

FPRG

Carpeta de
Montero Espinosa
(sólo teoría)

FPRG Teoría

PROGRAMA DE LA ASIGNATURA FPRG

Tema 1. Presentación

Tema 2. Introducción a los objetos

Tema 3. Tipos, valores y variables

Tema 4. Métodos

Tema 5. Estructuras de control

Tema 6. Clases

Tema 7. Relaciones entre clases

Tema 8. Colecciones de datos

Tema 9. Estructuras dinámicas de datos

TEMA 1: PRESENTACIÓN

Este tema es solo una introducción, para ponernos en situación. **NO será objeto de evaluación** en el examen final ni en ejercicios de clase.

En el punto 1.2 veremos lo que es el lenguaje máquina.

Debemos diferenciar los conceptos de "lenguaje informático" y "lenguaje de programación". Los lenguajes informáticos engloban a los lenguajes de programación (usados para crear programas) y a otros más, como los lenguajes de marcas (usados para describir a un ordenador el formato de un documento).

Por ejemplo, **HTML** no es un lenguaje de programación, sino un **lenguaje de marcas**. Esto es, es otro tipo de lenguaje informático.

IMPORTANTE: la diferencia entre compilador e intérprete: un **compilador** sólo traduce el código en lenguaje de programación a lenguaje máquina, para luego ser ejecutado cuando queramos. Un **intérprete** realiza ambas operaciones seguidas, cada vez que lo utilizamos.

Obviando aspectos relativos a la organización de la asignatura, profesorado, evaluación... (conceptos incluidos en el temario oficial de este primer capítulo), vamos a introducirnos en estos primeros apuntes en conceptos básicos de programación, para presentar el lenguaje con el que trabajaremos durante el curso: Java.

1.1 Programación

Se llama **programación** a la creación de un programa de computadora, un conjunto concreto de instrucciones que una computadora puede ejecutar.

El programa se escribe en un **lenguaje de programación**, aunque también se pueda escribir directamente en lenguaje de máquina, con cierta dificultad. Un lenguaje de programación es una técnica estándar de comunicación que permite expresar las instrucciones que han de ser ejecutadas en una computadora. Consiste en un conjunto de reglas sintácticas y semánticas que definen un **lenguaje informático** para crear programas.

Programas y algoritmos

Un algoritmo es una secuencia no ambigua, finita y ordenada de instrucciones que han de seguirse para resolver un problema.

Un programa normalmente **implementa** (traduce a un lenguaje de programación concreto) un algoritmo. Puede haber programas que no se ajusten a un algoritmo (pueden no terminar nunca), en cuyo caso se denomina procedimiento a tal programa.

Los programas suelen subdividirse en partes menores (módulos), de modo que la complejidad algorítmica de cada una de las partes sea menor que la del programa completo.

Se suele decir que un programa está formado por algoritmos y estructuras de datos.

Compilación

El programa escrito en un lenguaje de programación no es inmediatamente ejecutado en una computadora. La opción más común es **compilar** el programa, aunque también puede ser ejecutado mediante un **intérprete** informático.

El código fuente del programa se debe someter a un proceso de transformación para convertirse en lenguaje máquina, interpretable por el procesador. A este proceso se le llama **compilación**.

Normalmente la creación de un programa ejecutable (un típico .exe para Microsoft Windows) conlleva dos pasos. El primer paso es la compilación. El segundo paso se llama **enlazado** (del inglés link o linker); se junta el código de bajo nivel generado de todos los ficheros que se han mandado compilar y se añade el código de las funciones que hay en las bibliotecas del compilador para que el ejecutable pueda comunicarse con el sistema operativo.

Un programa podría tener partes escritas en varios lenguajes (generalmente C, C++ y Asm), que se podrían compilar de forma independiente y enlazar juntas para formar un único ejecutable.

En FPRG y LPRG utilizaremos el lenguaje Java.

En clase comentaremos detalles acerca de la bibliografía recomendada, así como del entorno que usaremos para crear nuestros programas, Bluej.

Código máquina: "1s" y "0s", traducidos a voltajes, que aplicados sobre determinados canales o buses, provocan un comportamiento específico.

En FDOR (2º curso) y SEDG (3º) estudiaremos **ensamblador**.

Ya hemos visto la diferencia entre compilador e intérprete.

El nombre del lenguaje "Ada" es un homenaje a Ada Lovelace, hija de Lord Byron, considerada la primera programadora de la historia por su trabajo junto a Charles Babbage, diseñador de la famosa "máquina analítica", primer computador moderno, de 1837.

La ejecución de programas implementados con Java es algo peculiar: primero se compila el código a un lenguaje intermedio, "casi máquina", llamado bytecode (archivos .class). Posteriormente, en cada máquina, se interpreta este bytecode, gracias a la **Máquina Virtual de Java**.

1.2 El lenguaje Java

Java es un **lenguaje de alto nivel**, moderno y vivo, que recoge los elementos de programación que típicamente se encuentran en todos los lenguajes, permitiendo la realización de programas profesionales.

Además, Java es un **lenguaje orientado a objetos**.

Existe una amplia bibliografía que presenta el lenguaje y ayuda a su aprendizaje, así como entornos de ayuda al programador, que automatizan las tareas de producción, prueba, y puesta a punto del código.

A continuación, explicamos los dos conceptos más importantes de entre los que acabamos de nombrar:

Java: lenguaje de alto nivel

Los lenguajes de programación han evolucionado a lo largo de la historia de los ordenadores, en lo que se denomina generaciones de lenguajes. Existen cinco generaciones:

- Primera generación: Código Máquina.

Es el nivel más bajo de los lenguajes de programación y está formado por los códigos máquina aceptados por los ordenadores. Como cada ordenador tiene un código máquina distinto, por estar formado por elementos electrónicos distintos, los programas escritos en código máquina sólo funcionan en el ordenador para el cual están escritos.

- Segunda Generación: Lenguaje Ensamblador.

Son lenguajes de bajo nivel que permiten usar abreviaciones (**macroinstrucciones**) para las instrucciones del código máquina. Aquí ya se hace necesaria la compilación.

- Tercera Generación: Lenguajes de Alto Nivel.

Se caracterizan por el uso de **macroinstrucciones**, construcciones más o menos intuitivas (típicamente en inglés) que permiten crear programas complejos y relativamente sencillos de mantener y modificar. Por supuesto, la compilación también será aquí necesaria para que el ordenador entienda nuestras instrucciones.

Son lenguajes de alto nivel: *Ada, Basic, C, C++, C#, Fortran, Java, Pascal, Perl, PHP...* y muchos otros.

- Cuarta Generación: Lenguajes de Muy Alto Nivel.

También denominados 4GL (4th Generation Languages), son mucho más orientados al usuario que los anteriores, y los programas se desarrollan usando instrucciones muy próximas al lenguaje humano. Se les suele llamar "lenguajes no procedimentales", ya que los programadores especifican qué es lo que quieren que el ordenador realice, y no cómo. Estos lenguajes se usan en ámbitos muy especializados o de investigación.

- Quinta Generación: Lenguajes Naturales.

Estos utilizan un lenguaje humano para ofrecer una relación con el ordenador totalmente intuitiva. Los lenguajes de quinta generación están todavía en su infancia, y actualmente puede considerarse que están más cerca de los lenguajes 4GL que de los lenguajes humanos.

Pues bien, Java pertenece a la tercera generación, con lo que en FPRG usaremos un código más o menos comprensible e intuitivo que, además, será independiente de la máquina. Así mismo, tenemos que acostumbrarnos a la idea de que Java es un lenguaje vivo, esto es, su juego de macroinstrucciones va evolucionando cada día. Como programadores de Java, deberemos adaptarnos constantemente.

Java: lenguaje Orientado a objetos

Como ya sabemos, Java es un lenguaje orientado a objetos.

POO – “Programación Orientada a Objetos”.

OOP – “Object-Oriented Programming”.

Lo repetiremos varias veces: en programación, los “objetos” no son objetos materiales, sino una forma estructurada de encapsular datos.

La Programación Orientada a Objetos (POO) define los programas en términos de “clases de objetos”, objetos que son entidades que combinan estado (datos), comportamiento (procedimientos ó métodos) e identidad. Expresaremos un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

Esto difiere de los **lenguajes imperativos tradicionales**, en los que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca en éstos es el procesamiento de unos datos de entrada para obtener otros de salida.

Los programadores de lenguajes imperativos escriben funciones y después les pasan datos. Sin embargo, los programadores que emplean lenguajes orientados a objetos definen objetos con datos y métodos y después envían mensajes a los objetos diciéndoles que realicen esos métodos en sí mismos.

No entraremos más en este concepto, puesto que su total comprensión es uno de los objetivos principales de esta asignatura, y a ello nos dedicaremos a lo largo de los 8 temas que nos restan.

Además de Java, son lenguajes orientados a objetos: *Ada*, *C++*, *C#*, *VB.NET*, *PHP*... y muchos otros.

Java en la actualidad

Desde su comienzo, nacido como proyecto de Sun Microsystems para la programación de dispositivos inteligentes (como PDAs), en diciembre de 1990, bajo la denominación de “Oak”, hasta la actualidad, el ya renombrado lenguaje Java ha evolucionado hasta convertirse en uno de los lenguajes de programación con mayor uso en todo el mundo.

Tras algún titubeo aplicándolo a dispositivos interactivos, en 1994 el equipo de Sun reorientó la plataforma hacia la **Web**, y para 1995 Java estaba ya soportado por los navegadores de Netscape. La primera versión instalable de Java fue publicada sólo un año después.

En la actualidad, Java está muy presente:

- En la Web: tanto en **clientes** (todos los navegadores Web tienen capacidad para ejecutar applets de Java, con lo que tiene asegurado el uso del gran público. Se usa por ejemplo en Yahoo! Games, en reproductores de video...) como en **servidores** (donde es más popular que nunca, con infinidad de sitios empleando tecnologías Java).
- En el PC de escritorio: aunque las aplicaciones Java han sido relativamente “raras” para el uso doméstico, por diversas razones, cada vez se acercan más al PC de sobremesa. Así, hay aplicaciones Java cuyo uso está ampliamente extendido, como Azureus (programa de intercambio de archivos P2P), y en otras, es el motor de alguna de sus partes. Es el caso de MATLAB, que usa Java como motor para su interface gráfica, y para parte del motor de cálculo.

Como hemos dicho, el lenguaje Java ha experimentado (y lo sigue haciendo) numerosos cambios desde la versión primigenia, JDK 1.0, así como un enorme incremento en el número de clases y paquetes que componen la librería estándar.

