resto de la división del número entre 2 con el operador %. Si el resto es igual a cero, entonces el número es par, en caso contrario el número es impar.

La sentencia if-else:

```
if (numero % 2 == 0)
    System.out.println("El número es par");
else
    System.out.println("El número es impar");
```

Como se ha comentado antes, los bloques de sentencias son mutuamente excluyentes. Si se cumple la condición se ejecuta un bloque de sentencias, en caso contrario se ejecuta el otro bloque de sentencias. Teniendo en cuenta esto, se podría escribir una sentencia if-else con la condición contraria y con los bloques de sentencias intercambiados.

```
if (numero % 2 != 0)
    System.out.println("El número es impar");
else
    System.out.println("El número es par");
```

Si fuera necesario evaluar más de una condición, entonces se deben utilizar varias estructuras de selección anidadas. Por ejemplo, para mostrar la calificación de un alumno, es necesario evaluar las condiciones que se indican en la siguiente tabla.

Calificación	Descripción
10	Matrícula de Honor
9	Sobresaliente
7, 8	Notable
6	Bien
5	Aprobado
0,1,2,3,4	Suspenso

De la tabla anterior, se puede ver que las condiciones son excluyentes entre sí. Si la calificación es 10 se muestra "Matrícula de Honor". En caso contrario la calificación es menor de 10 y es necesario seleccionar entre "Sobresaliente", "Notable", "Bien", "Aprobado" y "Suspenso". Si la calificación es 9 se muestra "Sobresaliente". En caso contrario, la calificación es menor de 9 y se debe seleccionar entre "Notable", "Bien", "Aprobado" y "Suspenso". Si la calificación es mayor o igual a 7 se muestra "Notable". En caso contrario la calificación es menor de 7 y se debe seleccionar entre "Bien", "Aprobado" y "Suspenso". Si la calificación es 6 se muestra "Bien". En caso contrario la calificación es menor o igual a 6 y se debe seleccionar entre "Aprobado" y "Suspenso". Si la calificación es 5 se muestra "Aprobado", en caso contrario "Suspenso".

La sentencia if-else:

```
int calificacion = 7;

if (calificacion == 10)
    System.out.println("Matrícula de Honor");
else
    if (calificacion == 9)
        System.out.println("Sobresaliente");
    else
        if (calificacion >= 7)
            System.out.println("Notable");
        else
            if (calificacion == 6)
                System.out.println("Bien");
            else
                if (calificacion == 5)
                    System.out.println("Aprobado");
                else
                    System.out.println("Suspenso");
```

### Estructura if else if

La estructura if-else-if se puede aplicar en los mismos casos en que se utiliza un if-else anidado. Esta estructura permite escribir de forma abreviada las condiciones de un if-else anidado.

Una sentencia if-else-if tiene la siguiente sintaxis:

```
if (condicion-1) {
    bloque-de-sentencias-condicion-1
} else if (condicion-2) {
    bloque-de-sentencias-condicion-2
} else {
    bloque-de-sentencias-else
}
```

La sentencia if-else-if para el ejemplo de las calificaciones:

```
int calificacion = 7;

if (calificacion == 10) {
    System.out.println("Matrícula de Honor");
} else if (calificacion == 9) {
    System.out.println("Sobresaliente");
} else if (calificacion >= 7) {
    System.out.println("Notable");
} else if (calificacion == 6) {
    System.out.println("Bien");
} else if (calificacion == 5) {
    System.out.println("Aprobado");
} else {
    System.out.println("Suspenso");
}
```

## Estructura switch

La estructura `switch` es una estructura de selección múltiple que permite seleccionar un bloque de sentencias entre varios casos. En cierto modo, es parecido a una estructura de `if-else` anidados. La diferencia está en que la selección del bloque de sentencias depende de la evaluación de una expresión que se compara por igualdad con cada uno de los casos. La estructura `switch` consta de una expresión y una serie de etiquetas `case` y una opción `default`. La sentencia `break` indica el final de la ejecución del `switch`.

Una sentencia `switch` tiene la siguiente sintaxis:

```
switch (expresion) {
    case valor-1:
        bloque-de-sentencias-1;
        break;
    case valor-2:
        bloque-de-sentencias-2;
        break;
    case valor-3:
        bloque-de-sentencias-3;
        break;
    case valor-4:
        bloque-de-sentencias-4;
        break;
    case valor-5:
        bloque-de-sentencias-5;
        break;
    default:
        bloque-de-sentencias-default;
        break;
}
```

## Estructuras de control

---

La expresión debe devolver un valor de tipo entero (`int`) o carácter (`char`) y es obligatorio que la expresión se escriba entre paréntesis. A continuación de cada `case` aparece uno o más valores constantes del mismo tipo que el valor que devuelve la expresión del `switch`.

Para interrumpir la ejecución de las sentencias del `switch` se utiliza la sentencia `break` que provoca la finalización del `switch`. El flujo del programa continúa en la sentencia inmediatamente posterior al `switch`.

Una vez que se evalúa la expresión del `switch`, se comprueba si coincide con el valor del primer `case`. En caso contrario, se comprueba si coincide con el valor del segundo `case` y así sucesivamente. Cuando el valor de la expresión coincide con el valor de uno de los `case`, se empieza a ejecutar el bloque de instrucciones correspondiente al `case` hasta encontrar una sentencia `break` o al llegar al final de la estructura `switch` donde se cierra la llave. Si no se encuentra un `case` que coincida con el valor de la expresión, se ejecuta el bloque de sentencias correspondiente a la etiqueta `default`.

Para asegurar el correcto flujo de ejecución de un programa durante la evaluación de una sentencia `switch`, es recomendable incluir una sentencia `break` al final del bloque de instrucciones de cada `case`, incluido el correspondiente a la etiqueta `default`. Esto es importante, porque si se omite la sentencia `break`, cuando finaliza la ejecución del bloque de sentencias de un `case`, el flujo del programa continúa ejecutando los `case` siguientes y esto puede provocar un comportamiento erróneo del programa. El siguiente ejemplo muestra la importancia del uso del `break` en una sentencia `switch`.

Suponga que en una empresa de consultoría la categoría profesional de un empleado se calcula a partir de su tasa de coste. La tabla muestra los valores de las tasas y sus correspondientes categorías.

Calificación	Descripción
Menor de 80	La categoría es 'C' de consultor Junior
Mayor o igual a 80 y menor de 120	La categoría es 'B' de consultor Senior
Mayor o igual a 120	La categoría es 'A' de socio

Programa que utiliza un switch para seleccionar la descripción correspondiente a cada categoría.

```
public class CategoriasProfesionales {
    public static void main(String[] args) {
        int tasaEstandar = 150;
        char categoriaProfesional;

        if (tasaEstandar < 80)
            categoriaProfesional = 'C';
        else
            if (tasaEstandar < 120)
                categoriaProfesional = 'B';
            else
                categoriaProfesional = 'A';

        System.out.print("Tasa " + tasaEstandar + " euros, ");
        System.out.print("categoría " + categoriaProfesional +
            " de ");

        switch (categoriaProfesional) {
            case 'A': System.out.print("Socio ");
            case 'B': System.out.print("Senior ");
            case 'C': System.out.print("Junior ");
            default: System.out.print(";Indefinida! ");
        }
    }
}
```

El valor de la `tasaEstandar` es 150 euros, de manera que se asigna el valor 'A' a la variable `categoriaProfesional`. En el `switch` se cumple el primer `case` y se muestra por la consola el texto "Socio". Según esto, el programa debería mostrar el mensaje:

```
Tasa 90 euros, categoría 'A' de Socio
```

No es así, el primer `case` no tiene `break` por lo que no finaliza la ejecución del `switch` y se ejecutan los bloques de sentencias correspondientes al

segundo case, al tercer case y al default. El programa muestra por la consola el mensaje:

```
Tasa 90 euros, categoría 'A' de Socio Senior Junior  
;Indefinida!
```

Para evitar que se ejecute más de un bloque de sentencias de un switch, se debe incluir un break al final del bloque de cada case.

```
switch (categoriaProfesional) {  
    case 'A': System.out.print("Socio ");  
        break;  
    case 'B': System.out.print("Senior ");  
        break;  
    case 'C': System.out.print("Junior ");  
        break;  
    default: System.out.print(";Indefinida! ");  
        break;  
}
```

La sentencia break al final de cada case asegura que solo se ejecuta un case y después finaliza el switch.



```
public class CategoriasProfesionales {
    public static void main(String[] args) {
        int tasaEstandar = 150;
        char categoriaProfesional;

        if (tasaEstandar < 80)
            categoriaProfesional = 'C';
        else
            if (tasaEstandar < 120)
                categoriaProfesional = 'B';
            else
                categoriaProfesional = 'A';

        System.out.print("Tasa " + tasaEstandar + " euros, ");
        System.out.print("categoría " + categoriaProfesional +
            " de ");

        switch (categoriaProfesional) {
            case 'A': System.out.print("Socio ");
                break;
            case 'B': System.out.print("Senior ");
                break;
            case 'C': System.out.print("Junior ");
                break;
            default: System.out.print(";Indefinida! ");
                break;
        }
    }
}
```

De nuevo, se asigna el valor 'A' a la variable `categoriaProfesional`. En el `switch` se cumple el primer `case` y la salida por la consola es:

```
Tasa 90 euros, categoría 'A' de Socio
```

Volviendo al ejemplo de las calificaciones que antes se ha codificado utilizando if-else anidados, ahora se utiliza un switch.

```
public class Calificaciones {
    public static void main(String[] args) {
        int calificacion = 9;
        switch (calificacion) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4: System.out.println("Suspenso");
                    break;
            case 5: System.out.println("Aprobado");
                    break;
            case 6: System.out.println("Bien");
                    break;
            case 7:
            case 8: System.out.println("Notable");
                    break;
            case 9: System.out.println("Sobresaliente");
                    break;
            case 10: System.out.println("Matrícula de
Honor");
                    break;
            default: System.out.println("No presentado");
                    break;
        }
    }
}
```

Es importante ver que los case correspondientes a los valores 0, 1, 2 y 3 se han dejado vacíos porque el bloque de sentencias para estos casos es el mismo que el del case 4. Para evitar repetir este código varias veces, se deja vacío el bloque correspondiente a estos casos y no se incluye el `break`. De esta manera, cuando se cumple uno de ellos, se ejecuta el bloque de sentencias correspondiente al case, que para los valores 0, 1, 2 y 3, está vacío. Como no hay `break`, se ejecutan las siguientes líneas del programa hasta llegar al bloque de sentencias correspondiente al case 4, que muestra el mensaje "Suspenseo" y, cuando encuentra el `break`, finaliza el `switch`.

El `switch` se diferencia de otras estructuras en que no es necesario delimitar entre llaves el bloque de sentencias de cada case. Solo son obligatorias las llaves de inicio y fin del `switch`. En una estructura `switch` es obligatorio que los valores de los distintos casos sean diferentes. Si no hay un caso que coincida con el valor de la expresión y no se incluye la etiqueta `default`, entonces el `switch` no ejecuta ninguno de los bloques de sentencias.

Por último, conviene recordar que un `switch` es una estructura apropiada para seleccionar entre un conjunto de opciones simples o predefinidas. No se puede aplicar cuando la selección se basa en opciones complejas o cuando dependen de un intervalo de valores. En ese caso es necesario utilizar una estructura `if-else` anidada.

## El operador condicional

El operador condicional (`?:`) se relaciona con la estructura `if-else`. Es el único operador de Java que utiliza tres operandos. El primer operando es una condición lógica, el segundo es el valor que toma la expresión cuando la condición es `true` y el tercero es el valor que toma la expresión cuando la condición es `false`.

El operador evalúa la condición delante del símbolo `?`, que puede escribirse entre paréntesis. Si vale `true` devuelve el valor que aparece a continuación del signo `?`. Si es `false` devuelve el valor que aparece a continuación de los dos puntos.

El operador condicional tiene la siguiente sintaxis:

```
condicion-logica ? valor-si-verdadero : valor-si-falso;
```

La condición lógica también se puede expresar entre paréntesis:

```
(condicion-logica)? valor-si-verdadero : valor-si-falso;
```

Después de evaluar la condición lógica, se devuelve el valor correspondiente al resultado lógico verdadero o falso. Por ejemplo, dada la edad de una persona, se desea mostrar un mensaje por la consola que indique si es mayor de edad o no.

```
int edad = 16;
String txt;
txt = (edad >= 18) ? "Mayor de edad" : "Menor de edad";
System.out.print(txt);
```

La condición lógica es edad mayor o igual a 18 años. Si es verdadera, el operador devuelve el texto "Mayor de edad", en caso contrario devuelve "Menor de edad".

En este ejemplo la variable edad se inicializa a 16, por lo que el mensaje que se muestra por la consola es:

```
Menor de edad
```

## Estructuras de repetición

Las estructuras de repetición permiten repetir muchas veces un bloque de sentencias. A estas estructuras también se les conoce como estructuras iterativas o bucles.

Como las estructuras de selección, las estructuras de repetición se pueden combinar y anidar. Es frecuente utilizar una estructura de repetición que contenga un bloque de sentencias que combine otras estructuras de repetición y de selección.

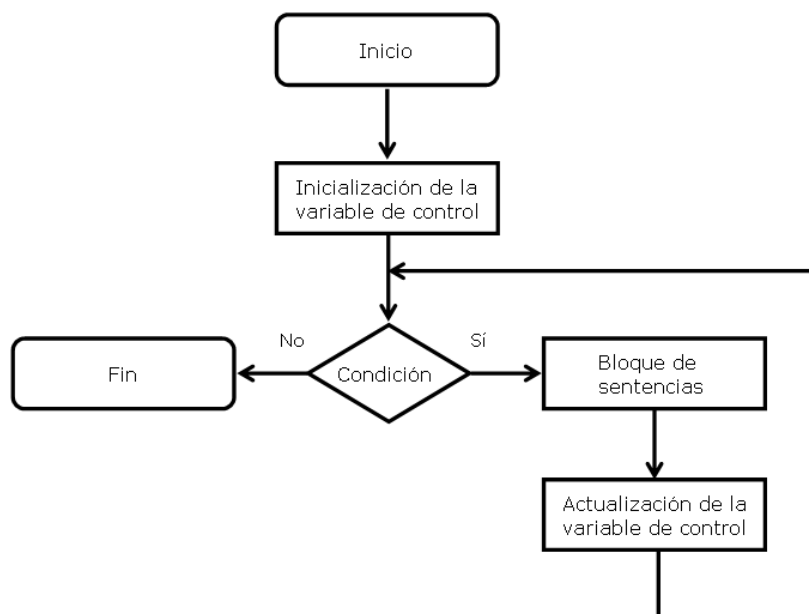
Las estructuras de repetición se componen de cuatro partes: la inicialización, la condición, el bloque de sentencias y la actualización.

- **Inicialización.** Permite inicializar la estructura iterativa, normalmente consiste en la declaración e inicialización de la variable de control del bucle.
- **Condición.** Define la condición que se evalúa para ejecutar el bloque de sentencias de la estructura iterativa. Dependiendo del tipo de estructura que se utilice, la condición se comprueba antes o después de realizar cada iteración.
- **Bloque de sentencias.** Conjunto de sentencias que se ejecutan dentro de la estructura iterativa.
- **Actualización.** Actualización de la variable de control del bucle. Normalmente se realiza al finalizar la ejecución del bloque de sentencias.

## Estructura while

La estructura de repetición `while` repite el bloque de sentencias mientras la condición del `while` es verdadera.

El diagrama de flujo de una estructura `while` muestra que la condición se verifica justo después de inicializar la variable de control. Si el resultado de evaluar la condición por primera es falso, entonces no se ejecuta el bloque de sentencias.



Un `while` tiene la siguiente sintaxis:

```
inicialización;
while (condición) {
    bloque-de-sentencias;
    actualización;
}
```

Esta es la sintaxis general. La condición del `while` se escribe obligatoriamente entre paréntesis.

Un `while` no necesariamente requiere inicialización y actualización de una variable de control. En ese caso solo es necesario incluir la condición y el bloque de sentencias:

```
while (condición) {
    bloque-de-sentencias;
}
```

Cuando el programa ejecuta un `while`, lo primero que hace es evaluar la condición. Si es verdadera ejecuta el bloque de sentencias, si es falsa finaliza el `while`.

En cada iteración, cuando finaliza la ejecución del bloque de sentencias se vuelve a evaluar la condición. De nuevo, si es verdadera ejecuta una vez más el bloque de sentencias, si es falsa finaliza el `while`. Cuando esto se produce, el flujo del programa continúa en la sentencia inmediatamente posterior al `while`.

Si la primera vez que se evalúa la condición el resultado es falso, entonces no se ejecuta el bloque de sentencias. Por esta razón, se dice que un `while` se ejecuta cero o más veces. Si la condición siempre es verdadera, entonces el `while` nunca termina y se ejecuta indefinidamente. Esto se conoce como bucle infinito.

El siguiente ejemplo muestra el uso del `while` para calcular la función factorial de un número entero positivo 'n'.

La función factorial se define:

$$0! = 1$$

$$1! = 1$$

$$2! = 1 \times 2$$

$$3! = 1 \times 2 \times 3$$

$$4! = 1 \times 2 \times 3 \times 4$$

...

$$n! = 1 \times 2 \times 3 \times 4 \times 5 \times \dots \times (n-2) \times (n-1) \times (n)$$

De la definición anterior, se puede calcular el factorial utilizando una estructura repetitiva con una variable de control que empiece en 1 y termine en 'n'. La actualización de la variable de control del while suma uno cada vez.

Programa que calcula la función factorial de un número utilizando la estructura while.

```
public class FactorialWhile {
    public static void main(String[] args) {
        int n = 5;          // n se inicializa a 5 para calcular 5!
        int factorial = 1 // factorial se inicializa a 1

        int i = 1;         // el valor inicial de i es 1

        while (i <= n) {
            factorial = factorial * i;
            i++;
        }

        System.out.println("El factorial de " + n + " es " +
                           factorial);
    }
}
```

## Estructuras de control

---

En la expresión `factorial = factorial * i` la variable `factorial` aparece dos veces. Primero se calcula el producto `factorial * i` y después se asigna este resultado a la variable `factorial`. Es por esto que la tabla muestra una columna con los valores de los operandos del producto y otra con el valor final de la variable `factorial`. La siguiente tabla muestra el proceso de cálculo que se realiza en el `while`.

i	n	factorial * i	factorial
1	5	1 * 1	1
2	5	1 * 2	2
3	5	2 * 3	6
4	5	6 * 4	24
5	5	24 * 5	120

La variable `factorial` se inicializa a 1. En la primera iteración `i` vale 1, se calcula el producto `factorial * i` con los valores `1 * 1` y se asigna 1 a la variable `factorial`. En la segunda iteración `i` vale 2, se calcula `factorial * i` con los valores `1 * 2` y se asigna 2 a la variable `factorial`. En la tercera iteración `i` vale 3, se calcula `factorial * i` con los valores `2 * 3` y se asigna 6 a la variable `factorial`. En la cuarta iteración `i` vale 4, se calcula `factorial * i` con los valores `6 * 4` y se asigna 24 a la variable `factorial`. En la última iteración `i` vale 5, se calcula el producto `factorial * i` con los valores `24 * 5` y se asigna 120 a la variable `factorial`.