En esta asignatura trabajaremos con la última versión: **J2SE 6.0**.

Como en la mayoría de los casos, los nombres que se han usado para denominar a este lenguaje tienen orígenes de lo más pintorescos:

“Oak” hacía referencia al roble que James Gosling, padre del lenguaje tenía junto a su oficina.

Con respecto a “Java”, hay muchas teorías; se dice que puede corresponder al acrónimo “James Gosling, Arthur Van Holf, y Andy Bechtolsheim”, creadores oficiales del lenguaje, o al de “Just Another Vague Acronym”; aunque la hipótesis que más fuerza tiene es la de que debe su nombre a un tipo de café disponible en la cafetería cercana. De ahí su anagrama. Un pequeño símbolo que da fuerza a esta teoría es que los 4 primeros bytes (el número mágico) de los archivos.class que genera el compilador, son en hexadecimal, 0xCAFEBAE.

1.3 ¿Qué necesitamos para programar con Java?

Puede que las notas siguientes sean obvias para alumnos con ciertos conocimientos de informática. En cualquier caso serán una buena referencia para saber por donde empezar.

JDK – "Java Development Kit".

OJO! No descargaremos las versiones de la primera lista de la página Web, tituladas "JDK 6 Update 14 with...", puesto que estas incluyen un entorno (NetBeans, por ejemplo) que no será el que usemos para nuestros ejercicios. ELIGE BIEN LA DESCARGA.

IDE – "Integrated Development Environment".

Es recomendable que todos usemos el mismo entorno de desarrollo, para entendernos en algún comentario de clase.

Instala BlueJ
DESPUÉS de instalar el kit de desarrollo JDK 6.0.

Aparte de los recursos disponibles en el laboratorio de la Escuela, todo aquel que disponga de un ordenador personal puede instalar (y es MUY recomendable) en el mismo un entorno de desarrollo Java que le permitirá comprobar los ejemplos y ejercicios que se proponen en la asignatura.

Sólo se necesita lo siguiente:

- Preparar un **entorno de trabajo**.

Se trata de crear, en nuestro ordenador, carpetas que nos facilitarán el orden en nuestro trabajo. Típicamente, la estructura de directorios será:

```
C:
C:\java
C:\java\fpgr
```

- **Kit de desarrollo Java.**

El kit de desarrollo Java (JDK) está disponible en Internet, en todas sus versiones, para diferentes sistemas operativos.

Un kit de desarrollo proporciona un compilador (*javac*), un intérprete (*java*), un generador de documentación (*javadoc*), y otras herramientas complementarias.

Actualmente (julio de 2009) la última edición existente en la Web (<http://java.sun.com/javase/downloads/index.jsp>) es la "**JDK 6 Update 14**".(ojo!! de la lista "Java SE Development Kit (JDK)"). Con acceder a la página indicada, y pulsar el botón "Download" junto a nuestra versión, accederemos a la instalación. A continuación, sólo faltará aceptar los acuerdos de licencia, elegir la opción adecuada para nuestro equipo (típicamente *Windows Offline Installation, Multi-language*), y dejar que se instale en el directorio que él mismo propone.

- **Entorno integrado de desarrollo.**

Aunque no es imprescindible (podríamos escribir nuestro código usando cualquier editor básico de texto, como el NotePad de Windows), es conveniente usar un entorno integrado de desarrollo (IDE), que nos facilite el trabajo.

Hay muchos entornos disponibles, académicos (*BlueJ, DrJava, jGrasp...*), todos ellos gratuitos, y profesionales (*NetBeans, Eclipse, Jbuilder, Jcreator, IntelliJ IDEA...*) algunos de ellos de pago.

Para FPRG y LPRG usaremos **BlueJ**.

BlueJ es un entorno de desarrollo muy sencillo de uso, pensado para aprender a programar en Java. Sus principales ventajas son que es gratuito, fácil de usar, ligero (no requiere un ordenador muy potente) y dispone de un libro de texto de acompañamiento (tenéis las señas del mismo en la Web que indicamos más abajo), además de multitud de tutoriales y documentos online al respecto.

Para instalarlo solo tenemos que descargarlo de la Web <http://www.bluej.org> para lo cual pulsamos sobre el botón "Download" del menú y elegimos la última versión oficial para nuestro sistema operativo (NUNCA versiones "Beta"). Ejecutando el fichero descargado, y siguiendo las instrucciones, el programa se instalará perfectamente.

TEMA 2: INTRODUCCIÓN A LOS OBJETOS

En este tema vamos a introducir brevemente el concepto de objeto y de clase. Esto no es más que una iniciación que desarrollaremos más en profundidad en el tema 6.

2.1 Clases

Este es el concepto principal de la POO (Programación Orientada a Objetos).

La programación orientada a objetos encapsula datos (atributos) y comportamientos (métodos).

Una **clase** es un programa java que agrupa una serie de datos (atributos) y de procedimientos (métodos) que tienen algo en común.

Estructura de una clase

Una clase siempre tiene una estructura similar:

```
DeclaraciondeClase {
    declaracionesdeAtributos
    declaracionesdeMetodos
}
```

Para elegir el nombre de una clase (también aplicable para toda la sintaxis de Java) debemos tener en cuenta que:

- se diferenciarán mayúsculas de minúsculas.
- no usaremos espacios, acentos, ni la letra "ñ".

Es importante diferenciar entre **atributo** (dato característico inherente a una clase) y **valor** (valor determinado que daremos a cada atributo, al crear y utilizar objetos de una clase).

El nombre del método y su tipo de retorno son partes indispensables de su declaración.

- La clase empieza por un nombre que comienza con letra mayúscula seguido por una llave.
- Luego se escriben los datos inherentes a la clase, que llamaremos **atributos**.
- A continuación se escriben los procedimientos para manipular esos datos, que llamaremos **métodos**.

A su vez, la programación de métodos seguirá la forma:

```
declaraciondeMetodo {
    cuerpodemetodo
}
```

Donde la declaración del método contiene información del nombre del método (que empezará por minúscula), el tipo de retorno del mismo, el número y tipo de argumentos necesarios, y qué otras clases y objetos pueden llamar al método.

El cuerpo del método contiene el código que expresa su funcionamiento.

Entre los métodos son de especial importancia los **constructores** que inicializan los atributos con unos valores determinados. Los constructores tienen que tener, obligatoriamente, el mismo nombre que la clase.

Clases ejecutables

Existen clases no ejecutables y ejecutables. Para que una clase se pueda ejecutar debe tener un método con la siguiente cabecera `public static void main (String[] args)`. En ese caso al ejecutar la clase lo que se ejecutará será lo que contenga el método `main`. En caso de que la clase no tenga método `main` podremos compilarla pero no ejecutarla.

Hay que tener clara la diferencia entre **clase** y **objeto**.

IMPORTANTE: en programación, los "objetos" no son objetos materiales, sino una forma estructurada de encapsular datos.

Este concepto se verá a fondo en el tema 6.

Fijate que en con la inicialización asignamos **valores a atributos**. En el ejemplo asignamos el valor 3.5 a un atributo del objeto Circulo que estamos creando (lo veremos en un ejemplo de clase).

Debemos entender bien el concepto de **referencia**, y distinguirla del objeto al que apunta.

En programación, esto es usar **alias**.

2.2 Objetos

Un objeto es un caso particular de una clase.

En Java, la unidad de programación es la clase de la cual es algún momento se instancian (esto es, se crean) objetos. Habitualmente un programa emplea varios objetos de la misma clase.

Para crear un objeto de una clase se utiliza el comando new. Frecuentemente, se verá la creación de un objeto java con una sentencia como esta:

```
→ Circulo c1 = new Circulo(3.5)
```

La anterior sentencia instancia (crea) un objeto de la clase Circulo, aunque realmente realiza tres acciones que podremos programar por separado: **declaración, ejemplarización e inicialización**.

- Circulo c1 es una declaración de variable que sólo le dice al compilador que el nombre c1 se va a utilizar para referirse a un objeto cuyo tipo es Circulo.
- El operador new ejemplariza la clase Circulo (crea un nuevo objeto Circulo).
- Circulo(3.5) inicializa el objeto.

Referencia de un objeto

Cuando se crea un objeto de una clase, el programador se queda con una **referencia** para poder llegar a ese objeto (en el ejemplo anterior la referencia es c1).

Es muy importante entender que una referencia es sólo un nombre que crearemos para apuntar a un objeto, y que este nombre lo podemos programar (declarar) antes de crear el propio objeto.

A los componentes (ya sean atributos o métodos) de un objeto se accede con la siguiente sintaxis:

```
referenciaDelObjeto.nombreDelComponente
```

Por ejemplo:

```
c1.escala(2.0);
c1.area();
```

Conviene que vayamos conociendo ya las siguientes cuestiones:

- Un objeto puede tener varias referencias. Esto es, dos o más referencias pueden apuntar al mismo objeto. En el siguiente ejemplo:

```
Circulo c1 = new Circulo(3.5);
Circulo c2 = c1;
```

Podemos acceder a los componentes del objeto usando cualquiera de las dos referencias, c1 y c2, puesto que ambas apuntan al mismo objeto y el resultado será el mismo.

- Cuando una referencia no se refiere a ningún objeto (por ejemplo, antes de crearlo, si aun sólo lo hemos declarado) se dice que apunta a null. En el siguiente ejemplo:

```
Circulo c1;
c1 = new Circulo(1.0)
```

En la primera sentencia sólo hemos declarado la referencia, con lo que c1 todavía no apunta a ningún objeto, esto es, apunta a null. Posteriormente inicializamos el objeto, con lo que la referencia c1 apunta ya al objeto creado.

Siempre que aun queden referencias apuntando a un objeto, podremos acceder a él.

- Cuando un objeto se queda sin referencias no es accesible por el programa y el sistema se reserva el derecho de **reciclarlo**. Esto ocurre en el siguiente ejemplo:

```
Circulo c1=new Circulo(1.0);
c1=null;
```

En el que el objeto creado pierde su única referencia, `c1`, al hacer que esta apunte a `null`, con la segunda sentencia. Con esto, ya no podremos usar el objeto, y el sistema lo reciclará.

Este concepto de **comparación** debe quedar muy claro.

- Cuando comparamos, en java, se comparan referencias. La comparación será cierta sólo si ambas referencias referencian al mismo objeto.

NO se comparan los objetos, si intentamos comparar dos objetos, el resultado será falso incluso si son dos objetos iguales en todos los atributos. Veamos el siguiente ejemplo:

```
Circulo c1=new Circulo(1.0);
Circulo c2=c1;

c1==c2           //true
c1.escala(2.0);
c1==c2           //true
```

Donde, aunque accedamos a un método de `Circulo` que cambia el valor de uno de los atributos del objeto `c1`, las referencias `c1` y `c2` siguen apuntando al mismo objeto.

En contraposición, en este otro:

```
Circulo c1=new Circulo(1.0);
Circulo c2=new Circulo(1.0);

c1==c2           //false
```

Aunque ambos objetos tengan idénticos valores de sus atributos, las referencias `c1` y `c2` no apuntan al mismo objeto, con lo que la comparación devolverá un `false`.

En programación este concepto recibe el nombre de **composición**.

- Un objeto puede ser parte integrante de otros. Lo veremos en los ejemplos de clase, donde un objeto de la clase `Rectángulo` puede tener como atributos cuatro objetos de la clase `Punto`.

2.3 Clases predefinidas

Con la práctica aprenderemos a manejarnos con paquetes, cuando haga falta.

La herramienta **javadoc** puede ser usada para documentar cualquier clase.

Usualmente, los programadores emplean el término "**API**" para referirse al documento generado, a partir de una clase, con **javadoc**.

Al final de estos apuntes se incluyen las API's de las clases `String` y `Math`.

En Java existen clases predefinidas que podremos usar a nuestro antojo. Esto es, habrá muchas operaciones que precisemos realizar, cuya implementación no tenga que correr por nuestra cuenta, puesto que ya están programadas de antemano. A la colección de clases predefinidas existentes en java, se le llama **API** (Application Programming Interface).

El lenguaje Java tiene muchos API's. Hay tantos que las clases han tenido que ser organizadas en grupos llamados **paquetes**. Por ejemplo, las clases que soportan I/O (input y output) están contenidas en el paquete `java.io`, y las clases para crear applets están en el paquete `java.applet`. Poner estas clases en paquetes las organiza convenientemente para el compilador y para nosotros.

Además, estas clases predefinidas están íntegramente documentadas, en páginas Web públicas, generadas automáticamente mediante la herramienta **javadoc** (aplicación integrada en el SDK de Java). En Internet podemos encontrar rápidamente estos documentos, para cualquier clase predefinida, para tener una visión general de los métodos y atributos de dicha clase, y así usarla efectivamente.

De entre todas las API's que usaremos, las más importantes son: `String` y `Math`.

La clase String

En contraposición, un objeto de la clase **StringBuffer** (no la veremos de momento) representa una cadena cuyo tamaño puede variar.

Tenemos que acostumbrarnos a esta sentencia, saldrá en casi todos los programas. Sirve para imprimir por pantalla la cadena que le introducimos como argumento.

Si queremos que la imprima con salto de línea, escribiremos `System.out.println`, si no, `System.out.print`.

No enumeramos más métodos de esta clase porque la estudiaremos a fondo en temas posteriores.

Un objeto **String** representa una cadena alfanumérica de un valor constante que no puede ser cambiada después de haber sido creada.

Java posee gran capacidad para el manejo de cadenas dentro de su clase **String**. La mayoría de las funciones relacionadas con cadenas esperan valores **String** como argumentos y devuelven valores **String**.

Por ejemplo, la sentencia que más emplearemos a lo largo de todo el curso, espera un **string** como argumento:

```
System.out.println(cadena);
```

Tal como uno puede imaginarse, las cadenas pueden ser muy complejas, existiendo muchas funciones muy útiles para trabajar con ellas y, afortunadamente, la mayoría están codificadas en la clase **String**.

Veamos algunas de sus funciones básicas:

```
int length();
char charAt(int indice);

boolean equals(Object obj);
int compareTo(String str2);
```

...

- Devuelve la longitud de la cadena.
- Devuelve el carácter que se encuentra en la posición que se indica en índice.
- Compara dos strings.
- Devuelve un entero menor que cero si la cadena es léxicamente menor que `str2`. Devuelve cero si las dos cadenas son léxicamente iguales y un entero mayor que cero si la cadena es léxicamente mayor que `str2`.

La clase Math

La clase **Math** representa la librería matemática de Java.

Si se importa la clase, se tiene acceso al conjunto de funciones matemáticas estándar. Algunas de las más usadas son:

```
Math.abs(x)
Math.sin(double)
Math.cos(double)
Math.tan(double)
Math.asin(double)
Math.acos(double)
Math.atan(double)
Math.atan2(double, double)
Math.exp(double)
Math.log(double)
Math.sqrt(double)
Math.pow(double, double)

Math.round(x)
Math.random()

Math.max(a, b)
Math.min(a, b)
Math.E
Math.PI
```

- Calcula el valor absoluto.
- Calcula el seno de un ángulo.
- Calcula el coseno de un ángulo.
- Calcula la tangente de un ángulo.
- Calcula el arco seno de un número.
- Calcula el arco coseno de un número.
- Calcula el arco tangente de un número.
- Convierte coordenadas cartesianas a polares.
- Calcula el valor de e elevado a un número.
- Calcula el logaritmo neperiano de un número.
- Calcula la raíz cuadrada positiva de un número.
- Calcula el valor del primer número elevado al segundo.
- Calcula el entero más cercano a un número.
- Devuelve un valor aleatorio mayor o igual que 0 y menor que 1.
- Calcula el máximo entre los dos valores.
- Calcula el mínimo entre los dos valores.
- Devuelve el valor del número e .
- Devuelve el valor del número π .

Ya estudiaremos lo que significa **importar**. De momento debes entender que es una acción necesaria para usar ciertas clases predefinidas.

Observa que todos estos son **métodos** de la clase **Math**.

TEMA 3: TIPOS, VALORES Y VARIABLES

Los tipos **referenciados** se llaman así porque el valor de una variable de referencia es un puntero (una referencia) hacia el valor real. En Java tenemos los **arrays**, las **clases** y los **interfaces** como tipos de datos referenciados.

Debemos tener cuidado de no emplear ninguna de las **palabras reservadas** del lenguaje, como **final** o **public**. Veremos la lista completa de estas palabras al final del tema

Ahorraremos tiempo si utilizamos nombres que indiquen para qué sirve la variable.

Los nombres de constantes se escriben en mayúsculas. Si se trata de un nombre compuesto, se separa cada palabra con el carácter guión bajo: _

Veremos más aplicaciones de la palabra reservada **final** en temas posteriores.

En Java existen dos categorías de datos principales: los tipos primitivos y los tipos referenciados. En este tema vamos a prestar atención a los primeros.

3.1 Valores, variables y constantes

Como ya sabemos, un programa maneja **valores**. De hecho, lo que se desea hacer con un programa es manejar los datos, de forma apropiada, para cambiarlos, hacer cálculos, presentarlos, solicitarlos al usuario, escribirlos en un disco, enviarlos por una red, etc.

Para poder manejar los valores en un programa se guardan en **variables**. Una variable guarda un único valor.

Una variable queda determinada por:

- Un **nombre**, que permitirá referirse a la misma. Este debe comenzar por minúscula, y emplear los caracteres permitidos.
- Un **tipo**, que permite conocer qué valores se pueden guardar en dicha variable.
- Un rango de valores que puede permitir. Esta característica se deriva directamente del tipo de variable que manejemos.

Así, la declaración de una variable tiene la forma:

```
tipoVariable nombreVariable;
```

Una vez declarada la variable ya se puede utilizar en cualquier lugar del programa poniendo su nombre. Siempre que se utilice el nombre de una variable es como si se pudiese el valor que tiene dicha variable.

También se puede asignar un valor inicial en la declaración de variables:

```
tipoVariable nombreVariable = valorInicial;
```

Si sólo se declara, toma un valor inicial por defecto:

```
Enteros → 0
Reales → 0.0
Booleanos → false
Caracteres → \n0000 //carácter nulo.
Referencias → null
```

Los tipos simples son exclusivamente los **enteros**, **reales**, **boolean** y **carácter**.

Constantes

Si en la declaración de una variable incluimos, al comienzo de esta, la palabra reservada **final**, indicamos con esto que el valor de la misma ya no se puede cambiar después, es decir, no se puede asignar un valor diferente más adelante en el programa, convirtiéndose, por tanto, en una **constante**.

```
→ final NOMBRE_CONSTANTE = valorConstante;
```

Si durante el programa se intenta modificar una constante, habrá un **error de compilación**.

3.2 Tipos primitivos

Tipos enteros

- Permiten representar números enteros positivos y negativos con distintos rangos de valores.
- Java tiene cuatro formas distintas de representar un número entero según sea su longitud:

Tipo	Tamaño	Rango de valores
byte	8 bits	[-128..127]
short	16 bits	[-32768..32767]
int	32 bits	[-2147483658..2147483647]
long	64 bits	[-9223372036854775808.. +9223372036854775807]

IMPORTANTE: la diferencia entre el **valor** de un número (único), y sus diferentes **representaciones** según los sistemas de numeración.

Las formas en que se pueden escribir valores de cada uno de los tipos, se denominan "**literales**".

- En los programas los valores enteros se pueden escribir en **decimal**, **hexadecimal** u **octal** (aunque estos dos últimos no son muy habituales en la práctica).

La forma normal es la decimal, si empleamos octal lo indicamos añadiendo el dígito cero (0) delante, y si lo hacemos en hexadecimal lo que añadimos es un (0x).

Ejemplo: 53 en decimal también se puede escribir 0x35 en hexadecimal o 065 en octal.

Además, Si al escribir un entero no se indica nada se supone que el valor pertenece al tipo `int`. A un número entero se le debe añadir detrás el carácter 'l' ó 'L' (una letra en minúscula o mayúscula) para indicar que el valor es del tipo `long`.

Tipos reales

- Permiten representar números reales.
- En Java 2 existen dos tipos de números reales. La diferencia entre ambos está en el número de decimales que se pueden expresar y en los rangos de valores posibles. Tenemos:

Tipo	Tamaño	Rango de valores y número de cifras significativas
float	32 bits	[±3.40282347e+38 a ±1.40239846e-45] 6 cifras significativas.
double	64 bits	[±1.79769313486231570e+308 a ±4.94065645841246544e-324] 15 cifras significativas.

Se trata de reales en Coma Flotante. Estudiarás los entresijos de este método de representación en asignaturas posteriores, como FDOOR.

- Para escribir valores reales en Java se puede hacer de las siguientes formas: 1e2, 2., .54, 0.45, 3.14, 56.34E-45. Es decir, un número real en Java siempre debe tener un punto decimal o, si no lo tiene, tiene un exponente indicado por la letra `e` minúscula o `E` mayúscula.