De los resultados de la tabla anterior, se puede observar que no es necesario calcular el producto `factorial * i` en la primera iteración cuando `i` vale 1. Este producto siempre va a dar como resultado 1.

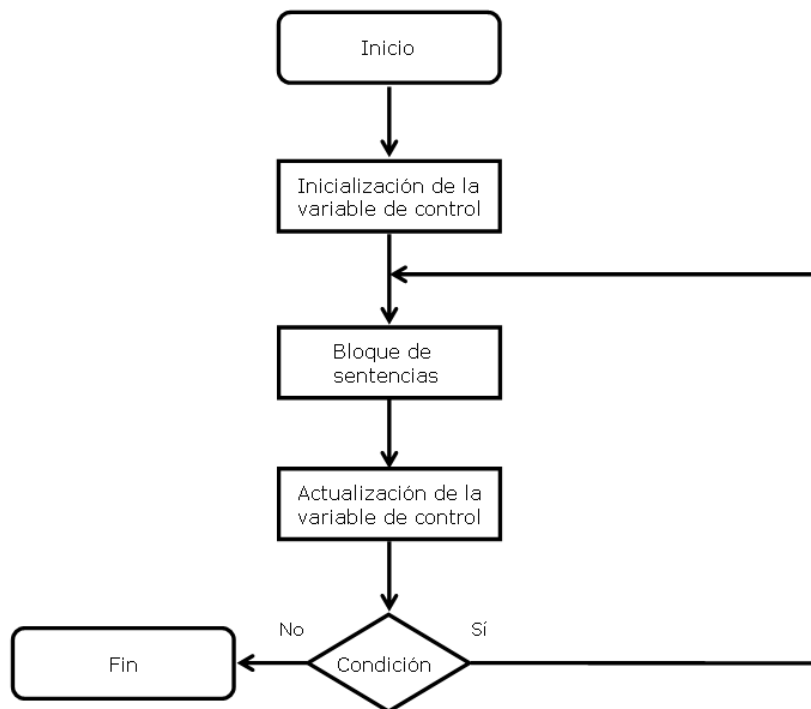
El algoritmo es más eficiente si se elimina la primera iteración, basta con inicializar la variable de control `i` a 2. En este ejemplo, el valor inicial de `i` es 1 para que el algoritmo sea más claro.



## Estructura do-while

La estructura de repetición do-while ejecuta el bloque de sentencias al menos una vez. Después comprueba la condición y repite el bloque de sentencias mientras la condición es verdadera.

El diagrama de flujo de una estructura do-while muestra que la condición se verifica al final, después de ejecutar el bloque de sentencias la primera vez.



Un do-while tiene la siguiente sintaxis:

```
inicialización;  
do {  
    bloque-de-sentencias;  
    actualizacion;  
} while (condición);
```

Esta es la sintaxis general. La condición del do-while se escribe obligatoriamente entre paréntesis.

## Estructuras de control

---

Un do-while no necesariamente utiliza una variable de control. En ese caso solo es necesario incluir la condición y el bloque de sentencias:

```
do {  
    bloque-de-sentencias;  
} while (condición);
```

Cuando el programa ejecuta un do-while, lo primero que hace es ejecutar el bloque de sentencias y luego evalúa la condición. Si es verdadera, ejecuta de nuevo el bloque de sentencias, si es falsa finaliza el do-while.

En cada iteración, cuando finaliza la ejecución del bloque de sentencias se vuelve a evaluar la condición. De nuevo, si es verdadera ejecuta una vez más el bloque de sentencias, si es falsa finaliza el do-while. Cuando esto se produce, el flujo del programa continúa en la sentencia inmediatamente posterior al do-while.

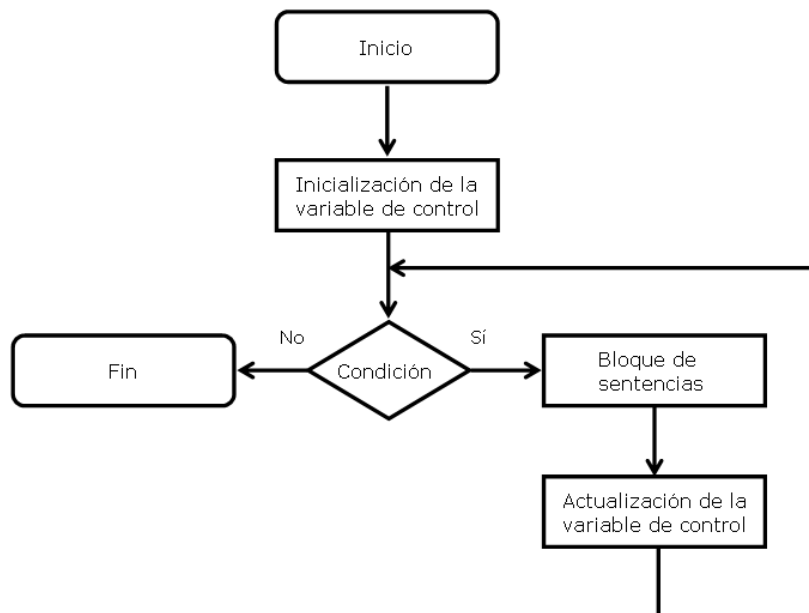
Programa que calcula la función factorial de un número utilizando la estructura do-while.

```
public class FactorialDoWhile {  
    public static void main(String[] args) {  
        int n = 5;          // n se inicializa a 5 para calcular 5!  
        int factorial = 1 // factorial se inicializa a 1  
  
        int i = 1;         // el valor inicial de i es 1  
  
        do {  
            factorial = factorial * i;  
            i++;  
        } while (i <= n);  
  
        System.out.println("El factorial de " + n + " es " +  
                           factorial);  
    }  
}
```

## Estructura for

La estructura de repetición `for` repite el bloque de sentencias mientras la condición del `for` es verdadera. Un `for` es un caso particular de la estructura `while`. Solo se debe utilizar cuando se sabe el número de veces que se debe repetir el bloque de sentencias.

El diagrama de flujo de una estructura `for` es igual que el de un `while`. Un `for` verifica la condición justo después de inicializar la variable de control. Si el resultado de evaluar la condición por primera es falso, entonces no se ejecuta el bloque de sentencias.



Un `for` tiene la siguiente sintaxis:

```
for (inicialización; condición; actualización) {  
    bloque-de-sentencias;  
}
```

Cuando el programa ejecuta un `for`, lo primero que hace es evaluar la condición. Si es verdadera ejecuta el bloque de sentencias, si es falsa finaliza el `for`.

## Estructuras de control

---

En cada iteración, cuando finaliza la ejecución del bloque de sentencias se vuelve a evaluar la condición. De nuevo, si es verdadera ejecuta una vez más el bloque de sentencias, si es falsa finaliza el `for`. Cuando esto se produce, el flujo del programa continúa en la sentencia inmediatamente posterior al `for`.

Programa que calcula la función factorial de un número utilizando la estructura `for`.

```
public class FactorialFor {
    public static void main(String[] args) {
        int n = 5;          // n se inicializa a 5 para calcular 5!
        int factorial = 1 // factorial se inicializa a 1

        for (int i=1; i <= n; i++) {
            factorial = factorial * i;
        }

        System.out.println("El factorial de " + n + " es " +
                           factorial);

    }
}
```

Normalmente la variable de control se declara y se inicializa en la sección de inicialización de la variable. En este ejemplo se hace `int i= 1`, es decir se declara una variable `i` de tipo `int` y se inicializa a 1. La condición del `for` es `i <= n`, la misma que se ha utilizado en el `while` y el `do-while`. Por último, la variable `i` se incrementa en 1 en cada iteración. En el `for`, el `while` y el `do-while` el incremento de `i` se realiza con el operador `++`.

Es posible combinar estructuras de selección y estructuras de iteración. Si se define una estructura de repetición dentro de otra, entonces se tiene una estructura de repetición anidada.

El siguiente ejemplo utiliza tres for anidados. ¿Cuántas veces se muestra por la consola el mensaje "Hola Mundo"?

```
public class ForAnidado {
    public static void main(String[] args) {

        for (int i=1; i <= 5; i++)
            for (int j=2; j <= 4; j++)
                for (int k=3; k <= 6; k++)
                    System.out.println("Hola Mundo");
    }
}
```

Para saber cuántas veces se imprime el mensaje es necesario saber cuántas veces se repite cada `for`. El `for` de `i` se repite 5 veces, el `for` de `j` se repite 3 veces y el `for` de `k` se repite 4 veces. Como el `for` de `k` está dentro del `for` de `j` y éste dentro del `for` de `i`, el mensaje se imprime  $5 \times 3 \times 4$  veces, un total de 60 veces.

## Uso de las estructuras de repetición

Es importante utilizar la estructura de repetición más apropiada para cada caso. En general, se recomienda seguir los siguientes criterios:

- El `while` se debe utilizar cuando no se sabe el número de veces que se va a repetir el bloque de sentencias.
- El `do-while` se debe utilizar cuando el bloque de sentencias se debe ejecutar al menos una vez.
- El `for` se debe utilizar cuando se sabe el número de veces que se va a repetir el bloque de sentencias. Un `for` es útil cuando se conoce el valor inicial para la variable de control del bucle y además es necesario utilizar una expresión aritmética para actualizar esta variable.

**Ejemplo de uso de while.** Utilice una estructura while para determinar mediante restas sucesivas si un número entero positivo es par.

Para saber si un número entero es par es necesario restar 2 sucesivamente mientras el número sea mayor o igual a 2. Si después de realizar las restas el número es cero, el número es par, si no, es impar.

```
public class NumeroParImpar {
    public static void main(String[] args) {

        // este programa verifica si un número positivo
        // es para o impar

        int numero = 12;    // el valor inicial del número

        while (numero >= 2) {
            numero = numero - 2;
        }

        if (numero == 0)
            System.out.println("El número es par");
        else
            System.out.println("El número es impar");

    }
}
```

**Ejemplo de uso de do-while.** Utilice una estructura do-while que muestre por la consola números enteros aleatorios entre 0 y 100 hasta que salga el número 50.

Para calcular un número aleatorio se utiliza el método `random()` de la clase `Math`. Este método devuelve un valor de tipo `double` entre 0 y 1. Este resultado se multiplica por 100 para que el valor esté en el rango entre 0 y 100. Antes de asignar el resultado a la variable `numero` se convierte a un valor entero utilizando `(int)`.

El `do-while` se ejecuta al menos una vez y muestra los números aleatorios calculados mientras el número sea diferente de 50.

```
public class NumerosAleatorios {
    public static void main(String[] args) {

        // este programa muestra números enteros aleatorios
        // entre 0 y 100 hasta que sale el 50

        do {
            numero = (int) (100 * Math.random());
            System.out.println("Número aleatorio: " + numero);
        } while (numero != 50);

    }
}
```

**Ejemplo de uso de for.** Utilice una estructura `for` para calcular la función potencia de un número entero positivo utilizando productos. La potencia se calcula como el producto de la base repetido tantas veces como el valor del exponente.

potencia = base x base x base x base x base x ... x base

Inicialmente, el valor de la variable potencia es 1 porque cualquier número elevado a la potencia cero es 1.

```
public class PotenciaFor {
    public static void main(String[] args) {

        // este programa calcula 2^10

        int base = 2;          // base
        int exponente = 10;    // exponente al que se eleva la base
        int potencia = 1;      // potencia se inicializa a 1
                               // porque x^0 = 1

        for (int i=1; i <= exponente; i++) {
            potencia = potencia * base;
        }

        System.out.println("La potencia es " + potencia);

    }
}
```



## Estructuras de salto

En Java existen dos sentencias que permiten modificar el flujo secuencial de un programa y provocan un salto en la ejecución. Estas sentencias son `break` y `continue`. Ambas se utilizan con las estructuras de repetición para interrumpir la ejecución con `break` o volver al principio con `continue`. Además, el `break` se utiliza para interrumpir la ejecución de un `switch`.

### Sentencia `break`

La sentencia `break` se utiliza para interrumpir la ejecución de una estructura de repetición o de un `switch`. Cuando se ejecuta el `break`, el flujo del programa continúa en la sentencia inmediatamente posterior a la estructura de repetición o al `switch`.

### Sentencia `continue`

La sentencia `continue` únicamente puede aparecer en una estructura de repetición. Cuando se ejecuta un `continue`, se deja de ejecutar el resto del bloque de sentencias de la estructura iterativa para volver al inicio de ésta.

## Uso de `break` y `continue`

A continuación se muestran ejemplos del uso de las sentencias `break` y `continue`.

**Ejemplo de uso de `break` en un `switch`.** Desarrolle un programa que cuente el número de vocales, consonantes y espacios de una cadena de caracteres.

Utilice un `for` para comparar cada una de las letras de la frase. Dentro del `for` utilice un `switch` para seleccionar entre vocales, consonantes y espacios. Las variables `vocales`, `consonantes` y `espacios` se inicializan a cero y se utilizan para contar el número de veces que aparecen en la frase.

Defina una variable `letra` de tipo `char`. Almacene la letra correspondiente a la posición `i` de la cadena de caracteres. Utilice el método `charAt(i)` de la clase `String` para copiar el valor de este carácter a la variable `letra`.

Utilice la sentencia `break` al final del bloque de sentencias de los `case` correspondientes a vocales, espacios y consonantes.

```
public class ConsonantesVocales {
    public static void main(String[] args) {
        String frase = "Hola Mundo";
        char letra;
        int vocales = 0, consonantes = 0, espacios = 0;

        for (int i=0; i<frase.length(); i++) {
            letra = frase.charAt(i);

            switch (letra) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                case 'A':
                case 'E':
                case 'I':
                case 'O':
                case 'U': vocales++;
                        break;
                case ' ': espacios++;
                        break;
                default: consonantes++;
                        break;
            }
        }

        System.out.println("La frase '" + frase + "' tiene " +
            vocales + " vocales, " +
            consonantes + " consonantes y " +
            espacios + " espacios. ");
    }
}
```

**Ejemplo de uso de break en un do-while.** Modifique el programa de los números aleatorios desarrollado en el ejemplo de uso de un do-while. Incluya un break que interrumpa el do-while cuando el número aleatorio sea igual a 25. El programa debe terminar cuando el número aleatorio sea 25 o 50.

```
public class NumerosAleatoriosConBreak {
    public static void main(String[] args) {

        // este programa muestra números enteros aleatorios
        // entre 0 y 100 hasta que sale el 25 o el 50

        do {
            numero = (int) (100 * Math.random());
            System.out.println("Número aleatorio: " + numero);

            if (numero == 25)
                break;

        } while (numero != 50);

    }
}
```

**Ejemplo de uso de continue en un for.** Desarrolle un programa que muestre por consola los números pares entre 2 y 10. Utilice un for para valores de i de 1 a 10 y aplique la sentencia continue para interrumpir la ejecución de las iteraciones impares.

```
public class NumerosPares {
    public static void main(String[] args) {

        for (int i=1; i<=10; i++) {
            if (i % 2 != 0)
                continue; //el número es impar, se interrumpe la iteración

            System.out.println("Números pares: " + i);
        }
    }
}
```

## 7. Estructuras de almacenamiento

---

### Arrays

Java proporciona una estructura de almacenamiento denominada array que permite almacenar muchos objetos de la misma clase e identificarlos con el mismo nombre.

La declaración de un array tiene la siguiente sintaxis:

```
tipo-o-clase[] identificador-array;  
o  
tipo-o-clase identificador-array[];
```

Por ejemplo, un array de números enteros se puede declarar de dos formas:

```
int[] numeros;  
o  
int numeros[];
```

Ambas declaraciones son equivalentes, el tipo base del array es `int` y el nombre del array es `numeros`. Todos los elementos de la estructura `numeros[]` almacenan un `int`. La primera declaración define un array de objetos de tipo primitivo `int` con identificador `numeros`. La segunda declaración dice que cada elemento de la forma `numeros[]` es de tipo `int`.

Ejemplos de declaraciones de arrays:

```
int[] numerosEnteros;    // array de tipo int  
double[] numerosReales; // array de tipo double  
String[] nombres;      // array de tipo String  
Object[] objetos;      // array de la clase Object  
Vehiculo[] vehiculos;  // array de la clase Vehiculo  
Turismo[] turismos;    // array de la clase Turismo
```

Se denomina tipo base del array al tipo que se declara para sus elementos. Este tipo base puede ser un tipo primitivo de Java, un objeto o una clase definida. En los ejemplos anteriores se han utilizado tipos primitivos y clases como tipo base. El array `numerosEnteros` almacena objetos del primitivo `int`. El array `nombres` almacena objetos de la clase `String`. El array `objetos` almacena referencias a instancias de la clase `Object` de Java. El array `vehiculos` almacena objetos de la clase `Vehiculo`.

Además de declarar un array es necesario indicar el número de elementos que va a almacenar. Un array es un objeto y como cualquier objeto de un programa Java, su valor inicial es `null`. Antes de hacer referencia a los elementos del array es necesario instanciar el objeto.

Cuando se instancia un objeto array se asigna un espacio de memoria para almacenar los elementos del array. Para esto es necesario saber el número total de elementos que va a almacenar.

La instanciación de un objeto array se hace de la siguiente forma:

```
nombres = new String[100];  
vehiculos = new Vehiculo[50];
```

En este ejemplo, el array `nombres` tiene capacidad para almacenar hasta 100 objetos de tipo `String`. El array `vehiculos` puede almacenar hasta 50 objetos de la clase `Vehiculo`.

Cuando se crea un array se inicializa el valor de todos sus elementos al valor por defecto del tipo base del array: cero para los números, `false` para los `boolean`, `\u0000` para los caracteres y `null` para las referencias a objetos.