La notación `1e2` significa $1 \cdot 10^2$, de la misma forma que `56.34E-45` significa $56.34 \cdot 10^{-45}$.

Además, Al igual que ocurría con los enteros, si no indicamos nada, se dará por supuesto que el valor representado es de tipo `double`. Si se desea que se interpreten como tipo `float` se debe añadir un carácter '`f`' ó '`F`' detrás del valor. Los ejemplos de antes podrían quedar: `1e2f`, `2.f`, `.54f`, `0.45f`, `3.14f`, `56.34E-45f`. Del mismo modo se puede añadir una '`d`' ó '`D`' para indicar explícitamente que el valor es del tipo `double`, de la siguiente forma: `1e2d`, `2.d`, `.54d`, `0.45d`, `3.14d`, `56.34E-45d`.

Tipo booleano

- Se usan en operaciones de comparación y lógicas. En Java se llama `boolean`.
- Sólo toman dos valores, verdadero (`true`) y falso (`false`).

Tipo carácter

- Permite representar cualquier carácter individual. En Java se llama `char`.
- En Java, los caracteres se representan utilizando la tabla de **caracteres Unicode**, creada para que se pudiese escribir en cualquier idioma del mundo. Así, contiene todos los caracteres con los que se escribe en español, como las letras ñ, Ñ, ü, ó, etc.
- Cada carácter ocupan 16 bits y toma un valor de `\u0000` a `\uFFFF`, aunque en la práctica es más cómodo llamarlos entre apóstrofes, mediante su representación: '`ñ`', '`Ñ`', '`ü`', '`ó`', etc.
- Los caracteres están ordenados '`A`' < '`B`'; '`a`' < '`b`'; '`0`' < '`1`' ... < '`9`'.
- Algunos caracteres tienen una representación especial. Los vemos en la siguiente tabla.

Descripción	Representación	Valor Unicode (en hexadecimal)
Carácter Unicode	<code>\u0000</code>	
Número octal	<code>\ddd</code>	
Barra invertida	<code>\\</code>	<code>\u005C</code>
Continuación	<code>\</code>	<code>\</code>
Retroceso	<code>\b</code>	<code>\u0008</code>
Retorno de carro	<code>\r</code>	<code>\u000D</code>
Tabulación horizontal	<code>\t</code>	<code>\u0009</code>
Salto de línea	<code>\n</code>	<code>\u000A</code>
Comillas simples	<code>\'</code>	<code>\u0027</code>
Comillas dobles	<code>\"</code>	<code>\u0022</code>
Números arábigos ASCII	<code>0-9</code>	<code>\u0030</code> a <code>\u0039</code>
Alfabeto ASCII en mayúsculas	<code>A.-Z</code>	<code>\u0041</code> a <code>\u005A</code>
Alfabeto ASCII en minúsculas	<code>a.-z</code>	<code>\u0061</code> a <code>\u007A</code>

Un texto en Java pertenece a la clase `String` y se expresa como el texto entre comillas dobles.

Un texto puede estar compuesto por 0 ó más caracteres.

Así por ejemplo, para inicializar una variable de nombre `comi`, que contenga un carácter `"` (unas comillas), podemos hacerlo de estas tres maneras:

```
char comi= "\" ";
char comi=0x0022;
char comi=34;
```

Siguiendo su representación, su valor hexadecimal, o su valor decimal.

3.3 Operadores

Sobre **caracteres** no existe ninguna operación predefinida. Todas son del API de Java. Por ejemplo: `isLowerCase(c)`, `isUpperCase(c)`, `toLowerCase(c)`, etc,...

Por ejemplo, los operadores aritméticos devuelven números, cuyo tipo depende del tipo de los operandos (enteros o reales).

Recuerda, por ejemplo, esta máxima: las operaciones entre enteros siempre devuelven enteros.

El lenguaje Java extiende la definición del operador `+` para incluir la concatenación de cadenas.

Aparte de todos estos operadores predefinidos, no debemos olvidar que también hay operaciones sobre reales y enteros disponibles en el API de Java (todas las de la clase `Math`).

Los operadores realizan algunas funciones en uno o dos operandos.

Los operadores que requieren un solo operando se llaman **operadores unarios**, y pueden utilizar en Java la notación de prefijo (el operador aparece antes que su operando) o de sufijo (después).

```
operador operando    (prefijo)
operando operador    (sufijo)
```

Los operadores que requieren dos operandos se llaman **binarios**. Su notación es:

```
op1 operador op2
```

Además de realizar una operación, los operandos devuelven un valor (el resultado). El valor y su tipo dependen del operador y del tipo de los operandos. Por esta razón se suele decir que “una operación evalúa su resultado”.

Es muy útil dividir los operadores Java en las siguientes categorías: aritméticos, relacionales, condicionales ó lógicos, de desplazamiento, y de asignación.

Operadores aritméticos

El lenguaje Java soporta varios operadores aritméticos en todos los números enteros y reales, incluyendo `+` (suma), `-` (resta), `*` (multiplicación), `/` (división), y `%` (módulo).

Esta tabla resume todas las operaciones aritméticas binarias en Java:

Operador	Uso	Descripción
<code>+</code>	<code>op1 + op2</code>	Suma <code>op1</code> y <code>op2</code>
<code>-</code>	<code>op1 - op2</code>	Resta <code>op2</code> de <code>op1</code>
<code>*</code>	<code>op1 * op2</code>	Multiplica <code>op1</code> por <code>op2</code>
<code>/</code>	<code>op1 / op2</code>	Divide <code>op1</code> entre <code>op2</code>
<code>%</code>	<code>op1 % op2</code>	Obtiene el resto de dividir <code>op1</code> entre <code>op2</code>

Observaciones

- Ojo con la división entre enteros. Por ejemplo, tenemos que:

$6/3=2$ pero $5/3=1$ y sobran 2

Orden de precedencia

El orden de precedencia a la hora de realizar operaciones aritméticas es:

- 1º los paréntesis
- 2º `-` menos unario.
- 3º `*`, `/`, `%` (los tres con igual prioridad).
- 4º `+`, `-` (los dos con igual prioridad).

en caso de empate la prioridad es de izquierda a derecha.

Los operadores + y - tienen sus versiones unarias que seleccionan el signo del operando:

Operador	Uso	Descripción
+	+ op	Indica un valor positivo
-	- op	Niega el operando

Observaciones

- El operador unario + también promociona el operando a un entero int, si es un byte, un short ó un char.

Además existen dos operadores de atajos aritméticos, ++ que incrementa en uno su operando, y -- que lo decrementa:

Operador	Uso	Descripción
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
--	op --	Decrementa op en 1; evalúa el valor antes de decrementar
--	-- op	Decrementa op en 1; evalúa el valor después de decrementar

La diferencia entre el preincremento y el postincremento se puede entender mediante el siguiente ejemplo:

```
System.out.println(i++);
```

es equivalente a:

```
System.out.println(i); i=i+1;
```

y análogamente tenemos el otro caso:

```
System.out.println(++i);
```

es equivalente a:

```
i=i+1; System.out.println(i);
```

Operadores relacionales

Los operadores de relación permiten comparar valores enteros, reales ó caracteres. El resultado de una operación con estos operadores es un booleano indicando si es cierta o falsa la relación. Además, los dos últimos operadores también pueden comparar booleanos.

Operador	Uso	Devuelve true si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Observaciones

- La prioridad operadores relacionales menor que ningún otro operador.

Operadores condicionales ó lógicos

Estos realizan operaciones sobre booleanos, devolviendo un booleano. Son:

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son ambos true. Si op1 vale false, no se evalúa op2.
	op1 op2	uno de los dos es true. Si op1 vale true, no se evalúa op2.
!	! op	op es falso
&	op1 & op2	op1 y op2 son ambos true. Siempre evalúa ambas expresiones.
	op1 op2	uno de los dos es true. Siempre evalúa ambas expresiones.
^	op1 ^ op2	op1 y op2 tienen valor distinto. Es decir, si uno vale true y el otro false.

Operaciones lógicas equivalentes:

Operación AND con cortocircuito.

Operación OR con cortocircuito.

Operación NOT.

Operación AND sin cortocircuito.

Operación AND sin cortocircuito.

Operación XOR.

Nota: "cortocircuito" significa que se realiza una evaluación parcial (empezando por la izquierda) para saber el resultado final.

Operadores lógicos sobre bits

Cuando se trata de trabajar sobre bits (por lo tanto, valores unitarios 0 ó 1), las operaciones lógicas se realizan con estos operadores:

Operador	Uso	Operación
&	op1 & op2	AND lógico de bits.
	op1 op2	OR lógico de bits
^	op1 ^ op2	XOR lógico de bits
~	~op	Complemento de bits.

Estas operaciones lógicas entre bits se verán en profundidad en la asignatura CEDG.

Operadores de desplazamiento

Los operadores de desplazamiento permiten realizar una manipulación de los bits de los datos. Son:

Operador	Uso	Operación
>>	op1 >> op2	Los bits de op1 los desplaza op2 posiciones a la derecha.
<<	op1 << op2	Los bits de op1 los desplaza op2 posiciones a la izquierda.
>>>	op1 >>> op2	Los bits de op1 los desplaza op2 posiciones a la derecha. SIN SIGNO

Observaciones

- Un desplazamiento a la derecha de un bit es equivalente (pero más eficiente) a dividir el operando de la izquierda por dos.
- Un desplazamiento a la izquierda de un bit es equivalente a multiplicar el operando de la izquierda por dos.

Ejemplo

13>>1 desplaza los bits del entero 13 una posición a la derecha.

La representación binaria del número 13 es 1101. Al desplazar los bits, el resultado es el binario 110 (observa que el bit situado más a la derecha desaparece), que en decimal es el entero 6.

Operador de asignación de valor

Si se quiere guardar un valor distinto en una variable se utiliza el operador de asignación de valor. Este operador es el carácter “igual” y se utiliza de la siguiente forma:

```
nombreVariable = nuevoValor;
```

Donde nuevoValor será cualquier expresión o cálculo que se desee, cuyo resultado se vaya a guardar en la variable.

Es muy importante entender que el símbolo = no es como el símbolo matemático de igualdad. Aquí, al utilizarlo en una sentencia, significa lo siguiente: “haz el cálculo de la expresión que se encuentra a la derecha del igual y, después, guarda el valor calculado en la variable que hay a la izquierda”.