De forma similar al resto de objetos de Java, un array se puede inicializar al momento de la declaración. En este caso se inicializa al valor por defecto del tipo del array.

```
int[] numerosEnteros = new int[10];  
String[] nombres = new String[100];  
Vehiculo[] vehiculos = new Vehiculo[50];  
Turismo[] turismos = new Turismo[50];
```

Un array también se puede inicializar indicando la lista de valores que va a almacenar:

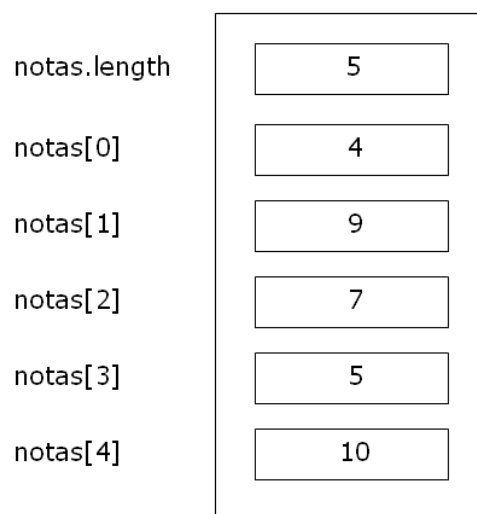
```
String[] diasLaborables = {"Lunes",  
                             "Martes",  
                             "Miércoles",  
                             "Jueves",  
                             "Viernes"};
```

```
int[] enteros = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

El array `diasLaborables` almacena 5 objetos de tipo `String` con los nombres de los días laborables de la semana. El array `enteros` almacena 11 números enteros con valores de 0 a 10.

Para hacer referencia a cada elemento de un array es necesario indicar la posición que ocupa en la estructura de almacenamiento. Esta posición se denomina índice. El primer elemento de un array se almacena en la posición cero y el último elemento en la posición  $n-1$ , donde  $n$  es el tamaño del array.

Por ejemplo, el array `notas` almacena 5 números enteros:



La declaración del array:

```
int[] notas = {4, 9, 7, 5, 10};
```

El primer elemento del array se almacena en la posición 0 y el último en la posición 4, que equivale a su tamaño menos 1. El atributo `length` almacena el tamaño de un array. En este ejemplo, la última posición del array es `notas.length - 1`.

Este array `notas` almacena 5 calificaciones, `notas[0]` es el primer elemento del array y `notas[4]` el último. Para mostrar las calificaciones almacenadas en el array, se puede utilizar un `for` con una variable de control que vaya de cero hasta la longitud del array menos 1.

```
for (int i=0; i<=notas.length - 1; i++)
    System.out.println("notas[" + i + "] es " +
        notas[i]);
```

La salida por la consola:

```
notas[0] es 4
notas[1] es 9
notas[2] es 7
notas[3] es 5
notas[4] es 10
```

El siguiente `for` es equivalente al anterior. En vez de definir el límite de la variable de control menor o igual a `notas.length-1`, se hace estrictamente menor que `notas.length`.

```
for (int i=0; i<notas.length; i++)
    System.out.println("notas[" + i + "] es " +
        notas[i]);
```



El atributo `length` de un array almacena un valor numérico que se puede consultar pero no se puede modificar. Es una buena práctica de programación utilizar el atributo `length` para hacer referencia al tamaño de un array.

Otra forma de mostrar los valores almacenados en un array es utilizando un `for` "para todo", donde la variable de control del `for`, con identificador `nota`, toma el valor de todos los elementos de la estructura de almacenamiento, en este caso, el array `notas`.

```
for (int nota : notas)
    System.out.println(nota);
```

La variable `nota` del `for` "para todo" toma los valores `nota[0]`, `nota[1]`, `nota[2]`, `nota[3]` y `nota[4]` en cada iteración. Cuando se utiliza un `for` "para todo" no hace falta indicar los límites de la variable de control del `for`, basta que esta variable sea del tipo almacenado en el array. En este ejemplo `nota` es de tipo `int`.

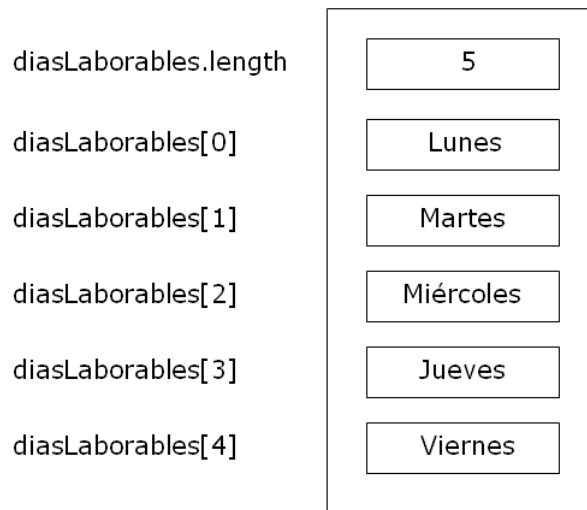
La salida por la consola:

```
4
9
7
5
10
```

## Estructuras de almacenamiento

---

El array `diasLaborables` almacena los nombres de los días laborables de la semana:



La declaración del array:

```
String[] diasLaborables = {"Lunes",  
                             "Martes",  
                             "Miércoles",  
                             "Jueves",  
                             "Viernes"};
```

De nuevo, se utiliza un `for` "para todo" para mostrar el contenido del array `diasLaborables` que almacena objetos de tipo `String`.

```
for (String dia: diasLaborables)  
    System.out.println(dia);
```

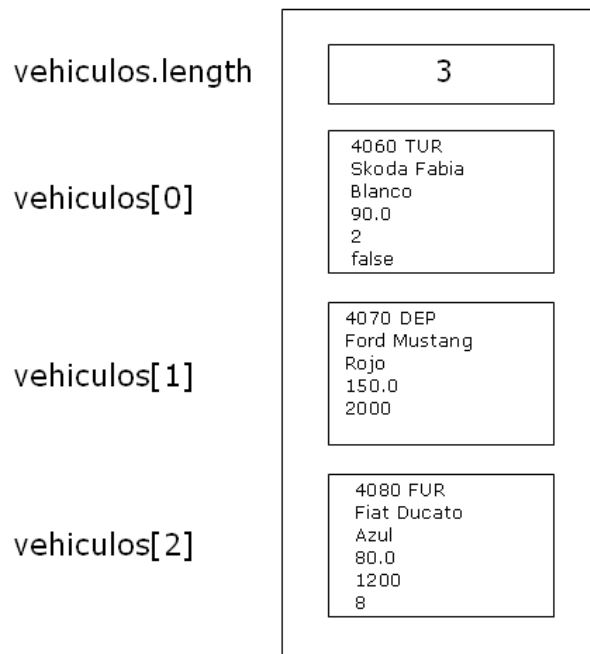
La variable `dia` del `for` "para todo" es de tipo `String` y toma los valores `diasLaborables[0]`, `diasLaborables[1]`, `diasLaborables[2]`, `diasLaborables[3]` y `diasLaborables[4]` en cada iteración.

La salida por consola:

Lunes  
Martes  
Miércoles  
Jueves  
Viernes

El array `vehiculos` almacena 3 objetos de la clase `Vehiculo`: un turismo, un deportivo y una furgoneta. Cada elemento del array es una instancia de las subclases de `Vehiculo`.

El array `vehiculos` almacena referencias a objetos de la clase `Vehiculo`.



La declaración del array:

```
Vehiculo[] vehiculos = { new Turismo("4060 TUR",
                                     "Skoda", "Fabia", "Blanco",
                                     90.0, 2, false),
                          new Deportivo("4070 DEP",
                                         "Ford", "Mustang", "Rojo",
                                         150.0, 2000),
                          new Furgoneta("4080 FUR",
                                         "Fiat", "Ducato", "Azul",
                                         80.0, 1200, 8) };
```

Para mostrar los datos de los vehículos almacenados en el array se debe ejecutar el método `getAtributos()` de la clase `Vehiculo`. Se puede utilizar un `for` con una variable de control `i` o un `for` "para todo".

```
// for con variable de control i

for (int i=0; i < vehiculos.length; i++)
    System.out.println(vehiculos[i].getAtributos());

// for "para todo"

for (Vehiculo vehiculo : vehiculos)
    System.out.println(vehiculo.getAtributos());
```

La variable `vehiculo` del `for` "para todo" es de tipo `Vehiculo` y toma los valores de `vehiculos[0]`, `vehiculos [1]` y `vehiculos[2]` en cada iteración.

La salida por consola es la misma en ambos casos:

```
Matrícula: 4060 TUR Modelo: Skoda Fabia Color: Blanco
Tarifa: 90.0 Disponible: false Puertas: 2
Marcha automática: false
```

```
Matrícula: 4070 DEP Modelo: Ford Mustang Color: Rojo
Tarifa: 150.0 Disponible: false Cilindrada (cm3): 2000
```

```
Matrícula: 4080 FUR Modelo: Fiat Ducato Color: Azul
Tarifa: 80.0 Disponible: false Carga (kg): 1200
Volumen (m3): 8
```

## Arrays multidimensionales

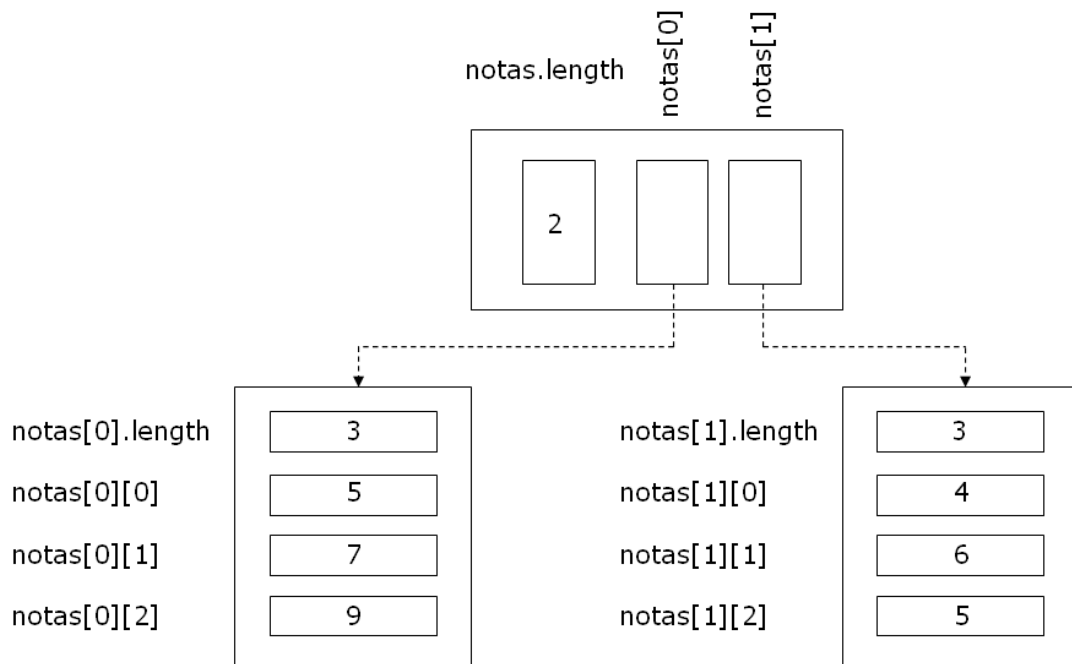
Un array de dos dimensiones es un array que contiene otro array en cada uno de sus elementos.

Por ejemplo, un array de dos dimensiones de números enteros de tamaño 2 x 3 se declara:

```
int[][] notas = new int[2][3];
```

Un array de dos dimensiones tiene forma de tabla. Para indicar la posición de uno de sus elementos es necesario indicar dos índices, uno para la fila y otro para la columna. La fila indica la posición en el primer array y la columna la posición en el segundo array.

El siguiente esquema muestra un array de dos dimensiones de números enteros. El array `notas` se inicializa con los valores `{{5, 7, 9}, {4, 6, 5}}`. Este array representa una tabla de 2 x 3 y en total almacena 6 números enteros. El array `notas` tiene dos elementos y cada uno de ellos es un array: `notas[0]` y `notas[1]`. El elemento `notas[0]` tiene a su vez 3 elementos: `notas[0][0]`, `notas[0][1]` y `notas[0][2]`. El elemento `notas[1]` tiene también 3 elementos: `notas[1][0]`, `notas[1][1]` y `notas[1][2]`.



Para mostrar los elementos del array `notas` es necesario utilizar dos `for` anidados.

```
for (int i = 0; i < notas.length; i++)
    for (int j = 0; j < notas[i].length; j++)
        System.out.println("notas[" + i + "][" + j + "] " +
            "es " + notas[i][j]);
```

La salida por la consola:

```
notas[0][0] es 5
notas[0][1] es 7
notas[0][2] es 9
notas[1][0] es 4
notas[1][1] es 6
notas[1][2] es 5
```

También se puede utilizar un `for` "para todo":

```
for (int[] fila : notas)
    for (int nota : fila)
        System.out.println(nota);
```

La salida por la consola:

```
5
7
9
4
6
5
```

En general, un array multidimensional es aquel que dentro de uno de sus elementos almacena uno o más arrays de objetos. Por ejemplo, un array de tres dimensiones de números enteros de tamaño 3 x 3 x 3 se declara:

```
int[][][] numeros = new int[3][3][3];
```

Este array de tres dimensiones tiene forma de cubo y almacena 27 números enteros. Para indicar la posición de uno de sus elementos es necesario utilizar tres índices. Si el array debe almacenar números del 1 al 27 consecutivos en cada una de sus dimensiones, entonces se inicializa utilizando tres `for` anidados:

```
int numero = 1;

for (int i=0; i<numeros.length; i++)
    for (int j=0; j<numeros[i].length; j++)
        for (int k=0; k<numeros[j].length; k++)
            numeros[i][j][k] = numero++;
```

Para mostrar los números almacenados:

```
for (int i=0; i<numeros.length; i++) {
    for (int j=0; j<numeros[i].length; j++) {
        for (int k=0; k<numeros[j].length; k++)
            System.out.print(numeros[i][j][k] + " ");
        System.out.println("");
    }

    System.out.println("");
}
```

La salida por la consola:

1 2 3

4 5 6

7 8 9

10 11 12

13 14 15

16 17 18

19 20 21

22 23 24

25 26 27



## Uso de arrays

Suponga que se desea almacenar en arrays el catálogo de vehículos y la relación de clientes de una empresa de alquiler de vehículos. Para esto es necesario definir una nueva clase, `EmpresaAlquilerVehiculos`, que almacene la información detallada de la empresa, los clientes y el catálogo de vehículos.

```
public class EmpresaAlquilerVehiculos {
    private String cif;
    private String nombre;
    private String paginaWeb;
    private int totalClientes;
    private Cliente[] clientes;
    private int totalVehiculos;
    private Vehiculo[] vehiculos;
    private int totalAlquileres;
    private VehiculoAlquilado[] alquileres;

    // se omiten los métodos 'get' y 'set' de la clase

    public EmpresaAlquilerVehiculos (String cif,
                                     String nombre,
                                     String paginaWeb) {

        this.cif = cif;
        this.nombre = nombre;
        this.paginaWeb = paginaWeb;
        this.totalClientes = 0;
        this.clientes = new Cliente[50];
        this.totalVehiculos = 0;
        this.vehiculos = new Vehiculo[50];
        this.totalAlquileres = 0;
        this.alquileres = new VehiculoAlquilado[100];
    }
}
```

## Estructuras de almacenamiento

---

La clase `EmpresaAlquilerVehiculos` está compuesta de arrays de objetos de las clases `Cliente`, `Vehiculo` y `VehiculoAlquilado`. A continuación se incluye la declaración de los atributos de estas clases, se omite la declaración de los métodos constructores y los métodos 'get' y 'set'.

```
public class Cliente {
    private String nif;
    private String nombre;
    private String apellidos;
}
```

```
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa;
    private boolean disponible;
}
```

```
public class VehiculoAlquilado {
    private Cliente cliente;
    private Vehiculo vehiculo;
    private int diaAlquiler;
    private int mesAlquiler;
    private int añoAlquiler;
    private int totalDiasAlquiler;
}
```

La clase `EmpresaAlquilerVehiculos` declara tres arrays para registrar a sus clientes, vehículos y el histórico de vehículos alquilados. El array `clientes` puede almacenar hasta 50 objetos de tipo `Cliente`, el array `vehiculos` puede almacenar hasta 50 objetos de tipo `Vehiculo` y el array `alquileres` tiene capacidad para almacenar 100 objetos de tipo `VehiculoAlquilado`.

Las variables numéricas `totalClientes`, `totalVehiculos` y `totalAlquileres` se utilizan para almacenar el total de objetos almacenados en cada uno de los arrays. Estas variables se inicializan a cero en el método constructor de la clase.

La clase `EmpresaAlquilerVehiculos` declara los siguientes métodos para realizar operaciones:

- `registrarCliente(Cliente cliente)`. Añade un nuevo cliente a la lista de clientes de la empresa.
- `registrarVehiculo(Vehiculo vehiculo)`. Añade un vehículo al catálogo de vehículos de la empresa.
- `imprimirClientes()`. Muestra la relación de clientes de la empresa.
- `imprimirVehiculos()`. Muestra el catálogo de vehículos de la empresa.
- `alquilarVehiculo(String matricula,String nif,int dias)`. Modifica la disponibilidad del vehículo para indicar que está alquilado y añade un objeto de tipo `VehiculoAlquilado` al array de vehículos alquilados. Este array almacena el cliente, el vehículo y los días de alquiler de cada vehículo alquilado.
- `recibirVehiculo(String matricula)`. Modifica la disponibilidad del vehículo para que se pueda alquilar de nuevo.

## Estructuras de almacenamiento

---

El método `registrarCliente(Cliente cliente)` almacena un objeto de la clase `Cliente` en la última posición del array `clientes`, dada por la variable `totalClientes` y a continuación incrementa la variable `totalClientes`.

```
public void registrarCliente(Cliente cliente) {
    this.clientes[this.totalClientes] = cliente;
    this.totalClientes++;
}
```

El método `registrarVehiculo(Vehiculo vehiculo)` almacena un objeto de la clase `Vehiculo` en la última posición del array `vehiculos`, dada por la variable `totalVehiculos` y a continuación incrementa la variable `totalVehiculos`.

```
public void registrarVehiculo(Vehiculo vehiculo) {
    this.vehiculos[this.totalVehiculos] = vehiculo;
    this.totalVehiculos++;
}
```

El método `imprimirClientes()` muestra la relación de clientes de la empresa de alquiler.