Además del operador de asignación básico, Java proporciona varios operadores de asignación que permiten realizar operaciones aritméticas, lógicas ó de bit, y una asignación al mismo tiempo. Estos operadores son:

Operador	Uso	Equivale a
+=	var += op	var = var + op
-=	var -= op	var = var - op
*=	var *= op	var = var * op
/=	var /= op	var = var / op
%=	var %= op	var = var % op
&=	var &= op	var = var & op
=	var = op	var = var op
^=	var ^= op	var = var ^ op
<<=	var <<= op	var = var << op
>>=	var >>= op	var = var >> op
>>>=	var >>>= op	var = var >>> op

Ejemplos

```
i=i+3    i+=3
i=i*2    i*=2
i=i-4    i-=4
etc.
```

Observaciones

- En concreto recuerda que existen dos operaciones de especial importancia en programación que son sumar y restar 1. Para ellas Java ofrece otra posibilidad de abreviatura:

```
i=i+1    i+=1    i++    ó    ++i    (postincremento ó preincremento)
i=i-1    i-=1    i--    ó    --i    (postdecremento ó predecremento)
```

IMPORTANTE: El tipo del valor resultado del cálculo que esté a la derecha debe coincidir con el tipo de la variable.

Otros operadores

Los siguientes son operadores cuyo uso veremos en profundidad en otros temas:

Operador	Uso	Descripción
? :	condicion ? op1 : op2	Si la condicion vale true, devuelve el valor de op1. Si no, devuelve el valor de op2.
[]	tipo [] Clase []	Declara un array, todavía sin tamaño conocido, de elementos de un determinado tipo u objetos de la clase.
[]	new tipo[expr-entera] new Clase[expr-entera]	Crea un objeto array del número de elementos que indique la expresión expr-entera. Se crea un array de elementos del tipo u objetos de la clase indicados.
[]	var[expr-entera]	Accede al elemento de la posición var en el array. Los índices van desde 0 al tamaño del array menos 1.
.	var.nombre	Es una referencia al atributo nombre del objeto var.
()	nombre(parametros)	Llama al método con el nombre dado utilizando como argumentos de la llamada los que se ponen entre paréntesis. Los parámetros pueden no existir. Si hay más de uno se ponen separados por comas.
(tipo)	(tipo)op	Convierte el valor de op al tipo dado.
(Clase)	(Clase)objeto	Convierte el objeto resultado en un objeto de la clase dada.
new	new Clase new nombre[]	Crea un nuevo objeto de una clase o un nuevo array.
instanceOf	objeto instanceof Clase	Devuelve true si el objeto dado es una instancia de la clase indicada o deriva de la clase indicada.

3.4 Conversiones de tipo

Las operaciones determinan el tipo a devolver a partir del tipo de los operandos.

Ejemplo

4/3 -> 1 (int, int) -> int Esto es, un cociente entre dos enteros (int) devuelve un entero (int).

Pero a veces, no es el resultado deseado, y es necesario realizar algunas conversiones.

Así, en programación existe la **Conversión de tipos** que puede llevarse a cabo de dos maneras distintas:

- Implícita (definida en el lenguaje).
- Explícita (casting o promoción), forzada por el programador.

Conversión implícita

La conversión implícita sólo se aplica a tipos primitivos y es realizada automáticamente por el lenguaje Java sin que nosotros hagamos nada.

Si es necesaria, el compilador realizará esta conversión de forma automática.

Ejemplo 1

5.0/4 -> 1.25

El 4 se convierte "automáticamente" en 4.0 para poder dividir

Ejemplo 2

Para realizar el cálculo de conversión de grados Fahrenheit a Centígrados podemos escribir, en Java:

`gradosC=(gradosF-32)*5/9;` Con las variables `gradosC` y `gradosF` de tipo `double`.

Al realizar la resta `gradosF-32` se realiza una conversión automática del número 32, de tipo `int` a tipo `double` (el tipo mayor, para que se pueda realizar la operación). Así, el resultado de esa resta es también un `double`. De igual modo, al multiplicar por 5, éste asciende a `double`, lo mismo que ocurre al dividir el resultado entre el 9.

Por esto recibe también el nombre de **conversión ascendente**.

La conversión implícita siempre es de tipos "pequeños" a "grandes".

```

byte    -> short, int, long, float, o double
short   -> int, long, float o double
char    -> int, long, float, o double
int     -> long, float o double
long    -> float o double
float   -> double

```

Ojo: Se puede perder información en los siguientes casos:

Esto es debido a la mayor precisión de `int` y `long`

- En el paso de `long` a `float` o `double`.
- En el paso de `int` a `float`.

A este tipo de conversión se le llama **Promoción o Casting**.

Conversión explícita

Si se desea realizar conversión descendente (de tipos "grandes" a "pequeños"), ha de ser explícita.

Es posible forzar la conversión de tipos, de la siguiente manera:

`(nombredeTipo) expresion;`

Así se convierte el valor de la expresión a un valor del tipo entre paréntesis.

Ejemplos

```

(int)10.0;  -> devuelve el entero 10.
(double)10; -> devuelve el double 10.0.
(char)('A'+2); -> devuelve el char C.
(char)98;   -> devuelve 'b'.

```

Un tipo `char` siempre se puede utilizar en una expresión junto con números enteros, evaluándose como entero. En el ejemplo, si no forzamos la conversión a `char`, el resultado devolvería el entero 67.

Observaciones

- Forzar la conversión de tipos puede hacer que se pierda parte del valor de un cálculo o de una variable.
- La promoción tiene mayor prioridad.

Ejemplo

`(float)5/4;` -> 1.25 (formato `float`)

El 5 se convierte por la promoción forzada a 5.0, el 4 se convierte "automáticamente".

3.5 Expresiones, orden de precedencia

Una expresión es una serie de variables, operadores y llamadas a métodos (construida de acuerdo a la sintaxis del lenguaje) que evalúa a un valor sencillo.

Una expresión permite realizar tanto operaciones simples como operaciones complejas entre valores utilizando distintos operadores. Permite, por ejemplo, representar fórmulas matemáticas que se utilizan para realizar cálculos.

En Java, una expresión puede ser tan compleja como sea necesario.

Entre las expresiones, merecen especial atención las **aritmético-lógicas**. Se trata de expresiones que devuelven un valor booleano, y donde se utilizan operadores aritméticos, relacionales y lógicos.

Toda expresión se evalúa a un valor de forma estricta. Cómo se evalúa una expresión depende del orden de prioridad de los operadores que contenga.

El orden de precedencia (o prioridad) de los operadores lo vemos en la siguiente tabla:

Precedencia	Tipo de operador	Operadores
1	Operadores prefijo o postfijo	++ -- + - ~ ! [] . (tipo) expr++ expr--
2	Operadores unarios	++expr --expr +expr -expr
3	Creación Conversión de tipo o clase	new (tipo)expr
4	Multiplicativos	* / %
5	Aditivos	+ -
6	Desplazamiento	<< >> >>>
7	Relacionales	< > <= >= instanceof
8	Igualdad	== !=
9	AND de bits	&
10	OR exclusivo de bits	^
11	OR de bits	
12	AND lógico	&&
13	OR lógico	
14	Condicional	? :
15	Asignación	= += -= *= /= %= &= ^= = <<= >>= >>>=

En cualquiera de los lugares donde se usa una expresión se puede utilizar un nombre de variable o una llamada a un método que devuelva un valor del tipo apropiado, como veremos en temas posteriores.

Observaciones

- Los operadores con mayor precedencia se evalúan antes que los operadores con una precedencia relativamente menor.
- Los operadores con la misma precedencia se evalúan de izquierda a derecha.
- Si se desea que una evaluación se realice en un orden específico, se pueden utilizar paréntesis. Si en una expresión hay paréntesis, siempre se empieza a evaluar por los paréntesis más internos.

3.6 Palabras reservadas en Java

Estas palabras tampoco podrán ser utilizadas como identificadores en otras facetas de nuestra programación, como cuando creamos métodos u objetos.

Como ya hemos comentado durante el tema, ciertas palabras no podemos usarlas como identificadores, por tratarse de palabras reservadas del lenguaje de programación con el que trabajamos.

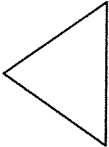
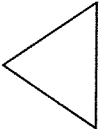
Las siguientes palabras están reservadas en Java:

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

Observaciones

- Las palabras null, false y true no son palabras reservadas, sino literales, pero igualmente no se pueden usar como identificadores.
- Aunque goto y const aparecen en la lista arriba, no se usan en el lenguaje.

CUADRO DE TIPOS PRIMITIVOS

CATEGORÍA	TIPO	TAMAÑO EN MEMORIA	RANGO DE VALORES	COMENTARIOS
Enteros	byte	8 bits		
	short	16 bits		
	int	32 bits		
	long	64 bits		
Reales	float	32 bits		
	double	64 bits		
Booleanos	boolean	---	<i>true ó false</i>	
Caracteres	char	16 bits	Códigos: 0 – 65535d 0x0000 – 0xFFFF	Cada código representa a un carácter de la tabla UNICODE . Ej.: char comi = '\n'; char comi = 0x0022; char comi = 34;

CUADRO DE OPERADORES

Binarios op1 ☐ op2

CLASIFICACIÓN DE OPERADORES	OPERAN SOBRE	DEVUELVEN	OPERADORES	DESCRIPCIÓN
Aritméticos	Enteros y reales	---	+ - * / %	Realizan una operación entre dos operandos.
Relacionales	Enteros, reales, caracteres, booleanos	Booleano	> >= < <= == !=	Devuelven <i>true</i> si se cumple la condición, si no, devuelven <i>false</i> .
Condicionales	Booleanos	Booleano	& & ^ &	Devuelven <i>true</i> si se cumple la condición, si no, devuelven <i>false</i> .
Desplazamiento	Int, long	---	>> << >>>	Desplazan los bits de un número dado, un número determinado de posiciones a izquierda ó derecha.

Unarios ☐ op ó op ☐

Aritméticos	Enteros y reales	---	+ -	Seleccionan el signo del operando.
Atajos aritméticos	Enteros y reales	---	++op --op op++ op--	Incrementan ó decrementan y después evalúan. Evalúan y después incrementan ó decrementan.
Condicionales	Booleanos	Booleano	!	Devuelve <i>true</i> si op es falso.

TEMA 4: MÉTODOS

4.1 Generalidades

- ¿Qué son los métodos?

Son funciones que determinan el comportamiento de un objeto, manipulando sus atributos.

Los métodos se declaran y definen en las clases. Así, cualquier objeto de esa clase tendrá disponibles esos métodos y podrán ser invocados.

- ¿Cómo se llama a un método?

Normalmente llamaremos a un método poniendo:

```
objeto.metodo(argumentos)
```

Si el método es estático (`static`) pondremos:

```
clase.metodo(argumentos)
```

Si estamos dentro de la misma clase donde se define el método entonces basta poner el nombre:

```
metodo(argumentos)
```

- ¿Qué devuelve un método?

Pueden devolver:

- Un valor de cualquier tipo o clase: para ello se antepone al nombre del método el nombre del tipo o clase a devolver.
- No devolver nada: se antepone al nombre del método la palabra `void`.

4.2 Definición de métodos

Un método tiene dos partes claramente diferenciadas, la **cabecera** y el **cuerpo**.

Cabecera

La cabecera está formada por:

- Accesibilidad del método: en el tema 6 describiremos las posibilidades al respecto.
- Tipo del valor a devolver: si el método devuelve un valor, hay que poner el tipo que se devuelve (`byte`, `short`, `int`, `long`, `float`, `double`, `boolean` ó `char`).

Si no devuelve valor, se pone `void`.

- Nombre del método: será el identificador con el que se invocará (usará) al método.
- Parámetros: los parámetros que requiere el método para su ejecución. Son valores que se utilizarán en el código que compone el cuerpo del método.

Los parámetros aparecen siempre entre paréntesis.

- Tipo de excepción: si el método puede lanzar excepciones, hay que indicarlo. Lo veremos en el tema 5.

Es decir, la **cabecera** de un método tiene la siguiente forma:

```
acceso tipo nombre (parámetros) excepciones
```

El comando `static` lo veremos en el tema 6.

De momento debemos acostumbrarnos a ver la palabra `public` en los métodos.

Es aconsejable que el nombre sea descriptivo de lo que hace el método, para facilitar la legibilidad de los programas.

Ejemplos

```
void vueltas (int i) {...}
boolean encendida () {...}
int calcula(double a, double b, boolean x) throws Exception {...}
```

Observaciones

- Las palabras relativas a la accesibilidad del método y al tipo de excepción, pueden no aparecer.
- Puede haber métodos que no requieran parámetros. En tal caso pondremos los paréntesis vacíos ().

Cuerpo

El **cuerpo** de un método está formado por una secuencia de instrucciones entre llaves (bloque).

Dentro de este bloque se pueden declarar variables que llamaremos **variables automáticas** ó **locales** y que existirán mientras estemos dentro del método. En cuanto salimos del método todas las variables locales se destruyen.

Salimos de un método cuando llegamos a } o hasta encontrar un `return`.

Signatura

Dentro de una clase, los métodos se identifican unívocamente por la **signatura** del método. Ésta es la forma de describir el método en base a su nombre y a los tipos de parámetros que recibe.

Por lo tanto, la signatura de un método es la tupla compuesta por

<clase, nombre, tiposDeLosParámetros>

Lo importante de este concepto es que **no puede haber dos métodos con igual signatura**. Esto en la práctica significa que no pueden existir dentro de una clase dos métodos con el mismo nombre y los mismos tipos de parámetros de entrada.

Observaciones

- La accesibilidad del método, el tipo de valor de retorno, y el tipo de la excepción lanzada NO forman parte de la signatura.
- Cuando tenemos dos métodos dentro de una clase con igual nombre pero distinto tipo de parámetros de entrada decimos que hemos **sobrecargado** el método.

Importante: los parámetros definidos en la cabecera de un método se comportan como variables locales dentro del método.

Si sucede esto el programa no compila.

4.3 Invocación de métodos

Para invocar un método, existen tres mecanismos distintos:

- Fuera de la clase en la que se define el método, se pone el nombre de un objeto que tenga el método, un punto y luego el nombre del método que se desea invocar:

```
objeto.metodo(argumentos)
```

- En el caso de que se trate de un método de clase (declarado `static`), se pone el nombre de la clase que define el método, un punto, y el nombre del método a invocar:

```
clase.metodo(argumentos)
```

- Cuando el método es utilizado en la misma clase que lo define, basta con poner directamente el nombre:

```
metodo(argumentos)
```

Como se puede ver, en cualquiera de los casos, después del nombre se ponen entre paréntesis los argumentos que necesite el método. Incluso si se trata de un método sin parámetros declarados, se ponen los paréntesis `()`.

Ejemplos

```
cl.vueltas(5) ...
Semaforo.encendida() ...
calcula(7.12, 1.8, false) ...
```

Funcionamiento de un método

Cuando llamamos a un método sucede lo siguiente:

1. En el punto del programa en el que se llama se calculan los valores de los argumentos.
2. Se salta al comienzo del método.
3. Se cargan los parámetros con los valores de los argumentos. La relación entre parámetros y argumentos se establece por el orden de aparición.
4. Se ejecuta el bloque hasta que se alcanza `return o }`
5. Si el método devuelve un valor, se sustituye la llamada por el valor devuelto.
6. Se sigue a continuación del punto en el que se llamó (siguiente instrucción tras la invocación del método).

4.4 El método `main()`

Hay un método especial, el `main()`, que es llamado por el intérprete de Java cuando ejecutamos un programa.

El intérprete Java invoca al método `main()`, y el código se ejecuta secuencialmente en el orden que aparecen las instrucciones

De momento, basta con que sepamos, como ya dijimos en el tema 2, que para que una clase se pueda ejecutar debe tener un método con la siguiente cabecera:

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

En ese caso al ejecutar la clase lo que se ejecutará será lo que contenga el método `main`.

La palabra reservada `static`, como ya se ha dicho, se estudiará en el tema 6.

Es el caso de los métodos de la clase `Math`.

Estas son llamadas a métodos definidos en el ejemplo anterior. Cada una de ellas siguiendo uno de los tres casos que acabamos de estudiar.

4.5 Parámetros y argumentos

En algunos libros a los **parámetros** se les llama "parámetros formales", y a los **argumentos** "parámetros reales".

Se ha visto que un método puede requerir ciertos datos para su ejecución. Los valores con los que se invoca el método constituyen los **argumentos** de la llamada. Cada vez que llamamos a un método los **parámetros** se cargan con los valores de los argumentos.

Así:

- **Parámetro:** el nombre asignado, al definir el método, a los valores que se usarán en el cuerpo del método.

Ejemplo:

```
public void mueve(float nuevov, float nuevoy, boolean b){...}
```

- **Argumento:** valor con el que se llama al método.

Ejemplo:

```
p.mueve (3.15, Math.sqrt(22.11*x)+9, true);
```

En el ejemplo anterior, al llamar al método mueve, se asigna al parámetro nuevov el valor 3.15, al parámetro b se le asigna true, etc.

Observaciones:

- En la creación de un método se pueden poner tantos parámetros como se deseen, separados por comas.
- **Importante:** Los parámetros que se definen en la cabecera de un método y los argumentos que se definen cada vez que se llama a ese método deben coincidir en número y tipo.

Paso de parámetros

Se ha explicado, que cuando se invoca a un método, se copian (se "pasan") los valores de los argumentos en los parámetros. A este respecto debemos hacer dos anotaciones importantes:

- Los tipos primitivos siempre se pasan **por valor**. Significa que se hace una copia del valor que tiene la variable y lo que utiliza el método es la copia, no el original.

Así, ninguna actuación del método sobre un parámetro de tipo primitivo puede variar el valor original.

- Los objetos siempre se pasan **por referencia**. Significa que al método se le pasa una referencia a la dirección donde está el objeto original.

Por tanto si el método modifica el objeto estará modificando el "original", no una copia.

Número variable de parámetros

Se ha explicado que el compilador comprueba el número y el tipo de los parámetros pasados a un método. Éste es el caso general, pero a veces es conveniente poder especificar un método al que se le pasa un número variable de argumentos.

Es posible admitir un número variable de argumentos, siempre que sean del mismo tipo, de la siguiente forma:

```
nombreDelMetodo(tipoParametro ... nombreParametro)
```

Así será la cabecera del método.

En caso contrario se producirá un error de compilación.

De todos modos, en este caso, la referencia original no se puede modificar, esto es, no se puede sustituir el objeto al que apunta por otro nuevo.

Ejemplo

Cabecera del método:

```
public void ponApellidos(String ... misApellidos) { ... }
```

...

Llamadas al método:

```
alumno1.ponApellidos("González", "Rodríguez");
alumno2.ponApellidos("Huecas", "Fernández", "Toribio");
```

Observaciones

- Los argumentos cuyo número no se especifica deben ser del mismo tipo.
- En un método se pueden especificar argumentos fijos junto con otros de número variable. En este caso los variables deben aparecer como el último parámetro del método.

4.6 Valor de retorno. La instrucción return

Los métodos que devuelven valores deben llevar obligatoriamente al menos una instrucción `return` seguida de la expresión a devolver:

```
return expresion;
```

El compilador de Java siempre comprueba que al menos esa instrucción `return` existe, y que el tipo de la `expresion` coincide con el declarado en la cabecera del método.

En los métodos que no devuelven nada (`void`) se puede usar la instrucción `return` sin expresión:

```
return;
```

Observaciones

- Hay que tener en cuenta que la instrucción de retorno termina la ejecución del cuerpo del método en que se encuentra. Esto significa que las instrucciones inmediatamente después de una instrucción `return`, nunca se ejecutarán. Salvo excepciones (usando estructuras de control para canalizar la ejecución de un método, dando la posibilidad de devolver diferentes valores, por ejemplo), poner una instrucción cualquiera después de un `return` provocará un error de compilación.
- En casos de bifurcaciones, el compilador comprobará también que cualquier posible bifurcación en el código del cuerpo del método dará lugar a una instrucción `return`.
Así, reiteramos, si un método devuelve un valor, el cuerpo debe estar codificado de forma que se asegure la ejecución de una instrucción `return`.
- No hay ninguna limitación del tipo a devolver, excepto que debe ser un único valor (no es posible devolver dos o más valores).
Así, tenemos métodos que devuelven valores de tipo simple, y otros que devuelven referencias a objetos.

Ejemplos

```
public boolean esHorarioDeMañana(){
    if(horario == Horario.MAÑANA)
        return true;
    if(horario == Horario.TARDE)
        return false;
}
```

Este primer ejemplo da error de compilación porque en un caso no hay `return`.

Esta es la codificación correcta del ejemplo anterior.