```
public void imprimirClientes() {
    System.out.println("NIF cliente\tNombre\n");

    for (int i=0; i<this.totalClientes; i++)
        System.out.println(clientes[i].getAtributos());
}
```

El método `imprimirVehiculos()` muestra el catálogo de vehículos de la empresa de alquiler. El método `getAtributosInforme()` muestra el detalle de atributos del vehículo.

```
public void imprimirVehiculos() {
    System.out.println("Matricula\tModelo  " +
                       "\tImporte Disponible\n");

    for (int i=0; i<this.totalVehiculos; i++)
        System.out.println(
            vehiculos[i].getAtributosInforme());
}
```

Para registrar el alquiler de un vehículo por un cliente se usa el método `alquilarVehiculo(String matricula, String nif, int dias)`. Este método modifica la disponibilidad del vehículo para indicar que está alquilado. El método `getClientes(String nif)` busca la referencia del cliente con el NIF dado en el array `clientes`. De forma similar, el método `getVehiculo(String matricula)` busca la referencia del vehículo con la matrícula dada en el array `vehiculos`. Una vez encontrado el vehículo con la matrícula indicada, se verifica si está disponible para alquilar y se modifica su disponibilidad. A continuación, almacena un objeto de tipo `VehiculoAlquilado` en el array `alquileres`. Este objeto relaciona un cliente, un vehículo, la fecha actual y los días de alquiler.

El método `getClientes(String nif)`.

```
private Cliente getClientes(String nif) {
    for (int i=0; i<this.getTotalClientes(); i++)
        if (this.clientes[i].getNIF() == nif)
            return this.clientes[i];

    return null;
}
```

```
public void alquilarVehiculo(String matricula,
                             String nif,
                             int dias) {
    Cliente cliente = getCliente(nif);
    Vehiculo vehiculo = getVehiculo(matricula);

    // busca el cliente con el NIF dado en el array
    // clientes y el vehículo con la matrícula dada en el
    // array vehiculos, si el vehículo está disponible se
    // alquila con la fecha actual, que se obtiene
    // ejecutando los métodos diaHoy(), mesHoy() y
    // añoHoy(), cuya declaración no se incluye

    if (vehiculo.getDisponible()) {
        vehiculo.setDisponible(false);
        this.alquileres[this.totalAlquileres]=
            new VehiculoAlquilado(cliente, vehiculo,
                                   diaHoy(), mesHoy(), añoHoy(), dias);
        this.totalAlquileres++;
    }
}
```

El método `recibirVehiculo(String matricula)` modifica la disponibilidad del vehículo para que se pueda alquilar de nuevo. Este método utiliza el método `getVehiculo(String matricula)` que busca el vehículo con la matrícula dada en el array `vehiculos`. Si lo encuentra, modifica su disponibilidad para indicar que nuevamente está disponible para alquiler.

```
public void recibirVehiculo(String matricula) {
    // busca el vehículo con la matrícula dada en el
    // array vehiculos y modifica su disponibilidad
    // para que se pueda alquilar de nuevo

    Vehiculo vehiculo = getVehiculo(matricula);

    if (vehiculo != null)
        vehiculo.setDisponible(true);
}
```

Una vez definida la clase `EmpresaAlquilerVehiculos` es necesario definir la clase del programa principal donde se van a crear las instancias de los objetos de esta aplicación. El método `main()` del programa principal crea una instancia de la clase `EmpresaAlquilerVehiculos`, denominada `easydrive` con CIF "A-28-187189", nombre "easydrive" y página web "www.easydrive.com".

```
// la instancia easydrive de EmpresaAlquilerVehiculos

EmpresaAlquilerVehiculos easydrive = new
    EmpresaAlquilerVehiculos("A-28-187189", "easy drive",
        "www.easydrive.com");
```

Al crear la instancia `easydrive`, el método constructor de la clase `EmpresaAlquilerVehiculos` inicializa los arrays `clientes` y `vehiculos` de este objeto. Una vez creada la instancia es necesario añadir clientes y vehículos al objeto `easydrive`. En este ejemplo se registran dos clientes y cinco vehículos de alquiler: tres turismos, un deportivo y una furgoneta.

Para registrar un nuevo cliente basta con invocar el método `registrarCliente(Cliente cliente)` con una instancia de la clase `Cliente` para añadir un nuevo cliente al array `clientes` del objeto `easydrive`.

## Estructuras de almacenamiento

---

```
// registro del cliente con NIF "X5618927C"
```

```
easydrive.registrarCliente(new Cliente("X5618927C",  
                                       "Juan", "González López"));
```

Para registrar un nuevo vehículo basta con invocar el método registrarVehiculo(Vehiculo vehiculo) con una instancia de la clase Vehiculo para añadir un nuevo vehículo al array vehiculos del objeto easydrive.

```
// registro del turismo con matrícula "4060 TUR"
```

```
easydrive.registrarVehiculo(new Turismo("4060 TUR",  
                                         "Skoda", "Fabia", "Blanco",  
                                         90.0, 2, false));
```

Una vez registrados los clientes y los vehículos de la empresa, se invocan los métodos imprimirClientes() e imprimirVehiculos() para mostrar la relación de clientes y el catálogo de vehículos de la empresa "easydrive".

```
// imprime la relación de clientes de "easydrive"
```

```
easydrive.imprimirClientes();
```

```
// imprime el catálogo de vehículos de "easydrive"
```

```
easydrive.imprimirVehiculos();
```



```
public class EmpresaAlquilerVehiculos {
    // se omiten los atributos y el resto de métodos de la clase

    public void registrarCliente(Cliente cliente) {
        this.clientes[this.totalClientes] = cliente;
        this.totalClientes++;
    }

    public void registrarVehiculo(Vehiculo vehiculo) {
        this.vehiculos[this.totalVehiculos] = vehiculo;
        this.totalVehiculos++;
    }

    public void imprimirClientes() {
        System.out.println("NIF cliente\tNombre\n");
        for (int i=0; i<this.totalClientes; i++)
            System.out.println(clientes[i].getAtributos());
    }

    public void imprimirVehiculos() {
        System.out.println("Matricula\tModelo  " +
            "\tImporte Disponible\n");
        for (int i=0; i<this.totalVehiculos; i++)
            System.out.println(vehiculos[i].getAtributosInforme());
    }

    public void alquilarVehiculo(String matricula,
                                String nif,
                                int dias) {
        Cliente cliente = getCliente(nif);
        Vehiculo vehiculo = getVehiculo(matricula);
        if (vehiculo.getDisponible()) {
            vehiculo.setDisponible(false);
            this.alquileres[this.totalAlquileres] =
                new VehiculoAlquilado(cliente, vehiculo,
                                       diaHoy(), mesHoy(), añoHoy(), dias);
            this.totalAlquileres ++;
        }
    }

    public void recibirVehiculo(String matricula) {
        Vehiculo vehiculo = getVehiculo(matricula);
        if (vehiculo != null)
            vehiculo.setDisponible(true);
    }
}
```

El programa principal de la aplicación.

```
public class MisVehiculos {
    public static void main(String[] args) {

        // la instancia easydrive de la clase EmpresaAlquilerVehiculos

        EmpresaAlquilerVehiculos easydrive = new
            EmpresaAlquilerVehiculos("A-28-187189", "easy drive",
                "www.easydrive.com");

        // registro de los clientes de la empresa

        easydrive.registrarCliente(new Cliente("X5618927C",
            "Juan", "González López"));
        easydrive.registrarCliente(new Cliente("Z7568991Y",
            "Luis", "Fernández Gómez"));

        // registro de los vehículos de la empresa

        easydrive.registrarVehiculo(new Turismo("4060 TUR", "Skoda",
            "Fabia", "Blanco", 90.0, 2, false));
        easydrive.registrarVehiculo(new Deportivo("4070 DEP", "Ford",
            "Mustang", "Rojo", 150.0, 2000));
        easydrive.registrarVehiculo(new Turismo("4080 TUR", "VW", "GTI",
            "Azul", 110.0, 2, false));
        easydrive.registrarVehiculo(new Turismo("4090 TUR", "SEAT",
            "Ibiza", "Blanco", 90.0, 4, false));
        easydrive.registrarVehiculo(new Furgoneta("4100 FUR", "Fiat",
            "Ducato", "Azul", 80.0, 1200, 8));

        // imprime la relación de clientes de easydrive

        easydrive.imprimirClientes();

        // imprime el catálogo de vehículos de easydrive

        easydrive.imprimirVehiculos();
    }
}
```

La salida por la consola muestra la relación de clientes y el catálogo de vehículos para alquiler:

La relación de clientes:

```
NIF cliente Nombre
X5618927C González López, Juan
Z7568991Y Fernández Gómez, Luis
```

El catálogo de vehículos:

Matrícula	Modelo	Color	Importe	Disponible
4060 TUR	Skoda Fabia	Blanco	90.0	true
4070 DEP	Ford Mustang	Rojo	150.0	true
4080 TUR	VW GTI	Azul	110.0	true
4090 TUR	SEAT Ibiza	Blanco	90.0	true
4100 TUR	Fiat Ducato	Azul	80.0	true

## Búsqueda binaria en arrays ordenados

Para buscar un elemento en un array ordenado se puede aplicar la técnica de la búsqueda binaria. El conjunto de búsqueda se delimita por dos posiciones: el límite inferior y el límite superior. El algoritmo empieza la búsqueda por el elemento que está almacenado en la mitad del conjunto de búsqueda. Si el elemento almacenado en la mitad del conjunto es mayor que el valor que se busca, entonces continúa la búsqueda en la primera mitad. Si el elemento almacenado en la mitad del conjunto es menor que el valor que se busca, entonces continúa la búsqueda en la segunda mitad. Si el elemento almacenado en la mitad del conjunto es igual que el valor que se busca, finaliza el proceso. En cada comparación, el algoritmo reduce el conjunto de búsqueda a la mitad. Si durante las sucesivas reducciones del conjunto de búsqueda el límite inferior es mayor que el límite superior, entonces el valor que se busca no está en el array y finaliza el proceso.

## Estructuras de almacenamiento

---

En este ejemplo el conjunto de búsqueda tiene 10 elementos, el límite inferior coincide con el primer elemento del array y el límite superior con el último elemento del array.

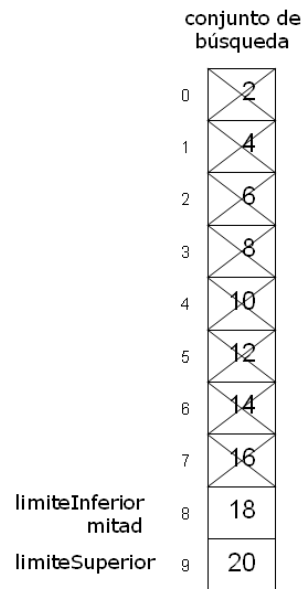
		conjunto de búsqueda
limiteInferior	0	2
	1	4
	2	6
	3	8
mitad	4	10
	5	12
	6	14
	7	16
	8	18
limiteSuperior	9	20

Si se aplica la búsqueda binaria para buscar el número 18, el algoritmo realiza las siguientes reducciones del conjunto de búsqueda.

Cuando se busca el número 18 en el array, en la primera iteración se compara el valor almacenado en la mitad con el 18. La mitad es la posición 4 y almacena un 10. Como 18 es mayor que 10, se descarta la primera mitad del conjunto de búsqueda y el límite inferior se hace igual a la mitad + 1. Ahora, el límite inferior es 5 y la nueva mitad es 7. Los valores del array que se han descartado en esta iteración se han tachado.

		conjunto de búsqueda
	0	<del>2</del>
	1	<del>4</del>
	2	<del>6</del>
	3	<del>8</del>
	4	<del>10</del>
limiteInferior	5	12
	6	14
mitad	7	16
	8	18
limiteSuperior	9	20

Una vez más, se compara el 18 con el valor almacenado en la mitad, que es 16. Como 18 es mayor que 16, se descarta la primera mitad del conjunto de búsqueda y el límite inferior se hace igual a la mitad + 1. Ahora, el límite inferior es 8 y la nueva mitad es 8. En la siguiente iteración se compara el valor almacenado en la posición central con el 18 y finaliza el algoritmo.



En este ejemplo, el algoritmo de búsqueda binaria ha realizado tres comparaciones para encontrar el número 18 en el array.

Durante el proceso de división del conjunto de búsqueda se modifica el valor del límite inferior o del límite superior, dependiendo de si el número que se busca está en la primera mitad o en la segunda mitad. Si durante este proceso el límite inferior es mayor que el límite superior, entonces el algoritmo finaliza porque el número que se busca no está en el array.

El siguiente programa utiliza el algoritmo de búsqueda binaria para buscar un número entre cero y 100 en un array de números ordenados.

```
public class BusquedaBinaria {
    public static void main(String[] args) {

        int[] numeros = {1,2,3,4,6,7,8,9,10,15,17,20,45,51,60,68,74,75};

        int mitad;
        int limiteInferior = 0;
        int limiteSuperior = numeros.length - 1;
        int numeroBusqueda = 68;
        boolean encontrado = false;

        while ((limiteInferior <= limiteSuperior) && (!encontrado)) {
            mitad = (limiteInferior + limiteSuperior) / 2;

            if (numeros[mitad] == numeroBusqueda) {
                encontrado = true;           // ¡encontrado!
            }
            else if (numeros[mitad] > numeroBusqueda) {
                limiteSuperior = mitad - 1; // buscar en la primera mitad
            } else {
                limiteInferior = mitad + 1; // buscar en la segunda mitad
            }
        }

        if (encontrado)
            System.out.println("He encontrado el número");
        else
            System.out.println("No he encontrado el número");

    }
}
```

## Ordenación de arrays

Una de las operaciones más comunes con arrays es la ordenación. Un algoritmo de ordenación clasifica un conjunto de datos de forma ascendente o descendente.

## El algoritmo de ordenación "Bubble Sort"

El algoritmo "Bubble Sort" se basa en comparar cada elemento del conjunto a ordenar con el siguiente. Si estos elementos no están ordenados, entonces se intercambian. En este algoritmo es necesario revisar varias veces todo el conjunto hasta que no sea necesario realizar más intercambios.

array de  
números

0	6
1	5
2	3
3	1
4	2

El algoritmo "Bubble Sort" ordena los valores almacenados en el array `numeros`.

```
int[] numeros = {6, 5, 3, 1, 2};

int tmp;

for (int i=0; i < numeros.length-1; i++)
    for (int j=i+1; j < numeros.length; j++)
        if (numeros[i] > numeros[j]) {
            tmp = numeros[i];
            numeros[i] = numeros[j];
            numeros[j] = tmp;
        }
```

La variable `tmp` se utiliza para realizar el intercambio de los valores almacenados en las posiciones `i` y `j` del array. Primero, se almacena el

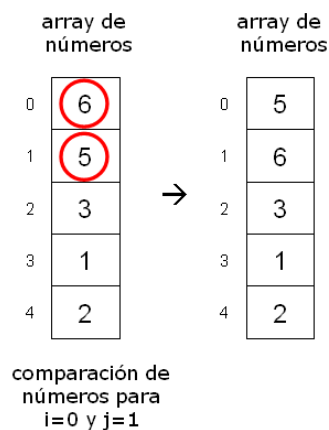
## Estructuras de almacenamiento

valor de `numeros[i]` en `tmp`, después se almacena el valor de `numeros[j]` en `numeros[i]`, por último se almacena el valor de `tmp` en `numeros[j]` y finaliza el intercambio.

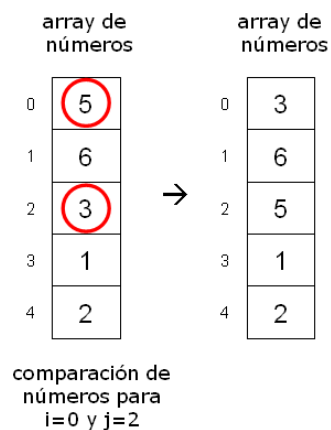
El algoritmo utiliza un `for` anidado. La variable de control del primer `for` es `i` y la del segundo `for` es `j`. La variable `i` del primer `for` toma los valores 0, 1, 2 y 3. Para `i = 0`, la variable `j` del segundo `for` toma los valores 1, 2, 3 y 4. Para `i = 1`, la variable `j` toma los valores 2, 3 y 4. Para `i = 2`, la variable `j` toma los valores 3, 4. Para `i = 3`, la variable `j` toma el valor de 4.

En diagramas se muestra el array de números antes y después de hacer el intercambio de los valores que se comparan cada vez.

Comparación de `numeros[0]` y `numeros[1]`.

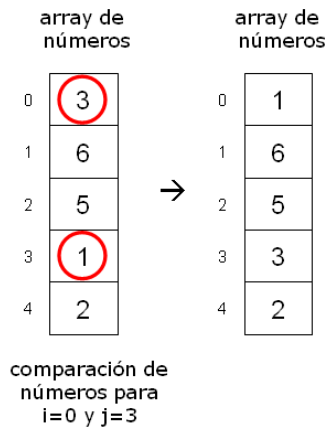


Comparación de `numeros[0]` y `numeros[2]`.

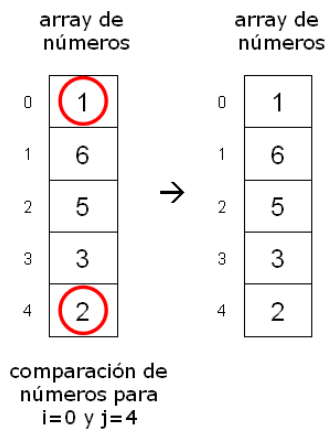




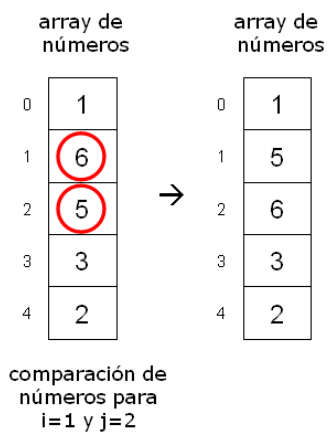
Comparación de `numeros[0]` y `numeros[3]`.



Comparación de `numeros[0]` y `numeros[4]`.



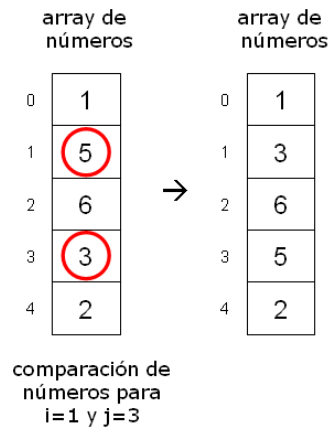
Comparación de `numeros[1]` y `numeros[2]`.



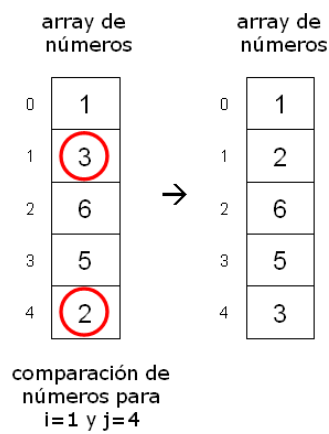
## Estructuras de almacenamiento

---

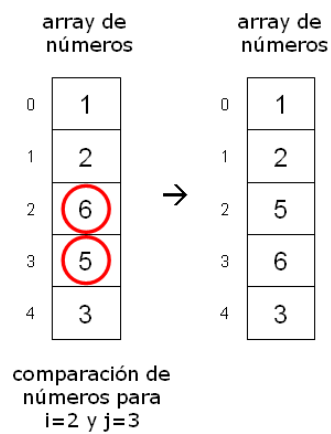
Comparación de `numeros[1]` y `numeros[3]`.



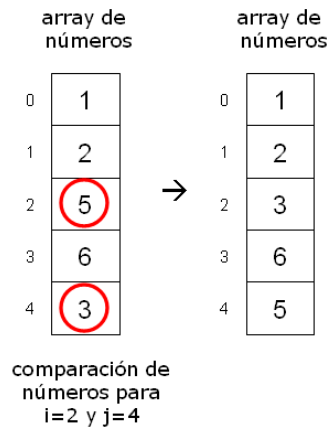
Comparación de `numeros[1]` y `numeros[4]`.



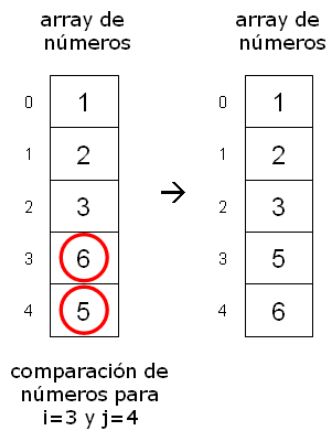
Comparación de `numeros[2]` y `numeros[3]`.



Comparación de `numeros[2]` y `numeros[4]`.



Comparación de `numeros[3]` y `numeros[4]`.



Al finalizar el algoritmo, el array ordenado es `{1, 2, 3, 5, 6}`.

## El método `sort` de la clase `Arrays`

Java define la clase `Arrays` con métodos que permiten realizar operaciones de ordenación y búsqueda en objetos de tipo array. Estos métodos se pueden utilizar con todos los tipos primitivos, `String` y con cualquier otro tipo de objeto.

Los métodos más utilizados de esta clase son:

- `Arrays.sort(array-de-datos)`. Ordena el contenido del array en orden ascendente.

`Arrays.sort(numeros)` ordena todos los elementos del array `numeros`.

- `Arrays.sort(array-de-datos, inicio, fin)`. Ordena el contenido del array en orden ascendente, desde la posición de inicial hasta la posición final.

`Arrays.sort(numeros, 0, 49)` ordena los elementos almacenados entre la posición 0 y la 49 del array `numeros`.

- `Arrays.binarySearch(array-de-datos, clave)`. Busca la clave indicada en el array de números enteros.

`Arrays.binarySearch(numeros, 1991)` busca el número 1991 en el array `numeros`.

- `Arrays.fill(array-de-datos, dato)`. Rellena el array con el valor dado. Se puede utilizar con todos los tipos primitivos, `String` y con cualquier otro tipo de objeto.

`Arrays.fill(numeros, 5)` rellena con el valor 5 todo el array `numeros`.

- `Arrays.fill(array-de-datos, dato, inicio, fin)`. Rellena el array con el valor dado, indicando la posición inicial y final.

`Arrays.fill(numeros, 5, 0, 5)` rellena con el valor 5 desde la posición 0 hasta la posición 5 del array `numeros`.

Programa que utiliza el método `sort` de la clase `Arrays` para ordenar ascendentemente un array de números enteros.