Este programa no imprime nada relativo al "grupo" si éste no ha sido asignado aun. Con la instrucción return termina la ejecución del método, aunque éste no devuelva nada.

```
public boolean esHorarioDeMañana(){
    return horario == Horario.MAÑANA;
}

public void imprime(){
    System.out.println("Datos personales: " + nombre
        + " " + apellidos
        + " " + annoDeNacimiento);

    if(grupo==null)
        return;
    System.out.println("Grupo; " + grupo + " " + horario);
}
```

4.7 Ámbito de las variables

En un método existen dos ámbitos, **estático** y **dinámico**. El estático existe en la definición del cuerpo. El dinámico aparece cuando se ejecuta el método.

Ámbito estático

Dicho de otro modo, el ámbito estático de un método tiene inicialmente todos los identificadores definidos en su clase.

El ámbito estático se crea en la definición de los métodos y contiene los identificadores declarados en el mismo (es decir, los nombres de atributos de la clase, de los parámetros, y de las variables locales).

Para usar un identificador en el cuerpo de un método debe estar incluido en el ámbito, por lo que dicho identificador debe haber sido declarado previamente.

Observaciones

- En el caso de los atributos, pueden ser utilizados en un método aunque textualmente en la clase aparezcan después de éste.

Ámbito dinámico

Recuerda que las **variables locales** o automáticas son las que se declaran en el cuerpo de un método.

El ámbito dinámico se crea al invocar un método. En él se introducen todos los identificadores de los parámetros del método, y se van incluyendo los nombres de las variables locales a medida que se ejecuta el cuerpo.

Cuando el método termina desaparece el ámbito dinámico. Es decir, el ámbito dinámico existe (está activo) mientras estemos ejecutando el método e incluso si se están ejecutando nuevos métodos invocados desde éste.

Existen unos métodos MUY importantes en programación, llamados **constructores**. Los estudiaremos en el tema 6.

TEMA 5: ESTRUCTURAS DE CONTROL

Vamos a introducirnos ya en el concepto de la **programación estructurada**.

La programación estructurada es una forma de implementar programas de forma clara. Para ello utiliza únicamente tres estructuras: **secuencial**, **selectiva** e **iterativa**.

Este estilo de programación tiene muchas ventajas. Entre ellas destacamos que los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación.

En esta tónica, usaremos las sentencias de control para determinar el orden en que se ejecutarán las otras sentencias dentro del programa. Java soporta varias sentencias de control de flujo, incluyendo:

Sentencias	palabras clave
condicionales	if-else, switch-case
bucles	for, while, do-while
manejo de excepciones	try-catch-finally, throw
salto	break, continue, label, return

5.1 La estructura de selección **if**

La sentencia /as gobernada por **if** se ejecuta si la **condicion** es verdadera. Se puede escribir así:

```
if (condicion) {
    sentencias;
}
```

Observaciones

- La **condicion** debe devolver un boolean (**true** o **false**) si no es así no compila.
- Si el bloque **sentencias** tiene una única instrucción no son necesarias las llaves.

Ejemplo

En el siguiente código:

```
if (nota < 5) {
    System.out.println("suspense");
}
```

Si, efectivamente, la variable **nota** es menor que 5, el sistema imprimirá por pantalla la palabra **suspense**.

Ante la duda,
pondremos llaves,
como en el ejemplo.

En este y en todos los
ejemplos del tema,
obviamos fragmentos
de código que
supondremos
correctos.

5.2 La estructura de selección `if..else`

Con esta, la sentencia /as gobernada por `if` se ejecuta si la `condicion` es verdadera. Pero ahora, además, si la condición es falsa, se ejecutará la sentencia /as gobernada por `else`.

Este uso particular de la sentencia `else` es la forma de capturarlo todo.

Mismas observaciones que para la sentencia `if` simple.

```
if (condicion) {
    sentencias1;
}
else {
    sentencias2;
}
```

Ejemplo

```
if (nota < 5) {
    System.out.println("suspenso");
}
else {
    System.out.println("aprobado");
}
```

Ahora, en caso de que `nota` sea menor que 5 imprimirá `suspenso`, pero si esta condición no se cumple, esto es, si `nota` es mayor o igual que 5, imprimirá `aprobado`.

5.3 La estructura de selección múltiple `if..else if`

El caso general de la sentencia `else` se usa para llevar a cabo una selección múltiple.

Veamos su forma:

```
if (condicion1) {
    sentencias1;
}
else if (condicion2){
    sentencias2;
}
else if (condicion3){
    sentencias3;
}
else if (condicion4){
    sentencias4;
}
...
else {
    sentenciasFinales;
}
```

Con esta, el sistema irá evaluando las condiciones de cada sentencia (`condicion1`, `condicion2`, `condicion3...`etc). En caso de encontrar una que devuelva `true`, ejecutará las sentencias correspondientes. Si ninguna lo hace, ejecutará las sentencias que haya en `else`.

Observaciones

Esta observación quedará muy clara con el ejemplo.

- Una sentencia `if` puede tener cualquier número de sentencias de acompañamiento `else if`.
- A veces la puede ocurrir, por ejemplo, que el valor que se compara para la condición satisfaga más de una de las expresiones que componen toda la sentencia `if`. Sin embargo, el sistema irá por orden, línea a línea, y una vez satisfecha una condición, ejecutará las sentencias apropiadas, e inmediatamente después saldrá fuera de la sentencia `if` sin evaluar las condiciones restantes.

Ejemplo

```
if (nota >= 9) {
    System.out.println("sobresaliente");
}
else if(nota >= 7){
    System.out.println("notable");
}
else if(nota >= 6){
    System.out.println("bien");
}
else if(nota >= 5){
    System.out.println("suficiente");
}
else {
    System.out.println("suspenso");
}
```

Si `nota` vale 9.5, está claro que satisfaceríamos todas las condiciones, pero como con la primera de ellas el sistema ya no atiende a las demás, la selección es perfecta.

5.4 La estructura de selección múltiple `switch`

La sentencia `switch` se utiliza para realizar sentencias condicionalmente basadas en alguna expresión.

Su forma es:

```
switch (expr) {
    case x:    sentencia1 ; break;
    case y:    sentencia2 ; break;
    case z:    sentencia3 ; break;
    ...
    default:   sentenciaDef ;
}
```

Donde `x`, `y`, `z`, etc serán expresiones del mismo tipo que `expr`, esto es, entero, booleano o carácter.

La sentencia `switch` evalúa la expresión `expr` y ejecuta la sentencia `case` apropiada. Mediante la sentencia `default` manejamos los valores que no se han manejado explícitamente por ninguna de las sentencias `case`.

Observaciones

- Las condiciones del `switch` sólo funcionan sobre enteros, booleanos o caracteres.
- Cada sentencia `case` debe ser única y el valor proporcionado a cada una debe ser del mismo tipo (entero, booleano o carácter) que el tipo de dato devuelto por la expresión proporcionada a la sentencia `switch`.

Hay escenarios en los que queremos que el control proceda secuencialmente a través de las sentencias case.

Decidir cuándo utilizar las sentencias if ó switch depende del juicio personal.

- La sentencia break hace que el control salga de la sentencia switch y continúe con la siguiente línea. Sin break el programa no saltaría al final y se ejecutarían todas las sentencias en cascada por debajo de la nuestra.
- La condición expr se evalúa N veces, hasta que encuentra un case apropiado, que además hace break en sus sentencias.
- El default es opcional.
- La sentencia switch siempre se puede realizar con varios if..else. La idea es que una sentencia switch tiene mejor legibilidad que la misma estructura escrita con sentencias if..else.

Ejemplo 1

```
int mes;

...

switch (mes) {
    case 1: System.out.println("Enero"); Break;
    case 2: System.out.println("Febrero"); Break;
    case 3: System.out.println("Marzo"); Break;
    case 4: System.out.println("Abril"); Break;
    case 5: System.out.println("Mayo"); Break;
    case 6: System.out.println("Junio"); Break;
    case 7: System.out.println("Julio"); Break;
    case 8: System.out.println("Agosto"); Break;
    case 9: System.out.println("Septiembre"); Break;
    case 10: System.out.println("Octubre"); Break;
    case 11: System.out.println("Noviembre"); Break;
    case 12: System.out.println("Diciembre"); Break;
    default: System.out.println("Mes incorrecto"); Break;
}
```

Ejemplo 2

```
int mes;
int numeroDias;
int anno;

...

switch (mes) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        numeroDias = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        numeroDias = 30;
        break;
    case 2:
        if( ((anno%4==0)&&!(anno%100==0))|| (anno%400==0) )
            numeroDias = 29;
        else
            numeroDias = 28;
        break;
    default: System.out.println("Mes incorrecto"); Break;
}
```

Recuerda que un año es bisiesto si es divisible por 4, pero no por 100, excepto los divisibles por 400, que también son bisiestos.

Comienzan aquí las sentencias de bucle.

5.5 La estructura de repetición **while**

Generalmente hablando, una sentencia **while** realiza una acción mientras se cumpla cierta condición.

La sintaxis general de esta sentencia es:

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

Si el bloque sentencias tiene una única instrucción no son necesarias las llaves.

Esto es, mientras **expresion** devuelva **true**, se ejecutará la sentencia.

Observaciones

- El bucle **while** se ejecuta 0 ó más veces.

Ejemplo

```
int i= 0;
while (i < 10) {
    System.out.println(i);
    i++;
}
```

Este fragmento de código imprime los números 0 a 9.

5.6 La estructura de repetición **for**

Podremos utilizar este bucle cuando conozcamos los límites del mismo, esto es: su instrucción de inicialización, su criterio de terminación y su instrucción de incremento.

Por ejemplo, se utiliza frecuentemente para iterar sobre los elementos de un array, o los caracteres de una cadena.

La forma general del bucle **for** puede expresarse así:

```
for (inicializacion; condicion; incremento) {
    sentencias;
}
```

De nuevo, si el bloque sentencias tiene una única instrucción no son necesarias las llaves.

Donde:

- **inicializacion** es la sentencia que inicializa el bucle. Se ejecuta una vez al iniciar el bucle.
- **condicion** determina cuando termina el bucle. Esta expresión se evalúa al principio de cada iteración del bucle. Cuando su expresión se evalúa a **false**, el bucle se termina.
- **incremento** es una expresión que se invoca en cada iteración del bucle. Es muy importante tener claro que el incremento se realiza siempre al terminar de ejecutar la iteración, antes de volver a comprobar la condición

La **condicion** también recibe el nombre de **terminacion**.

El **incremento** también recibe el nombre de **actualizacion**.