```
public class SortArray {
    public static void main(String[] args) {

        int[] numerosDesordenados = {5, 4, 6, 7, 5, 6, 4, 8, 7, 10};

        // numerosOrdenados es una copia de numerosDesordenados
        // el método clone() copia del objeto al que se aplica

        int[] numerosOrdenados = numerosDesordenados.clone();

        System.out.println("\nEl array de números desordenados\n");

        for (int numero : numerosDesordenados)
            System.out.println(numero);

        // el método sort(array-de-datos) de la clase Array ordena
        // ascendentemente todos los elementos de numerosOrdenados

        Arrays.sort(numerosOrdenados);

        System.out.println("\nEl array de números ordenados\n");

        for (int numero : numerosOrdenados)
            System.out.println(numero);
    }
}
```

La salida por la consola:

El array de números desordenados

5

4

6

7

5

6

4

8

7

10

El array de números ordenados

4

4

5

5

6

6

7

7

8

10

## Arrays redimensionables

Un `ArrayList` es un array redimensionable. Puede almacenar un número indefinido de elementos.

La declaración de un `ArrayList` tiene la siguiente sintaxis:

```
List <tipo-o-clase> identificador-lista;
```

Por ejemplo, el `ArrayList` `vehiculos` de la clase `Vehiculo` se inicializa:

```
List <Vehiculo> vehiculos = new ArrayList<Vehiculo>();
```

Los métodos más utilizados de la clase `ArrayList` son:

- `add(Object o)`. Añade un objeto a la lista.

```
vehiculos.add(new Turismo("4060 TUR", "Skoda", "Fabia",  
                          "Blanco", 90.0, 2, false));
```

- `get(int posicion)`. Extrae el objeto almacenado en la posición indicada. Es necesario indicar el tipo del objeto que se extrae.

```
Vehiculo v = (Vehiculo)vehiculos.get(0);
```

- `size()`. Devuelve el número de electos almacenados en la lista.

```
int totalVehiculos = vehiculos.size();
```

Para mostrar los elementos almacenados en una lista se puede utilizar un `for` con una variable de control o un `for` 'para todo'.

```
// recorrido de un ArrayList for con variable de control  
  
for(int i=0; i < vehiculos.size(); i++) {  
    Vehiculo v = (Vehiculo)vehiculos.get(i);  
    System.out.println(v.getAtributos());  
}
```

```
// recorrido de un ArrayList for con variable de control

for (Vehiculo v : vehiculos)
    System.out.println(v.getAtributos());
```

El método `add(Object o)` almacena objetos en un `ArrayList`:

```
List <Vehiculo> vehiculos = new ArrayList<Vehiculo>();
vehiculos.add(new Turismo("4060 TUR", "Skoda", "Fabia",
                          "Blanco", 90.0, 2, false));
vehiculos.add(new Deportivo("4070 DEP", "Ford", "Mustang",
                             "Rojo", 150.0, 2000));
vehiculos.add(new Turismo("4080 TUR", "VW", "GTI",
                           "Azul", 110.0, 2, false));
vehiculos.add(new Turismo("4090 TUR", "SEAT", "Ibiza",
                           "Blanco", 90.0, 4, false));
vehiculos.add(new Furgoneta("4100 FUR", "Fiat", "Ducato",
                             "Azul", 80.0, 1200, 8));
```

El `ArrayList vehiculos` almacena objetos de la clase `Vehiculo`. Para añadir un nuevo elemento se ejecuta el método `add(Object o)` con un objeto de tipo `Vehiculo`. En este ejemplo se almacenan instancias de las clases `Turismo`, `Deportivo` y `Furgoneta`, todas ellas subclases de `Vehiculo`.

El método `size()` devuelve el número de objetos almacenados.

```
int totalVehiculos = vehiculos.size();
```

Para más información sobre los métodos de la clase `ArrayList`, consulte el API de Java.

<http://docs.oracle.com/javase/1.4.2/docs/api/java/util/ArrayList.html>



Programa que define un `ArrayList` para almacenar objetos de la clase `Vehiculo` y muestra su contenido por la consola. Para utilizar la clase `ArrayList` es necesario importar los paquetes:

```
import java.util.ArrayList;
import java.util.List;
```

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListVehiculos {
    public static void main(String[] args) {

        // se declara e inicialización el ArrayList

        List <Vehiculo> vehiculos = new ArrayList<Vehiculo>();

        // se añaden objetos de la clase Vehiculo

        vehiculos.add(new Turismo("4060 TUR", "Skoda", "Fabia",
            "Blanco", 90.0, 2, false));
        vehiculos.add(new Deportivo("4070 DEP", "Ford", "Mustang",
            "Rojo", 150.0, 2000));
        vehiculos.add(new Turismo("4080 TUR", "VW", "GTI", "Azul",
            110.0, 2, false));
        vehiculos.add(new Turismo("4090 TUR", "SEAT", "Ibiza", "Blanco",
            90.0, 4, false));
        vehiculos.add(new Furgoneta("4100 FUR", "Fiat", "Ducato",
            "Azul", 80.0, 1200, 8));

        // muestra el contenido del ArrayList

        System.out.println("Vehiculos");

        for (Vehiculo v: vehiculos)
            System.out.println(v.getAtributos());
    }
}
```

La salida por la consola:

Vehiculos

```
Matrícula: 4060 TUR Modelo: Skoda Fabia Color: Blanco
Tarifa: 90.0 Disponible: true Puertas: 2 Marcha
automática: false
```

```
Matrícula: 4070 DEP Modelo: Ford Mustang Color: Rojo
Tarifa: 150.0 Disponible: true Cilindrada (cm3): 2000
```

```
Matrícula: 4080 TUR Modelo: VW GTI Color: Azul
Tarifa: 110.0 Disponible: true Puertas: 2 Marcha
automática: false
```

```
Matrícula: 4090 TUR Modelo: SEAT Ibiza Color: Blanco
Tarifa: 90.0 Disponible: true Puertas: 4 Marcha
automática: false
```

```
Matrícula: 4100 FUR Modelo: Fiat Ducato Color: Azul
Tarifa: 80.0 Disponible: true Carga (kg): 1200
Volumen (m3): 8
```

## Uso de arrays redimensionables

Suponga que se desea almacenar en arrays redimensionables el catálogo de vehículos y la relación de clientes de la empresa de alquiler de vehículos. Basta con sustituir los arrays de la clase `EmpresaAlquilerVehiculos` por objetos de tipo `ArrayList` y modificar los métodos de la clase que utilizan los objetos clientes, vehículos y alquileres.

En esta nueva implementación de la clase, no es necesario definir variables para almacenar el total de clientes, vehículos o vehículos alquilados. Un `ArrayList` es un array que modifica su tamaño de forma dinámica para almacenar tantos elementos como sea necesario.

La nueva clase `EmpresaAlquilerVehiculos` declara los objetos clientes, vehículos y alquileres de tipo `ArrayList` para almacenar la información de los clientes, el catálogo de vehículos y el histórico de vehículos alquilados, respectivamente.

```
import java.util.ArrayList;
import java.util.List;

public class EmpresaAlquilerVehiculos {
    private String cif;
    private String nombre;
    private String paginaWeb;
    private List <Cliente> clientes;
    private List <Vehiculo> vehiculos;
    private List <VehiculoAlquilado> alquileres;

    // se omiten los métodos 'get' y 'set' de la clase

    public EmpresaAlquilerVehiculos (String cif,
                                     String nombre,
                                     String paginaWeb) {

        this.cif = cif;
        this.nombre = nombre;
        this.paginaWeb = paginaWeb;
        this.clientes = new ArrayList<Cliente>();
        this.vehiculos = new ArrayList<Vehiculo>();
        this.alquileres = new ArrayList<VehiculoAlquilado>();
    }
}
```

La clase `EmpresaAlquilerVehiculos` declara los siguientes métodos para realizar operaciones:

- `registrarCliente(Cliente cliente)`. Añade un nuevo cliente a la lista de clientes de la empresa.
- `registrarVehiculo(Vehiculo vehiculo)`. Añade un vehículo al catálogo de vehículos de la empresa.
- `imprimirClientes()`. Muestra la relación de clientes de la empresa.
- `imprimirVehiculos()`. Muestra el catálogo de vehículos de la empresa.

- `alquilarVehiculo(String matricula,String nif,int dias)`. Modifica la disponibilidad del vehículo para indicar que está alquilado y añade un objeto de tipo `VehiculoAlquilado` a la lista de vehículos alquilados. Esta lista almacena el cliente, el vehículo y los días de alquiler de cada vehículo alquilado.
- `recibirVehiculo(String matricula)`. Modifica la disponibilidad del vehículo para que se pueda alquilar de nuevo.

El método `registrarCliente(Cliente cliente)` añade un objeto de la clase `Cliente` a la lista `clientes`.

```
public void registrarCliente(Cliente cliente) {
    this.clientes.add(cliente);
}
```

El método `registrarVehiculo(Vehiculo vehiculo)` añade un objeto de la clase `Vehiculo` a lista `vehiculos`.

```
public void registrarVehiculo(Vehiculo vehiculo) {
    this.vehiculos.add(vehiculo);
}
```

El método `imprimirClientes()` muestra la relación de clientes de la empresa de alquiler.

```
public void imprimirClientes() {
    System.out.println("NIF cliente\tNombre\n");

    for (Cliente c : this.clientes)
        System.out.println(c.getAtributos());
}
```

El método `imprimirVehiculos()` muestra el catálogo de vehículos de la empresa de alquiler. El método `getAtributosInforme()` muestra el detalle de atributos del vehículo.

```
public void imprimirVehiculos() {
    System.out.println("Matricula\tModelo  " +
                       "\tImporte Disponible\n");
    for (Vehiculo v : this.vehiculos)
        System.out.println(v.getAtributosInforme());
}
```

Para registrar el alquiler de un vehículo por un cliente se usa el método `alquilarVehiculo(String matricula, String nif, int dias)`. Este método modifica la disponibilidad del vehículo para indicar que está alquilado. El método `getClientes(String nif)` busca la referencia del cliente con el NIF dado en la lista `clientes`. De forma similar, el método `getVehiculo(String matricula)` busca la referencia del vehículo con la matrícula dada en la lista `vehiculos`. Una vez encontrado el vehículo con la matrícula indicada, se verifica si está disponible para alquilar y se modifica su disponibilidad. A continuación, almacena un objeto de tipo `VehiculoAlquilado` en la lista `alquileres`. Este objeto relaciona un cliente, un vehículo, la fecha actual y los días de alquiler.

El método `getClientes(String nif)`.

```
private Cliente getCliente(String nif) {
    for (Cliente c : this.clientes)
        if (c.getNIF() == nif)
            return c;

    return null;
}
```

```
public void alquilarVehiculo(String matricula,
                             String nif,
                             int dias) {
    Cliente cliente = getCliente(nif);
    Vehiculo vehiculo = getVehiculo(matricula);

    if (vehiculo.getDisponible()) {
        vehiculo.setDisponible(false);
        this.alquileres.add(
            new VehiculoAlquilado(cliente, vehiculo,
                                   diaHoy(), mesHoy(), añoHoy(), dias));
    }
}
```

El método `recibirVehiculo(String matricula)` modifica la disponibilidad del vehículo para que se pueda alquilar de nuevo. Este método utiliza el método `getVehiculo(String matricula)` que busca el vehículo con la matrícula dada en la lista `vehiculos`. Si lo encuentra, modifica su disponibilidad para indicar que nuevamente está disponible para alquiler.

```
public void recibirVehiculo(String matricula) {
    Vehiculo vehiculo = getVehiculo(matricula);

    if (vehiculo != null)
        vehiculo.setDisponible(true);
}
```

Una vez modificada la clase `EmpresaAlquilerVehiculos`, es importante ver que los cambios realizados a las variables privadas de la clase solo afectan a la implementación de los métodos de esta clase. Las modificaciones quedan "encapsuladas" dentro de la clase y no afectan a otras clases o programas.

Esto significa que no es necesario modificar la clase `MisVehiculos` que registra los clientes y los vehículos de la empresa de alquiler. El método `main()` crea una instancia de la clase `EmpresaAlquilerVehiculos`, denominada `easydrive` con CIF "A-28-187189", nombre "easydrive" y página web "www.easydrive.com".

```
// la instancia easydrive de EmpresaAlquilerVehiculos

EmpresaAlquilerVehiculos easydrive = new
    EmpresaAlquilerVehiculos("A-28-187189", "easy drive",
                              "www.easydrive.com");
```

Al crear la instancia `easydrive`, el método constructor de la clase `EmpresaAlquilerVehiculos` inicializa las listas `clientes` y `vehiculos` de este objeto. Una vez creada la instancia es necesario añadir clientes y vehículos al objeto `easydrive`. En este ejemplo se registran dos clientes y cinco vehículos de alquiler: tres turismos, un deportivo y una furgoneta.

Para registrar un nuevo cliente basta con invocar el método `registrarCliente(Cliente cliente)` con una instancia de la clase `Cliente` para añadir un nuevo cliente a la lista `clientes` del objeto `easydrive`.

```
// registro del cliente con NIF "X5618927C"

easydrive.registrarCliente(new Cliente("X5618927C",
                                         "Juan", "González López"));
```

## Estructuras de almacenamiento

---

Para registrar un nuevo vehículo basta con invocar el método `registrarVehiculo(Vehiculo vehiculo)` con una instancia de la clase `Vehiculo` para añadir un nuevo vehículo a la lista `vehiculos` del objeto `easydrive`.

```
// registro del turismo con matrícula "4060 TUR"

easydrive.registrarVehiculo(new Turismo("4060 TUR",
                                         "Skoda", "Fabia", "Blanco",
                                         90.0, 2, false));
```

Una vez registrados los clientes y los vehículos de la empresa, se invocan los métodos `imprimirClientes()` e `imprimirVehiculos()` para mostrar la relación de clientes y el catálogo de vehículos de la empresa "easydrive".

```
// imprime la relación de clientes de "easydrive"

easydrive.imprimirClientes();

// imprime el catálogo de vehículos de "easydrive"

easydrive.imprimirVehiculos();
```



```
import java.util.ArrayList;
import java.util.List;

public class EmpresaAlquilerVehiculos {
    // se omiten los atributos y el resto de métodos de la clase

    public void registrarCliente(Cliente cliente) {
        this.clientes.add(cliente);
    }

    public void registrarVehiculo(Vehiculo vehiculo) {
        this.vehiculos.add(vehiculo);
    }

    public void imprimirClientes() {
        System.out.println("NIF cliente\tNombre\n");
        for (Cliente c : this.clientes)
            System.out.println(c.getAtributos());
    }

    public void imprimirVehiculos() {
        System.out.println("Matricula\tModelo " +
            "\tImporte Disponible\n");
        for (Vehiculo v : this.vehiculos)
            System.out.println(v.getAtributosInforme());
    }

    public void alquilarVehiculo(String matricula,
        String nif,
        int dias) {
        Cliente cliente = getCliente(nif);
        Vehiculo vehiculo = getVehiculo(matricula);
        if (vehiculo.getDisponible()) {
            vehiculo.setDisponible(false);
            this.alquileres.add(
                new VehiculoAlquilado(cliente, vehiculo,
                    diaHoy(), mesHoy(), añoHoy(), dias));
        }
    }

    public void recibirVehiculo(String matricula) {
        Vehiculo vehiculo = getVehiculo(matricula);
        if (vehiculo != null)
            vehiculo.setDisponible(true);
    }
}
```

El programa principal de la aplicación.

```
public class MisVehiculos {
    public static void main(String[] args) {

        // la instancia easydrive de la clase EmpresaAlquilerVehiculos

        EmpresaAlquilerVehiculos easydrive = new
            EmpresaAlquilerVehiculos("A-28-187189", "easy drive",
                "www.easydrive.com");

        // registro de los clientes de la empresa

        easydrive.registrarCliente(new Cliente("X5618927C",
            "Juan", "González López"));
        easydrive.registrarCliente(new Cliente("Z7568991Y",
            "Luis", "Fernández Gómez"));

        // registro de los vehículos de la empresa

        easydrive.registrarVehiculo(new Turismo("4060 TUR", "Skoda",
            "Fabia", "Blanco", 90.0, 2, false));
        easydrive.registrarVehiculo(new Deportivo("4070 DEP", "Ford",
            "Mustang", "Rojo", 150.0, 2000));
        easydrive.registrarVehiculo(new Turismo("4080 TUR", "VW", "GTI",
            "Azul", 110.0, 2, false));
        easydrive.registrarVehiculo(new Turismo("4090 TUR", "SEAT",
            "Ibiza", "Blanco", 90.0, 4, false));
        easydrive.registrarVehiculo(new Furgoneta("4100 FUR", "Fiat",
            "Ducato", "Azul", 80.0, 1200, 8));

        // imprime la relación de clientes de easydrive

        easydrive.imprimirClientes();

        // imprime el catálogo de vehículos de easydrive

        easydrive.imprimirVehiculos();
    }
}
```

## 8. Entrada y salida

---

### Los flujos de Java

Prácticamente todos los programas deben leer datos del exterior para procesarlos y después presentar los resultados. La información que necesita un programa normalmente se obtiene mediante la entrada de datos por el teclado o leyendo un fichero. Los resultados de la ejecución de un programa se pueden presentar por la consola, la impresora o en un fichero. El tipo de información que se utiliza tanto en las entradas como en las salidas puede tener diversos formatos: texto, imagen, sonido, binario, etc.

En Java, la entrada de datos se realiza mediante un flujo de entrada. Para realizar la entrada de datos es necesario abrir el flujo de entrada, leer la información del flujo hasta el final y por último cerrar el flujo. La salida se hace mediante un flujo de salida. Para realizar la salida de datos es necesario abrir el flujo de salida y a continuación se escribe en él toda la información que se desee, por último, se cierra el flujo.

Este esquema de entradas y salidas basadas en un flujo permite que las entradas sean independientes de la fuente de datos y que las salidas sean independientes del destino de los datos.

Un flujo en Java es un objeto que se utiliza para realizar una entrada o salida de datos. Representa un canal de información del que se puede leer o escribir datos de forma secuencial. Existen dos tipos de flujos en Java, los que utilizan bytes y los que utilizan caracteres.

## Entrada y salida

---

La siguiente tabla muestra los flujos de entrada de datos de Java:

Flujos con bytes	Flujos con caracteres
InputStream ByteArrayInputStream FileInputStream FilterInputStream BufferedInputStream DataInputStream LineNumberInputStream PushBackInputStream ObjectInputStream PipedInputStream SequenceInputStream StringBufferInputStream	Reader BufferedReader LineNumberReader CharArrayReader FilterReader PushBackReader InputStreamReader FileReader PipedReader StringReader

La siguiente tabla muestra los flujos de salida de datos de Java:

Flujos con bytes	Flujos con caracteres
OutputStream ByteArrayOutputStream FileOutputStream FilterOutputStream BufferedOutputStream DataOutputStream PrintStream ObjectOutputStream PipedOutputStream	Writer BufferedWriter CharArrayWriter FilterWriter OutputStreamWriter FileWriter PipedWriter PrintWriter StringWriter

Existen flujos con bytes y flujos con caracteres que se aplican a la misma entrada o salida. `FileInputStream` y `FileOutputStream` son flujos para leer y escribir bytes en un fichero, `FileReader` y `FileWriter` también son flujos que se aplican a ficheros, pero en este caso para leer y escribir caracteres.

Los flujos se pueden utilizar solos o combinados. Si se combinan dos flujos, por ejemplo uno que lea caracteres de un archivo con otro que convierta a mayúsculas los caracteres, entonces el resultado final es un flujo del que se leen caracteres en mayúsculas.

En los siguientes ejemplos se muestran las aplicaciones de algunos de estos flujos para leer datos del teclado, leer y escribir en ficheros de texto o leer y escribir ficheros de objetos.

### Entrada de datos desde el teclado

El flujo de entrada `System.in` lee los datos que se introducen en el teclado. Si este flujo se pasa como argumento a una instancia de la clase `Scanner`, permite realizar la lectura de datos del teclado. Basta con utilizar los métodos `next()`, `nextLine()` y `nextInt()` para leer una palabra, una línea y un número entero, respectivamente.

A continuación se muestra el uso de la clase `Scanner` y el flujo de entrada del teclado `System.in`. Para utilizar la clase `Scanner` en un programa Java es necesario importar la librería `java.util.Scanner`.

Suponga que se desea realizar un programa que pide un nombre, el día, el mes y el año de la fecha de nacimiento de una persona. En este programa se declara una instancia de la clase `Scanner`, con nombre `entradaTeclado` que se inicializa con el flujo de entrada del teclado `System.in`.

```
Scanner entradaTeclado = new Scanner(System.in);
```

La clase `Scanner` ofrece los métodos `next()`, `nextLine()` o `nextInt()`, entre otros. El método `next()` lee una palabra, el método `nextLine()` lee una línea completa y el método `nextInt()` lee un número de tipo entero.

La variable `nombre` de tipo `String` se inicializa con el nombre que se introduce en el teclado. Como se invoca el método `nextLine()` de `entradaTeclado`, se lee toda la línea, de manera que la variable `nombre` almacena tanto el nombre como los apellidos de la persona.

```
System.out.print("¿Cómo te llamas? ");  
nombre = entradaTeclado.nextLine();
```

## Entrada y salida

---

Las variables numéricas de tipo `int` `diaNacimiento`, `mesNacimiento` y `añoNacimiento` se inicializan con el valor numérico introducido, invocando el método `nextInt()` de `entradaTeclado`.

```
System.out.print("¿Qué día naciste?      ");  
diaNacimiento = entradaTeclado.nextInt();
```

```
import java.util.Scanner;  
  
public class EntradaDatosTeclado {  
  
    public static void main(String args[]) {  
        String nombre;  
        int diaNacimiento, mesNacimiento, añoNacimiento;  
  
        // inicialización de la instancia de Scanner con el flujo de  
        // entrada del teclado  
  
        Scanner entradaTeclado = new Scanner(System.in);  
  
        System.out.print("¿Cómo te llamas?      ");  
        nombre = entradaTeclado.nextLine();  
  
        System.out.print("¿Qué día naciste?      ");  
        diaNacimiento = entradaTeclado.nextInt();  
  
        System.out.print("¿En qué mes?          ");  
        mesNacimiento = entradaTeclado.nextInt();  
  
        System.out.print("¿En qué año?          ");  
        añoNacimiento = entradaTeclado.nextInt();  
  
        System.out.println("Hola " + nombre + ", naciste el " +  
            diaNacimiento + "/" + mesNacimiento + "/" + añoNacimiento);  
    }  
}
```

Una vez introducidos todos los datos, el programa muestra un mensaje por la consola con el nombre y fecha de nacimiento.

```
Hola Juan, naciste el 10/12/1982
```

## Leer y escribir en ficheros de texto

La lectura y escritura de datos en un fichero de texto requiere el uso de las clases `PrintWriter`, `File` y `Scanner`. Para escribir en un fichero de texto es necesario utilizar la clase `PrintWriter`. Esta clase permite crear un fichero de texto para almacenar datos. Esta clase ofrece los métodos `print()` y `println()` para escribir datos en el fichero. El método `close()` cierra el fichero de datos.

Por ejemplo, para escribir un array de números enteros en un fichero de texto, es necesario crear una instancia de la clase `PrintWriter`.

```
int[][] numeros = { { 1, 2, 3, 4, 5},
                    { 6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15},
                    {16, 17, 18, 19, 20},
                    {21, 22, 23, 24, 25}};

// la instancia ficheroSalida de la clase PrintWriter
// crea y escribe en el fichero "c:\\Numeros.txt"

String idFichero = "c:\\Numeros.txt";
PrintWriter ficheroSalida = new PrintWriter(idFichero);

// el for anidado escribe en ficheroSalida los elementos
// del array separados por el carácter ","

for (int i=0; i<numeros.length; i++) {
    for (int j=0; j<numeros[i].length; j++)
        ficheroSalida.print(numeros[i][j] + ",");
    ficheroSalida.println("");
}

ficheroSalida.close();
```

El `for` anidado escribe todos los elementos del array en el fichero de texto. Al finalizar se invoca al método `close()` del objeto `ficheroSalida` para cerrarlo.

El contenido del fichero de texto `Numeros.txt`.

```
1,2,3,4,5,
6,7,8,9,10,
11,12,13,14,15,
16,17,18,19,20,
21,22,23,24,25,
```

La clase `File` ofrece el método `exists()` para saber si un fichero existe o no. Para leer el contenido del fichero se crea una instancia de la clase `Scanner` que recibe como argumento la instancia del fichero de texto.

La clase `Scanner` ofrece el método `hasNext()` para saber si hay más elementos que leer y `next()` para leer el siguiente elemento.

La clase `StringTokenizer` facilita la división de una cadena de texto en componentes separados por espacios o por un carácter delimitador. En este ejemplo se crea una instancia con el delimitador `","`.

Cuando se realiza una operación de lectura o escritura de un fichero es necesario gestionar los errores en tiempo de ejecución de Java, denominados excepciones. Durante la lectura o escritura de un fichero se puede producir una excepción de tipo `IOException`.

Para utilizar las clases `PrintWriter`, `File`, `Scanner`, `StringTokenizer`, `IOException` es necesario importar los siguientes paquetes:

```
import java.io.PrintWriter;
import java.io.File;
import java.util.Scanner;
import java.util.StringTokenizer;
import java.io.IOException;
```



Para abrir un fichero de texto se crea una instancia de la clase `File`. El contenido del fichero se lee con una instancia de la clase `Scanner`. Como los números almacenados en el fichero están separados por comas, se utiliza una instancia de `StringTokenizer` para leer cada número.

```
String idFichero = "c:\\Numeros.txt";

File ficheroEntrada=new File (idFichero);

if (ficheroEntrada.exists()) {
    Scanner datosFichero = new Scanner(ficheroEntrada);

    System.out.println("Números del fichero");

    while (datosFichero.hasNext()) {
        StringTokenizer numerosFichero = new
            StringTokenizer(datosFichero.next(),"");

        while (numerosFichero.hasMoreTokens())
            System.out.print(numerosFichero.nextToken() +
                "\t");

        System.out.println("");
    }

    datosFichero.close();
}

else
    System.out.println(";El fichero no existe!");
```

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import java.io.PrintWriter;

public class LecturaEscrituraFichero {
    public static void main(String[] args) throws IOException {
        int[][] numeros = { { 1, 2, 3, 4, 5},
                            { 6, 7, 8, 9, 10},
                            {11, 12, 13, 14, 15},
                            {16, 17, 18, 19, 20},
                            {21, 22, 23, 24, 25}};

        String idFichero = "c:\\\\Numeros.txt";

        PrintWriter ficheroSalida = new PrintWriter(idFichero);

        for (int i=0; i<numeros.length; i++) {
            for (int j=0; j<numeros[i].length; j++)
                ficheroSalida.print(numeros[i][j] + ",");
            ficheroSalida.println("");
        }
        ficheroSalida.close();

        File ficheroEntrada = new File (idFichero);

        if (ficheroEntrada.exists()) {
            Scanner datosFichero = new Scanner(ficheroEntrada);
            System.out.println("Números del fichero");
            while (datosFichero.hasNext()) {
                StringTokenizer numerosFichero = new
                    StringTokenizer(datosFichero.next(),",");
                while (numerosFichero.hasMoreTokens())
                    System.out.print(numerosFichero.nextToken() + "\\t");
                System.out.println("");
            }
            datosFichero.close();
        }
        else
            System.out.println(";El fichero no existe!");
    }
}
```

El resultado de leer el fichero de texto:

Números del fichero

```
1      2      3      4      5
6      7      8      9     10
11     12           13    14    15
16     17           18    19    20
21     22           23    24    25
```

## Leer y escribir objetos en ficheros

La lectura y escritura de objetos en un fichero binario requiere el uso de las clases `FileOutputStream` y `ObjectOutputStream`. La clase `ObjectOutputStream` permite escribir objetos en un fichero utilizando el método `writeObject(Object o)`. Los datos que se almacenan en el fichero de salida tienen un formato binario distinto de los ficheros de texto. El método `close()` de la clase `ObjectOutputStream` cierra el fichero de datos.

Para escribir objetos utilizando la clase `ObjectOutputStream` es necesario codificar los objetos dentro del flujo de salida. A la codificación de los objetos dentro de un flujo de entrada o salida se le denomina "serialización". Para que los objetos de una clase sean "serializables" es necesario implementar la interfaz `Serializable` de Java.

Para leer objetos almacenados en un fichero binario se utiliza el método `readObject()` de la clase `ObjectInputStream`. Después de leer el objeto del fichero se debe convertir a la clase a la que pertenece.

Cuando se utilizan los flujos `ObjectInputStream` y `ObjectOutputStream` de Java es necesario atrapar los errores de ejecución que se producen mientras se lee o escribe el fichero de datos con los flujos de entrada y salida. Para atrapar los errores de ejecución o excepciones, se utilizan las sentencias `try` y `catch`. Las excepciones que se producen durante la ejecución de las sentencias definidas en el cuerpo del `try` se atrapan con `catch`.

Para escribir un objeto de la clase `Persona` en un fichero binario, es necesario indicar que esta clase es "serializable", es decir, que sus objetos

se codifican dentro de los flujos de entrada y salida de Java. Para indicar que la clase `Persona` es "serializable" se modifica su declaración indicando que implementa la interfaz `Serializable`.

```
public class Persona implements java.io.Serializable {  
  
}
```

La declaración completa de la clase `Persona`, con la implementación de la interfaz `Serializable`, su método constructor y sus métodos 'get' y 'set'.

```
public class Persona implements java.io.Serializable {  
    private String dni;  
    private String nombre;  
    private String apellidos;  
  
    public Persona(String dni, String nombre, String apellidos) {  
        this.dni = dni;  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
    }  
  
    public String getDNI() {  
        return this.dni;  
    }  
  
    public String getNombre() {  
        return this.nombre;  
    }  
  
    public String getApellidos() {  
        return this.apellidos;  
    }  
  
    public String getAtributos() {  
        return this.getDNI() + " " + this.getApellidos() + ", " +  
            this.getNombre();  
    }  
}
```

Para escribir un objeto de la clase `Persona` en un fichero de texto se utilizan las clases `FileOutputStream` y `ObjectOutputStream`. Se crea una instancia de la clase `FileOutputStream` para inicializar la instancia `objetoSalida` de `ObjectOutputStream` que escribe en el fichero binario `Objetos.dat`.

```
String nombreFichero = "c:\\\\Objetos.dat";

try {
    FileOutputStream ficheroSalida = new
        FileOutputStream(nombreFichero);
    ObjectOutputStream objetoSalida = new
        ObjectOutputStream(ficheroSalida);

    // se escriben dos objetos de la clase Persona

    objetoSalida.writeObject(new Persona("55287188B",
        "María", "Ruiz Ramos"));
    objetoSalida.writeObject(new Persona("40302010A",
        "Juan", "González López"));

    objetoSalida.close();
} catch (FileNotFoundException e) {
    System.out.println("¡El fichero no existe!");
} catch (IOException e) {
    System.out.println(e.getMessage());
} catch (Exception e) {
    System.out.println(e.getMessage());
};
```

Para leer los objetos almacenados en el fichero binario se utilizan las clases `FileInputStream` y `ObjectInputStream`. Se crea una instancia de la clase `FileInputStream` para inicializar la instancia objetoEntrada de `ObjectInputStream`.

```
try {
    FileInputStream ficheroEntrada = new
        FileInputStream(nombreFichero);
    ObjectInputStream objetoEntrada = new
        ObjectInputStream(ficheroEntrada);

    // se leen dos objetos de la clase Persona

    Persona p1 = (Persona)objetoEntrada.readObject();
    Persona p2 = (Persona)objetoEntrada.readObject();

    // se cierra el flujo de objetos objetoEntrada

    objetoEntrada.close();

    System.out.println("DNI\t Nombre");
    System.out.println(p1.getAtributos());
    System.out.println(p2.getAtributos());

} catch (FileNotFoundException e) {
    System.out.println("¡El fichero no existe!");
} catch (IOException e) {
    System.out.println(e.getMessage());
} catch (Exception e) {
    System.out.println(e.getMessage());
};
```

La salida por la consola:

```
DNI          Nombre
55287188B   Ruiz Ramos, María
40302010A   González López, Juan
```

Las sentencias `try` y `catch` se utilizan para atrapar las excepciones que se producen durante la ejecución del programa: `FileNotFoundException`, `IOException` o `Exception`. De esta forma se atrapan los errores que se producen cuando el fichero de datos no existe o cuando hay un problema de lectura o escritura en el fichero.

Las excepciones son el mecanismo que proporciona Java para gestionar los errores de ejecución de una aplicación.

La sentencia `try-catch-finally` tiene la siguiente sintaxis:

```
try {
    sentencias-que-pueden-producir-una-excepción;
} catch (Excepción-tipo-1 e) {
    sentencias-para-excepción-tipo-1;
} catch (Excepción-tipo-2 e) {
    sentencias-para-excepción-tipo-2;
} catch (Excepción-tipo-3 e){
    sentencias-para-excepción-tipo-3;
} finally {
    sentencias-que-se-ejecutan-si-hay-excepción-o-no;
};
```

En una sentencia `try-catch-finally`, los bloques `catch` se pueden repetir tantas veces como excepciones de distinto tipo se desee atrapar. El bloque `finally` es opcional y solo puede aparecer una vez. Este bloque se ejecuta siempre.

El programa Java que escribe y lee objetos.

```
import java.io.*;

public class LecturaEscrituraObjetos {
    public static void main(String[] args) {
        String nombreFichero = "c:\\Objetos.dat";

        try {
            FileOutputStream ficheroSalida = new
                FileOutputStream(nombreFichero);
            ObjectOutputStream objetoSalida = new
                ObjectOutputStream(ficheroSalida);

            objetoSalida.writeObject(new Persona("55287188B",
                "María", "Ruiz Ramos"));
            objetoSalida.writeObject(new Persona("40302010A",
                "Juan", "González López"));
            objetoSalida.close();

            FileInputStream ficheroEntrada = new
                FileInputStream(nombreFichero);
            ObjectInputStream objetoEntrada = new
                ObjectInputStream(ficheroEntrada);

            Persona p1 = (Persona)objetoEntrada.readObject();
            Persona p2 = (Persona)objetoEntrada.readObject();

            objetoEntrada.close();

            System.out.println("DNI\t Nombre");
            System.out.println(p1.getAtributos());
            System.out.println(p2.getAtributos());
        } catch (FileNotFoundException e) {
            System.out.println("¡El fichero no existe!");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```



## A. Operadores del lenguaje Java

---

### Operadores aritméticos

Los operadores aritméticos son símbolos que representan operaciones aritméticas con dos operandos.

Operador	Operación aritmética
+	Suma de números y concatenación de cadenas
-	Resta
*	Producto
/	División
%	Módulo o resto

El operador + se utiliza para sumar dos números y también permite concatenar o unir dos cadenas de caracteres. Este operador está "sobrecargado" porque opera con números y con cadenas de caracteres.

### Operadores unarios y compuestos

Estos operadores aritméticos se utilizan para hacer más claro el código Java. Combinan la operación de asignación con una operación aritmética.

Los operadores ++ y -- son operadores unarios, es decir, utilizan un solo operando. Los operadores unarios se pueden aplicar antes y después del operando. Por ejemplo, x++ y ++x representan dos operaciones distintas aplicando el mismo operador. Ambas operaciones suman 1 a la variable x pero tienen un significado diferente.

## Operadores del lenguaje Java

---

Operador	Operación	Equivale a
++	a++	a = a + 1
--	a--	a = a - 1
+=	a+=b	a = a + b
-=	a-=b	a = a - b
*=	a*=b	a = a * b
/=	a/=b	a = a / b
%=	a%=b	a = a % b

La siguiente tabla muestra ejemplos de uso de los operadores unarios y de los operadores compuestos.

Expresión	Equivale a	Significado
x++	x = x + 1	Suma 1 al valor de x
x+=5	x = x + 5	Suma 5 al valor de x
y-=2	y = y - 2	Resta 2 al valor de y
z*=10	z = z * 10	Multiplica por 10 el valor de z
a/=b	a = a / b	Divide a entre b
c%=3	c = c % 3	Calcula el módulo de c dividido entre 3

## Operadores de relación

Los operadores de relación permiten comparar dos o más valores.

Operador	Significado	Ejemplo
=	Igual	nota = 10
<	Menor que	nota < 5
>	Mayor que	nota > 9
<=	Menor o igual	nota <= 7
>=	Mayor o igual	nota >= 5
<>	Distinto de	nota <> 0

## Operadores lógicos

Java utiliza tres operadores lógicos: el O lógico (disyunción), el Y lógico (conjunción) y la negación.

Los operadores O lógico (||) y el Y lógico (&&) se utilizan para evaluar expresiones lógicas que solo pueden tomar el valor falso o verdadero. El operador Y lógico (&&) devuelve false si uno de los operandos es false. El O lógico (||) devuelve true si uno de los operandos es true.

El operador de negación (!) es unario y devuelve el valor negado de una expresión lógica.

## Orden de precedencia de los operadores

La siguiente tabla muestra el orden de prioridad de los operadores. Indica qué operador se aplica primero en una expresión.

Operador	Descripción
++ -- !	Operadores unarios, negación
* / %	Producto, división, módulo
+ -	Suma, resta
< > <= >=	Menor, mayor, menor o igual, mayor igual
== !=	Igual, diferente
&&	Operador lógico AND
	Operador lógico OR
?	Operador condicional ternario ?
= += -= *= /= %=	Asignación y operadores combinados de asignación

## B. Referencias

---

Enlaces útiles sobre el lenguaje de programación Java y el desarrollo de aplicaciones.

### El lenguaje de programación Java

#### Wikibooks Fundamentos de programación Java

[http://es.wikibooks.org/wiki/Programaci%C3%B3n\\_en\\_Java](http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Java)

#### Java

[http://es.wikipedia.org/wiki/Java\\_\(lenguaje\\_de\\_programaci%C3%B3n\)](http://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n))

#### Estructuras de control y estructuras de selección de Java

<http://www.programacionfacil.com/java/if>

<http://www.programacionfacil.com/java:switch>

[http://www.programacionfacil.com/java/ciclo\\_for](http://www.programacionfacil.com/java/ciclo_for)

<http://www.programacionfacil.com/java/while>

[http://www.programacionfacil.com/java/do\\_while](http://www.programacionfacil.com/java/do_while)

[http://www.programacionfacil.com/java/ciclos\\_conclusiones](http://www.programacionfacil.com/java/ciclos_conclusiones)

<http://www.codexion.com/tutorialesjava/java/nutsandbolts/if.html>

<http://www.codexion.com/tutorialesjava/java/nutsandbolts/switch.html>

<http://www.codexion.com/tutorialesjava/java/nutsandbolts/for.html>

<http://www.codexion.com/tutorialesjava/java/nutsandbolts/while.html>

#### Blog sobre programación en Java

<http://aprender-java.blogspot.com>

## El API de Java

<http://download.oracle.com/javase/1,5,0/docs/api/allclasses-noframe.html>

### La clase Arrays

<http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Arrays.html>

### La clase ArrayList

<http://docs.oracle.com/javase/1.4.2/docs/api/java/util/ArrayList.html>

### La clase File

<http://download.oracle.com/javase/1,5,0/docs/api/java/io/File.html>

### La clase PrintWriter

<http://download.oracle.com/javase/1,5,0/docs/api/java/io/PrintWriter.html>

### Las clase Scanner

<http://download.oracle.com/javase/1,5,0/docs/api/java/util/Scanner.html>

### La clase StringTokenizer

<http://download.oracle.com/javase/1,5,0/docs/api/java/util/StringTokenizer.html>

## C. Glosario

---

### A

**Abstract Window Toolkit (AWT).** Componente Java para diseñar un interfaz gráfico de usuario.

**Active Server Page (ASP).** Una página ASP es un tipo especial de página HTML que contiene pequeños programas, denominados "scripts" (normalmente escritos en lenguaje VBScript), que son ejecutados en servidores Microsoft Internet Information Server antes de ser enviados al usuario para su visualización en forma de página HTML. Esos programas normalmente realizan consultas a bases de datos, de forma que estos resultados determinan la información que se envía al usuario. Los ficheros de este tipo llevan el sufijo asp.

**Active X.** Lenguaje desarrollado por Microsoft para la elaboración de aplicaciones que funcionan en cualquier plataforma, normalmente utilizando navegadores web. Los objetos ActiveX permiten hacer páginas web dinámicas.

**Ámbito.** Indica el entorno desde donde se puede utilizar un identificador. De forma general, un identificador solo puede ser utilizado en el bloque donde se ha declarado.

**Applet.** Programa desarrollado en Java que se ejecuta en un navegador o browser. Un applet forma parte de una página web y es utilizado para introducir acciones dinámicas tales como funciones de cálculo, animaciones o tareas simples sin necesidad de enviar una petición del usuario al servidor web.

**Application Program Interface (API).** Interfaz que se publica para facilitar el uso de los servicios que soporta un sistema operativo o una aplicación. Un API describe detalladamente los métodos y los argumentos de las funciones que constituyen la interfaz del sistema operativo o aplicación. Permite que un programador haga uso de las funciones almacenadas.

**Application Server.** Un servidor de aplicaciones es un programa servidor que se ejecuta en un ordenador en una red distribuida y proporciona la lógica de negocio de una aplicación. Un servidor de aplicaciones forma parte de la arquitectura de tres capas (Three-tier architecture), compuesta por un servidor de interfaz gráfica de usuario (GUI), un servidor de aplicaciones (lógica de negocio) y un servidor de base de datos y transacciones.

Esta arquitectura se compone de tres capas, denominadas comúnmente:

Top-tier / front-end. Interfaz gráfica de usuario basada en un navegador web (browser). Este interfaz se ejecuta normalmente utilizando un ordenador personal o una estación de trabajo.

Middle-tier / business logic. Aplicaciones que se ejecutan normalmente en un servidor intranet.

Third-tier / back-end. Servidor de bases de datos y transacciones. Este servicio se ejecuta normalmente en un servidor grande o mainframe.

Los sistemas heredados (legacy systems) con frecuencia forman parte del back-end, en tanto que el servidor de aplicaciones está integrado en la capa intermedia que existe entre la interfaz de usuario (front-end) y los sistemas heredados.

Un servidor de aplicaciones trabaja normalmente con un servidor web (Hypertext Transfer Protocol – HTTP) y es por ello denominado Web Application Server. El interfaz de usuario (front-end) se basa en HTML y XML, mientras que el servidor web soporta diferentes formas de enviar una petición al servidor de aplicaciones y devolver la respuesta al usuario,

usualmente enviando una nueva página web.

**Array.** Conjunto de datos de un mismo tipo de dato, por ejemplo entero, lógico o de clase. Un array está formado por un conjunto de casillas que almacenan variables. Cada casilla o elemento del array se identifica por un número de posición.

**Asignación.** Proceso por el cual se almacena un valor en una variable.

**Asymmetric Digital Subscriber Line (ADSL).** Tecnología de transmisión de tipo xDSL, que permite a los hilos telefónicos de cobre convencionales una alta velocidad de transmisión. Se denomina asimétrica porque permite mayor velocidad en la recepción que en la emisión de datos.

**Asymmetrical Keys.** Uso de una pareja de claves, una pública y la otra privada. Una clave se utiliza para cifrar y la otra para descifrar la información que viaja por una red pública de comunicaciones.

**Autenticación.** Autenticación o verificación de la identidad de un usuario o sistema mediante el uso de algoritmos de cifrado, normalmente basados en claves asimétricas.



## B

**Backbone.** Línea de telecomunicaciones de gran capacidad a la que se conectan otras líneas de menor capacidad.

En una red local un backbone es una línea o conjunto de líneas de la red que conectan con una conexión de ámbito extenso o a una red local para unir dos edificios. En Internet o en cualquier otra red de ámbito extenso (WAN), un backbone es un conjunto de rutas (paths) que conectan redes locales a una conexión de larga distancia. Los puntos de conexión se conocen como nodos.

**Back-end.** Front-end y back-end son términos que se utilizan para distinguir las interfaces y servicios que tienen relación directa con el usuario final. Una aplicación de tipo back-end sólo tiene una relación indirecta con el usuario a través de la interfaz.

**Base de datos.** Colección de datos organizados de forma que los contenidos pueden ser recuperados o actualizados fácilmente. Los modelos empleados por los sistemas de bases de datos son: el relacional y el orientado a objetos.

El modelo relacional se basa en el concepto matemático denominado "relación", que gráficamente se

puede representar como una tabla. En el modelo relacional, los datos y las relaciones existentes entre los datos se representan mediante estas relaciones matemáticas, cada una con un nombre que es único y con un conjunto de columnas. En este modelo la base de datos es percibida por el usuario como un conjunto de tablas.

El modelo orientado a objetos define una base de datos en términos de objetos, sus propiedades y sus operaciones. Los objetos con la misma estructura y comportamiento pertenecen a una clase, y las clases se organizan en jerarquías. Las operaciones de cada clase se definen en términos de procedimientos predefinidos denominados métodos.

El lenguaje estándar para realizar consultas y actualizaciones a una base de datos es SQL (Structured Query Language).

**Bit.** Dígito binario que almacena un valor 0 o 1.

**Bloque.** Código fuente Java que se escribe entre los caracteres { y }.

**Booleano.** Tipo de dato primitivo que almacena un valor lógico falso o verdadero.

**Bytecode.** Código intermedio independiente de la plataforma. El

código fuente Java se compila a Bytecode y este código es interpretado por la máquina virtual de Java.

**Browser.** Navegador web que permite visualizar documentos HTML o XML y navegar por el espacio Internet. Un navegador web es un programa cliente que utiliza el protocolo HTTP para realizar peticiones a servidores web a través de Internet y desplegar de forma gráfica al usuario la información recibida del servidor.

## C

**Clase.** Una clase describe a un conjunto de objetos que comparte los mismos atributos, comportamiento y semántica.

**Comentario.** Parte de un programa Java delimitado por los símbolos `/*` y `*/` o por `//` si se trata de un comentario de una línea. Los comentarios son útiles para explicar el diseño o el comportamiento de un programa.

**Compilador.** Programa que traduce el código fuente de un lenguaje en un código ejecutable o en un código intermedio como el Bytecode. Al proceso de análisis y traducción del lenguaje de programación se le denomina compilación.

**Constante.** Se refiere a las variables de un programa que mantienen el mismo valor durante la ejecución de un programa. Las constantes en Java se declaran con el delimitador final.

**Constructor.** Método que se utiliza para crear un objeto en una clase.

## D

**Declaración.** Sentencia en la que se define un nombre de atributo y el tipo o clase a la que pertenece.

**Dominio.** Un dominio identifica de forma unívoca a una organización o cualquier otra entidad en Internet. Un identificador de dominio se compone de dos niveles:

**Top-level domain (TLD).** Identifica la parte más general del nombre de dominio en una dirección de Internet. Un TLD puede ser genérico (gTLD) o código de país (ccTLD). "com" o "edu" son ejemplos de TLD's genéricos, en tanto que "es" o "fr" son ejemplos de códigos de país.

**Second-level domain (SLD).** Identifica al propietario del dominio con una dirección IP. "nebrija" es un ejemplo de un dominio de segundo nivel.

## E

**Encapsulación.** Consiste en definir todos los datos y métodos dentro de una clase. La encapsulación consiste en formar un paquete con los atributos (datos) y el comportamiento (métodos) de un objeto.

**Enterprise Java Bean (EJB).** Arquitectura de componentes desarrollada por Sun Microsystems para diseño de objetos distribuidos en Java. Un EJB se construye a partir de la tecnología JavaBeans. Un componente EJB tiene la ventaja de ser un elemento reutilizable en diferentes aplicaciones.

**Excepción.** Evento inesperado que se produce durante la ejecución de un programa. Una excepción rompe interrumpe el flujo de ejecución normal de un programa.

**Expresión.** Código que se forma uniendo expresiones simples formadas por literales o variables con operadores. El valor de una expresión se calcula considerando la precedencia de los operadores aritméticos y lógicos.

**eXtensible Markup Language (XML).** XML es un lenguaje de marcado para la descripción de datos estructurados. Permite declarar los contenidos de forma precisa y separar el contenido del

formato. XML ofrece una representación estructural de los datos, es un subconjunto de SGML optimizado para el Web que ha sido definido por el World Wide Web Consortium (W3C). Garantiza que los datos estructurados son uniformes e independientes de aplicaciones o fabricantes, lo que incrementa la interoperabilidad y ha dado origen a una nueva generación de aplicaciones de comercio electrónico en la Web.

Los objetivos de XML son:

1. XML debe ser directamente utilizable sobre Internet
2. XML debe soportar una amplia variedad de aplicaciones
3. XML debe ser compatible con SGML
4. Un programa que procese documentos XML debe ser fácil de escribir
5. El número de características opcionales en XML debe ser mínima, idealmente cero
6. Los documentos XML deben ser legibles y claros
7. El diseño de XML debe ser conciso
8. Los documentos XML deben crearse fácilmente
9. No importa si las marcas XML no son concisas

### F

**File Transfer Protocol (FTP).** El protocolo FTP se incluye como parte del TCP/IP. Es el protocolo de nivel de aplicación destinado a proporcionar el servicio de transferencia de ficheros en Internet. El FTP depende del protocolo TCP para las funciones de transporte, y guarda alguna relación con TELNET (protocolo para la conexión remota).

El protocolo FTP permite acceder a algún servidor que disponga de este servicio y realizar tareas como moverse a través de su estructura de directorios, ver y descargar ficheros al ordenador local, enviar ficheros al servidor o copiar archivos directamente de un servidor a otro de la red. Lógicamente y por motivos de seguridad se hace necesario contar con el permiso previo para poder realizar todas estas operaciones. El servidor FTP pedirá el nombre de usuario y clave de acceso al iniciar la sesión (login), que debe ser suministrado correctamente para utilizar el servicio.

**Firewall.** Dispositivo o componente software que utiliza reglas para especificar que protocolos o comunicaciones no pueden acceder a la red.

**Front-end.** Front-end y back-end son términos que se utilizan para

distinguir las interfaces y servicios que tienen relación directa con el usuario final. Una aplicación de tipo front-end interactúa directamente con el usuario.

### G

**Gateway.** Punto de una red que actúa como punto de entrada a otra red. En internet, un nodo de la red puede ser de dos tipos: gateway o host. Tanto los ordenadores de los usuarios de Internet como los ordenadores que sirven páginas son nodos de tipo host. Los ordenadores que controlan el tráfico en una red local o en un ISP - Internet Service Provider son nodos de tipo gateway. En una red local, un ordenador que actúa como nodo gateway comúnmente hace las funciones de servidor proxy y firewall.

**Graphical User Interface (GUI).** Interfaz gráfica de usuario. Es el medio a través del que un usuario interactúa con una aplicación informática. Un interfaz de usuario se compone normalmente de ventanas, botones, menús desplegables, menús contextuales, campos de texto, listas y otros objetos.

### H

**Herencia.** Concepto por el que una clase queda formada por

todos los atributos y métodos de una clase de orden superior de la que hereda.

**Hypertext Markup Language (HTML).** Lenguaje de texto con marcadores, denominados tags, que se utiliza para especificar el formato y comportamiento de las páginas web. HTML permite especificar todas las características del texto a presentar (tipo de letra, tamaño, color, posición, etc.), así como el inicio y fin de las zonas activas del texto, con la referencia del documento a presentar.

**Hypertext Transfer Protocol (HTTP).** Protocolo de transporte de hipertexto. Consta de un conjunto de reglas para intercambio de ficheros de texto, imágenes, sonido, video y otros formatos multimedia gráficos a través del World Wide Web.

### I

**Identificador.** Nombre que se da a un elemento de un programa. Mediante este nombre se hace referencia a cualquier elemento de un programa Java. Se aplica a clases, atributos, métodos y argumentos.

**Inicializar.** Asignar un valor a una variable antes de que sea utilizada en una expresión.

**Internet.** Red de telecomunicaciones nacida en 1969 en los EE.UU. a la cual están conectadas millones de personas, organismos y empresas en todo el mundo, mayoritariamente en los países más desarrollados. Internet es una red multiprotocolo cuyo rápido desarrollo está teniendo importantes efectos sociales, económicos y culturales, convirtiéndose en uno de los medios más influyentes de la llamada "Sociedad de la Información". La red Internet tiene una jerarquía de tres niveles formados por redes troncales, redes de nivel intermedio y redes aisladas.

**Internet Protocol (IP).** El IP es un protocolo que pertenece al nivel de red. Es utilizado por los protocolos del nivel de transporte como TCP para encaminar los datos hacia su destino. IP tiene la misión de encaminar el datagrama, sin comprobar la integridad de la información que contiene.

**Internet Protocol Address (IP Address).** La dirección de Internet se utiliza para identificar tanto a cada ordenador y a la red a la que pertenece, de manera que sea posible distinguir a todos los ordenadores conectados a una misma red. Con este propósito, y teniendo en cuenta que en Internet se conectan redes de diverso tamaño, existen tres

clases diferentes de direcciones, las cuales se representan mediante tres rangos de valores:

**Clase A.** El primer byte tiene un valor comprendido entre 1 y 126. Estas direcciones utilizan únicamente este primer byte para identificar la red, quedando los otros tres bytes disponibles para cada uno de los hosts que pertenezcan a esta misma red. Esto significa que podrán existir más de dieciséis millones de ordenadores en cada una de las redes de esta clase. Este tipo de direcciones es usado por redes muy extensas, pero hay que tener en cuenta que sólo puede haber 126 redes de este tamaño. ARPAnet es una de ellas, aunque son pocas las organizaciones que obtienen una dirección de "clase A". Lo normal para las grandes organizaciones es que utilicen una o varias redes de "clase B".

**Clase B.** Estas direcciones utilizan en su primer byte un valor comprendido entre 128 y 191, incluyendo ambos. En este caso el identificador de la red se obtiene de los dos primeros bytes de la dirección, teniendo que ser un valor entre 128.1 y 191.254 (no es posible utilizar los valores 0 y 255 por tener un significado especial). Los dos últimos bytes de la dirección constituyen el identificador del host, permitiendo un número máximo de 64516 ordenadores en la misma red. Este

tipo de direcciones tendría que ser suficiente para la gran mayoría de las organizaciones grandes. En caso de que el número de ordenadores que se necesita conectar fuese mayor, sería posible obtener más de una dirección de "clase B", evitando de esta forma el uso de una de "clase A".

**Clase C.** En este caso el valor del primer byte tendrá que estar comprendido entre 192 y 223, incluyendo ambos valores. Este tercer tipo de direcciones utiliza los tres primeros bytes para el número de la red, con un rango desde 192.1.1 hasta 223.254.254. De esta manera queda libre un byte para el host, lo que permite que se conecten un máximo de 254 ordenadores en cada red. Estas direcciones permiten un menor número de host que las anteriores, aunque son las más numerosas pudiendo existir un gran número de redes de este tipo (más de dos millones).

## J

**Java.** Entorno de desarrollo de aplicaciones web diseñado por Sun Microsystems.

**Java Database Connectivity (JDBC).** Estándar para acceso a bases de datos desde programas desarrollados en Java. JDBC utiliza una API basada en el lenguaje de consulta de bases de datos SQL.

### **Java Development Kit (JDK).**

Entorno de desarrollo y librerías para diseño de programas Java.

### **Java Runtime Environment (JRE).**

Subconjunto del JDK que permite ejecutar programas compilados en Bytecode. Está formado por una máquina virtual de Java y por librerías estándar.

### **Java Server Page (JSP).**

Java Server Page es una tecnología que se utiliza para controlar el contenido y apariencia de las páginas web mediante el uso de servlets. Un servlet es un pequeño programa Java que se direcciona desde la página web y es ejecutado en el servidor web para modificar el contenido de la página antes de que ésta sea enviada al usuario que la ha solicitado. Una página JSP contiene servlets para consultar bases de datos y generar de forma dinámica el contenido de una página HTML.

### **Java Virtual Machine (JVM).**

Programa que ejecuta programas java compilados en Bytecode. La máquina virtual de Java es un entorno seguro de ejecución de aplicaciones.

**Javascript.** Lenguaje de programación que permite dinamizar el contenido de una página HTML. Javascript es un lenguaje interpretado. Se utilizan normalmente para desarrollar funciones tales como:

1. Cambiar el formato de una fecha de forma automática en una página web
2. Desplegar un enlace a una página web en una ventana pop-up.
3. Modificar textos o gráficos mientras se realiza una acción de ratón de tipo "mouse rollover".

## L

### **Local Area Network (LAN).**

Red de área local que une servidores y puestos cliente. La extensión de este tipo de redes suele estar restringida a una sala o edificio, aunque también podría utilizarse para conectar dos más edificios próximos.

## M

**Mainframe.** Término que se utiliza para denominar a grandes ordenadores diseñados para satisfacer las necesidades de procesamiento de información de las organizaciones de mayor tamaño a nivel mundial.

**Método.** Función definida dentro de una clase. Un método puede devolver un valor o no, en tal caso se indica que devuelve `void`.

### N

#### **Network User Interface (NUI).**

Interfaz de usuario de red en un entorno Internet.

### O

#### **Open System Interconnection (OSI).**

El modelo OSI es utilizado por prácticamente la totalidad de las redes de ordenadores del mundo. Este modelo fue creado por el International Standard Organization ISO, consiste en siete niveles o capas donde cada una de ellas define las funciones que deben proporcionar los protocolos con el propósito de intercambiar información entre varios sistemas. Esta clasificación permite que cada protocolo se desarrolle con una finalidad determinada, lo cual simplifica el proceso de desarrollo e implementación. Cada nivel depende de los que están por debajo de él, y a su vez proporciona alguna funcionalidad a los niveles superiores.

A continuación se describen las funciones básicas de cada nivel.

**Aplicación.** El nivel de aplicación es el destino final de los datos donde se proporcionan los servicios al usuario.

**Presentación.** Convierte los datos que serán utilizados en el nivel de aplicación.

**Sesión.** Encargado de ciertos aspectos de la comunicación como el control de los tiempos de transmisión.

**Transporte.** Transporta la información de una manera fiable para que llegue correctamente a su destino.

**Red.** Nivel encargado de encaminar los datos hacia su destino eligiendo la ruta más efectiva.

**Enlace de datos.** Controla el flujo de datos, la sincronización y los errores que puedan producirse.

**Físico.** Se encarga de los aspectos físicos de la conexión, tales como el medio de transmisión o el hardware.

### P

**Plataforma.** En informática, una plataforma es un sistema que sirve como base para hacer funcionar determinados módulos de hardware o de software con los que es compatible. Una plataforma se define por una arquitectura hardware y una plataforma software que incluye sistemas operativos y entornos de desarrollo de aplicaciones.

**Point to Point Protocol (PTP).** Protocolo de comunicación entre dos ordenadores basado en una interfaz serie, típicamente un ordenador personal conectado vía



telefónica con un servidor. En este caso, el ISP proporcionaría una conexión punto a punto para atender a las peticiones que el ordenador personal realizaría a través de Internet.

**Portabilidad.** Característica que posee un programa para ejecutarse en diferentes plataformas informáticas.

**Programación orientada a objetos.** La programación orientada a objetos es una técnica de análisis y diseño de software que orienta a los elementos de un sistema, sus atributos y responsabilidades en vez de centrarse en el flujo de los procesos. El modelo abstracto está formado de clases. Una clase describe a un conjunto de objetos que comparte los mismos atributos, comportamiento y semántica.

La programación orientada a objetos ha cambiado las reglas de desarrollo de software. Este paradigma se basa en los objetos y en los datos, en vez de en las acciones. Tradicionalmente, todo programa era concebido como un procedimiento lógico que recibía datos de entrada y, tras procesarlos, generaba datos de salida. Bajo este modelo, el reto consistía en cómo codificar el proceso lógico y no en cómo definir los datos. En la programación orientada a objetos

el esfuerzo se centra en modelar los objetos que componen un sistema, sus responsabilidades y en las relaciones que existen entre diferentes objetos.

**Protocolo.** Protocolo, descripción formal de formatos de mensaje y de reglas que dos ordenadores deben seguir para intercambiar información. Un protocolo puede describir detalles de bajo nivel de las interfaces máquina-a-máquina o intercambios de alto nivel entre programas.

**Proxy Server.** Servidor especial encargado, entre otras cosas, de centralizar el tráfico entre Internet y una red privada, de forma que evita que cada una de las máquinas de la red interior tenga que disponer necesariamente de una conexión directa a la red. Al mismo tiempo contiene mecanismos de seguridad (firewall) que impiden accesos no autorizados desde el exterior hacia la red privada.

## R

**Runtime.** Programa que permite ejecutar programas Java compilados en Bytecode. El sistema runtime dispone de todo lo necesario para la carga dinámica de las clases de un programa, las librerías estándar del lenguaje y una máquina virtual de Java.

### S

**Secure Socket Layer (SSL).** Protocolo que soporta cifrado para garantizar la privacidad de la comunicación entre un browser y un servidor web. Es el protocolo de seguridad más utilizado en Internet, es una tecnología diseñada por Netscape Communications Inc. que dispone un nivel seguro entre el servicio clásico de transporte en Internet (TCP) y las aplicaciones que se comunican a través de él.

Las comunicaciones tienen lugar en dos fases, en una primera fase se negocia entre el cliente y el servidor una clave simétrica sólo válida para esa sesión. En la segunda fase, se transfieren datos cifrados con dicha clave. Este sistema es transparente para las aplicaciones finales, que simplemente saben que el canal (mantenido por el navegador y el servidor de comercio o servidor seguro) se encarga de proporcionarles confidencialidad punto a punto.

La fase inicial que utiliza tecnología de cifrado de clave pública se realiza muy cuidadosamente para evitar tanto la intromisión de terceras partes como para evitar suplantaciones de identidad del centro servidor. El navegador incluye las claves públicas de ciertos notarios electrónicos o entidades

certificadoras autorizadas y se pone en comunicación con el servidor seguro que le envía su clave pública, rubricada por el notario. La identificación se completa enviando al servidor un mensaje aleatorio que éste debe firmar. De esta forma sabe el cliente que al otro lado está quien dice ser.

Verificada la identidad del servidor, el cliente genera una clave de sesión y la envía cifrada con la clave pública del servidor. Conociendo ambos la clave de sesión (y el servidor es el único en poderla descifrar al requerir su clave privada), se intercambian datos con seguridad cifrados por el algoritmo de clave secreta.

**Servlet.** Programa Java que aporta más funcionalidad a un servidor web generando contenidos dinámicos e interactuando con clientes web utilizando el modelo Request-Response.

**Simple Mail Transfer Protocol (SMTP).** El protocolo SMTP proporciona el servicio de correo electrónico. Permite enviar mensajes de texto y archivos binarios de cualquier tipo a otros usuarios de la red. Los mensajes de correo electrónico no se envían directamente a los ordenadores personales de cada usuario, sino que se utiliza un ordenador que actúa como servidor de correo

electrónico permanentemente. Los mensajes permanecen en este sistema hasta que el usuario los transfiere a su propio ordenador.

Sistema operativo. Programa o conjunto de programas que efectúan la gestión de los procesos básicos de un sistema informático y permite la normal ejecución del resto de las operaciones. Un sistema operativo gestiona los recursos de un sistema informático.

**Standard Generalized Markup Language (SGML).** SGML es un lenguaje de marcado de texto que se utiliza para especificar el formato de documentos. SGML permite que la estructura de un documento pueda ser definida en base a la relación lógica de sus partes, se basa en la idea de que los documentos se componen de una estructura y elementos semánticos que pueden describirse sin necesidad de indicar su apariencia.

Un documento SGML se marca de modo que no dice nada respecto a su representación en la pantalla o en papel. Un programa de presentación debe unir el documento con la información de estilo a fin dar al documento su apariencia final.

**Swing.** Conjunto de componentes desarrollados para diseñar una interfaz gráfico de usuario.

## T

**Transmission Control Protocol (TCP).** Protocolo de comunicación que permite el enlace entre aplicaciones a través de Internet. Este protocolo pertenece al nivel de transporte y es el encargado de dividir el mensaje original en datagramas de menor tamaño. Los datagramas serán dirigidos a través del protocolo IP de forma individual. El protocolo TCP se encarga además de añadir información necesaria en la cabecera de cada datagrama.

Cuando la información se divide en datagramas para ser enviados, el orden en que éstos lleguen a su destino no tiene que ser el correcto. Cada uno de ellos puede llegar en cualquier momento y con cualquier orden, e incluso puede que algunos no lleguen a su destino o lleguen con información errónea. Para evitar todos estos problemas el TCP numera los datagramas antes de enviarlos, de manera que sea posible volver a unirlos en el orden adecuado. Esto permite también solicitar el envío de los datagramas individuales que no se hayan recibido o que contengan errores, sin que sea necesario volver a enviar el mensaje completo.

**Transmission Control Protocol / Internet Protocol (TCP/IP).** TCP/IP es el protocolo que utilizan los ordenadores conectados a

Internet para comunicarse entre sí. TCP/IP se encarga de que la comunicación entre diferentes ordenadores sea posible ya que es compatible con cualquier hardware y sistema operativo. TCP/IP no es un único protocolo, sino un conjunto de protocolos que cubren los distintos niveles del modelo OSI. Los dos protocolos más importantes son el TCP (Transmission Control Protocol) y el IP (Internet Protocol), que son los que dan nombre al conjunto. En Internet se diferencian cuatro niveles o capas en las que se agrupan los protocolos, relacionadas con los niveles OSI de la siguiente forma:

**Aplicación.** Corresponde con los niveles OSI de aplicación, presentación y sesión. Aquí se incluyen protocolos destinados a proporcionar servicios, tales como correo electrónico SMTP (Simple Mail Transfer Protocol), transferencia de ficheros FTP (File Transfer Protocol), conexión remota (TELNET) y otros más recientes como el protocolo HTTP (Hypertext Transfer Protocol).

**Transporte.** Coincide con el nivel de transporte del modelo OSI. Los protocolos de este nivel, tales como TCP y UDP, se encargan de manejar los datos y proporcionar la fiabilidad necesaria en el transporte de los mismos.

**Red.** Incluye al protocolo IP, que se encarga de enviar los paquetes de información a sus destinos correspondientes y es utilizado por los protocolos del nivel de transporte.

**Enlace de datos.** Los niveles OSI correspondientes son el de enlace y el nivel físico. Los protocolos que pertenecen a este nivel son los encargados de la transmisión a través del medio físico al que se encuentra conectado cada host, como puede ser una línea punto a punto o una red Ethernet.

El TCP/IP necesita funcionar sobre algún tipo de red o de medio físico que proporcione sus propios protocolos para el nivel de enlace de Internet. Por este motivo hay que tener en cuenta que los protocolos utilizados en este nivel pueden ser muy diversos y no forman parte del conjunto TCP/IP. Sin embargo, esto no debe ser un problema, puesto que una de las funciones y ventajas principales del TCP/IP es proporcionar una abstracción del medio de forma que sea posible el intercambio de información entre medios diferentes y tecnologías que inicialmente son incompatibles.

Para transmitir información a través de TCP/IP, ésta debe ser dividida en unidades de menor tamaño, lo que proporciona grandes ventajas en el manejo de los datos. En TCP/IP cada una de

estas unidades de información recibe el nombre de datagrama. Un datagrama es un conjunto de datos que se envía como un mensaje independiente.

**Tunneling.** Permite la transmisión segura de datos a través de Internet.

## U

**Unicode.** Sistema de codificación de caracteres que utiliza 16 bits para representar cada carácter. Esto permite representar prácticamente cualquier alfabeto del mundo. Los programas Java utilizan caracteres Unicode.

**Unified Modeling Language (UML).** El UML se ha convertido en el lenguaje de modelado de la industria del software. UML es el lenguaje estándar para análisis y diseño de aplicaciones orientadas a objetos. Es resultado de años de investigación en el ámbito de la ingeniería del software.

**Uniform Resource Locator (URL).** Dirección de un fichero o recurso que es accesible a través de Internet. El recurso puede ser una página HTML, una imagen, un programa o cualquier otro tipo de fichero soportado por el protocolo HTTP. La dirección URL contiene el protocolo requerido para acceder al recurso, el dominio que identifica a un ordenador en Internet y una descripción de la

ubicación física del fichero en el ordenador.

## V

**Variable.** Elemento de un programa Java identificado por un nombre. Una variable almacena un valor y tiene un tipo de dato y un ámbito.

**VBScript.** Lenguaje interpretado desarrollado por Microsoft que es un subconjunto de Visual Basic. Este lenguaje ha sido diseñado para ser interpretado por navegadores web. VBScript es comparable a otros lenguajes para diseño de páginas web como Javascript, o Perl.

## W

**Web Server.** Un servidor web es un equipo conectado a Internet con un conjunto de documentos almacenados, normalmente escritos en formato HTML, y un programa que atiende a las peticiones de documentos realizadas por los usuarios. Un servidor web se comunica con los navegadores o browsers mediante el protocolo HTTP. Un servidor web ejecuta programas que realizan consultas a bases de datos y generan páginas HTML dinámicas.

**Web Site.** Conjunto de páginas web almacenadas en un punto de red con una dirección única a las

que se accede a través de una página de inicio. Un sitio web es público y cualquier usuario puede acceder a él para obtener información.

**Wide Area Network (WAN).**

Red de ordenadores que cubre un espacio extenso, conectando a puestos de trabajo de una ciudad o un país completo. Este tipo de redes se basan en las líneas de teléfono y otros medios de transmisión más sofisticados, como pueden ser las microondas. La velocidad de transmisión suele ser inferior que en las redes locales.

**World Wide Web.** Sistema utilizado para explorar sitios web que residen en Internet. World Wide Web es el componente más visible y más conocido de Internet.