Cualquiera (o todos) de estos componentes puede ser una sentencia vacía (un punto y coma):

```
for( ; ; )
```

Observaciones

MUY IMPORTANTE!!!
La excepción es que la sentencia `continue` autoincrementa la variable de control en un bucle `for`, pero no en un bucle `while`. Por esto, en un bucle `while`, `continue` debe ir precedido por el incremento de esta variable (sentencia `i++` en el caso más típico) para que el comportamiento sea análogo al de un `continue` en un bucle `for`.

- El bucle `for` se ejecuta un número fijo de veces.
- La sentencia `while` y la sentencia `for` son equivalentes, con una pequeña excepción. Dicho de otra forma, todo lo que se puede programar con un `for` también se puede programar con un `while`:

```
for (expresion1, expresion2, expresion3)
    sentencias;

expresion1;
while (expresion2) {
    sentencias;
    expresion3;
}
```

Donde `expresion1` será la inicialización, `expresion2` será la condición y `expresion3` será la actualización.

- Si la variable estaba declarada antes del bucle, al acabar el `for` queda con el valor final.
- Si la variable se declara en la cabecera del `for` sólo existe en el bucle, por lo tanto su valor no podrá ser usado fuera.

Ejemplo

En este ejemplo vamos a conseguir que el ordenador realice la misma operación que antes habíamos implementado con la sentencia `while`.

```
for (int i=0; i<10; i++) {
    System.out.println(i);
}
```

Nuevamente, este fragmento imprime los números 0 a 9.

5.7 La estructura de repetición `do..while`

Éste es similar al bucle `while`, excepto en que la condición se evalúa al final del bucle.

Su estructura es:

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

Observaciones

- El bucle `do..while` se ejecuta 1 ó más veces.

Ejemplo

```
int i=0;
do {
    System.out.println(i);
    i++;
} while (i < 10);
```

Nuevamente, este fragmento imprime los números 0 a 9.

Esta sentencia se usa muy poco en la construcción de bucles pero tiene sus usos. Es conveniente usarla cuando el bucle debe ejecutarse al menos una vez, por ejemplo, para leer información de un fichero, donde al menos tendremos que leer un carácter.

Comenzamos aquí las instrucciones de ruptura.

Ya vimos a la instrucción `break` en acción dentro de la sentencia `switch`.

Las rupturas etiquetadas son una alternativa a la sentencia `goto`, utilizada en otros lenguajes de programación, que no está soportada por el lenguaje Java.

Esta instrucción sólo se puede llamar desde dentro de un bucle.

Recuerda lo explicado respecto al autoincremento de la variable de control usando `continue` en los dos bucles estudiados:

- `Continue` autoincrementa la variable de control cuando lo usamos en un bucle `for`.
- No la autoincrementa en un `while` (debemos añadir nosotros previamente la sentencia de incremento).

5.8 Instrucción `break`

La sentencia `break` hace que el control de flujo salte a la sentencia siguiente, esto es, cuando se ejecuta salimos directamente a la siguiente instrucción después del bucle.

`break` causa la salida del bucle más interno, esto es, el que lo encierra directamente.

Ejemplo

Vamos a ver un ejemplo de uso en un bucle `for`:

```
for (int n=0; ;n++) {
    System.out.println(n);
    if (n==5)
        break;
}
```

Este bucle imprimirá los números del 0 al 5.

Salto etiquetado con `break`

Además de esta funcionalidad básica, hay otra forma de `break` que hace que el flujo de control salte a una sentencia etiquetada. Se puede etiquetar una sentencia utilizando un identificador legal de Java (la etiqueta) seguido por dos puntos (`:`) antes de la sentencia.

Se usa para salir de una serie de estructuras anidadas de una sola vez. Su forma es:

```
etiqueta:
bucleExterno {
    ...
    buclesInternos {
        ...
        break etiqueta;
        ...
    }
}
```

Así, en un momento dado, provocará la salida de todo el bucle externo (no sólo del interno, como sucedería sin el uso de la etiqueta), continuando con la ejecución del código desde la última llave.

5.9 Instrucción `continue`

La sentencia `continue` hace que el control de flujo salte de la sentencia actual al comienzo del bucle, esto es, cuando se ejecuta pasamos directamente a la siguiente iteración del bucle.

Ejemplo

Vamos a ver un ejemplo de uso en un bucle `for`:

```
for (int n=0;n<100;n++) {
    if (n%2==0)
        continue;
    System.out.println(n);
}
```

Este bucle imprimirá sólo los números impares existentes entre 0 y 99.

Salto etiquetado con `continue`

De la misma manera que con `break`, `continue` se puede usar para salir de varios bucles anidados, a la vez, y situarnos en la siguiente iteración de aquel que deseemos.

Así, `continue` etiquetado salta el resto del cuerpo de la estructura que lo encierra y todas aquellas hasta seguir con la estructura etiquetada.

Su forma es:

```
etiqueta:
bucleExterno {
    ...
    buclesInternos {
        ...
        continue etiqueta;
        ...
    }
}
```

Cuando se ejecute ese `continue`, el programa seguirá con la siguiente iteración del bucle etiquetado, `bucleExterno`.

Para entender esta explicación es necesario ver antes el tema 8, en lo que se refiere a arrays.

Las llaves no son necesarias ya que, en este caso, el bucle for solo tiene una sentencia

Fijate que el array no tiene que ser obligatoriamente de enteros

Fijate que el break hace que el bucle acabe antes de tiempo y por eso no se imprimen todos los elementos del array.

5.10 Nueva estructura de repetición for

En la última versión de java aparece una nueva forma alternativa de implementar un bucle for. La mejor forma de entender cómo funciona es estudiar algunos ejemplos:

Ejemplo 1:

```
int[] datos= { 1, 2, 3, 5, 7, 5, 3, 2, 1 };
for (int n: datos) {
    System.out.print(n + " ");
}
```

Al ejecutar el anterior código veríamos por pantalla lo siguiente:

```
1 2 3 5 7 5 3 2 1
```

Por tanto, el **funcionamiento del nuevo bucle for** es el siguiente: se trata de definir un array de elementos (en este caso enteros, `int`). Tras ello se implementa el bucle for donde la variable de control del bucle va tomando el valor de cada uno de los elementos del array. Como se puede observar este bucle for solo necesita inicialización (no necesita ni condición ni actualización) ya que el bucle termina cuando la variable de control del bucle ha tomado todos los valores del array previamente definido.

Ejemplo 2:

En este caso definimos un conjunto de caracteres `char`

```
char[] letras= {'a', 'b', 'c', 'd'};
for (char n: letras) {
    System.out.print( n + " ");
}
```

Y por pantalla veríamos:

```
a b c d
```

Ejemplo 3:

Ahora definimos un conjunto de boolean:

```
boolean[] datos2= {true, false, true, true, false};
for (boolean n: datos2) {
    System.out.print( n + " ");
}
```

Y por pantalla se verá:

```
true false true true false
```

Ejemplo 4: Pregunta del parcial del grupo 15 (diciembre 2005)

Dado el siguiente código:

```
int[] datos= { 1, 2, 3, 5, 7, 5, 3, 2, 1 };
for (int n: datos) {
    if (n > 3) break;
    else System.out.print(n);
}
```

¿qué se imprime por pantalla?

```
123
```

TEMA 5b: RECURSIVIDAD

Un método puede llamar a cualquier otro método que esté definido, incluido él mismo. A esto se denomina **recursividad**.

Algunos problemas se resuelven mejor y resultan más claros y legibles cuando se programa su solución utilizando recursividad.

Ejemplo

Lo veremos muy bien mediante un ejemplo típico: el cálculo del factorial de un número.

El factorial de un número no negativo se define de la siguiente forma:

$$n! = \begin{cases} 1 & n = 0, n = 1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$

Como podemos observar, se trata en sí misma de una solución recursiva, en la que el factorial de un número se calcula en función del factorial del número anterior. Podemos programar un método que implemente este algoritmo utilizando la misma técnica:

```
public int factorial (int n) throws Exception {
    if (n<0)
        throw new Exception("Factorial no definido para negativos");
    if ((n==0) || (n==1))
        return 1;

    return n*factorial(n-1);
}
```

Como se puede observar, el método hace lo siguiente:

- Si el argumento (n) es negativo, el método lanza una excepción.
- Si el argumento es 0, ó es 1, se devuelve 1.
- En otro caso, se devuelve el valor del factorial de n-1, multiplicado por n.

En la ejecución, la invocación del método factorial provocará la aparición de sucesivos ámbitos dinámicos, con expresiones parciales por resolver.

Vamos a suponer ahora que en un momento se ha invocado el método con parámetro de valor 3:

```
factorial(3);
```

1. Como ya hemos dicho, se genera una variable llamada n en la que se guarda el valor 3. Se empieza a ejecutar el método, por lo que se compara el valor de n en las dos estructuras if, resultando falsas ambas condiciones, por lo que se llega a la última sentencia, donde sustituyendo el valor actual de n queda:

```
return 3*factorial(3-1);
```

2. En esta expresión aparece una nueva invocación al método factorial, por lo que se calcula el parámetro resolviendo la expresión entre paréntesis (2). Como en cualquier invocación de método, se genera un nuevo ámbito para dicho método, con lo que aparece una nueva variable n en la nueva llamada, en la que se guarda el valor 2. Análogamente al paso anterior, llegamos a una expresión:

```
return 2*factorial(2-1);
```

Estudiaremos este comportamiento (lanzamiento de excepciones), y sus posibilidades, en el siguiente capítulo.

No confundir esta n con la variable n de la llamada anterior, pues están en ámbitos distintos.

En general, las soluciones recursivas son más legibles y fáciles de escribir. Sin embargo, se necesitan más recursos de memoria porque van generando ámbitos nuevos con cada llamada.

Lo más importante es que igual que en cualquier otra llamada a un método, se genera un ámbito dinámico que desaparece cuando el cuerpo del método termina de ejecutarse.

El programa también podría terminar si, el número de veces que se llama recursivamente es muy elevado.

3. Aplicando ahora el mismo razonamiento, estamos en una nueva llamada al método `factorial`, con argumento 1, con lo que se genera un nuevo ámbito dinámico, con una nueva variable `n`, que ahora vale 1. Esta vez, con la ejecución de esta llamada al método, el segundo `if` evalúa su condición a `true`, con lo que se ejecuta la sentencia que devuelve el entero 1:

```
return 1;
```

Así termina el cuerpo de esta llamada y desaparece el ámbito más interno.

4. En ese momento se vuelve al punto donde se invocó al último `factorial` sustituyendo su invocación por el valor de retorno, y así sucesivamente, en cascada, evaluando y devolviendo todas las expresiones hasta llegar a la primera invocación.

Finalmente, el valor de `factorial(3)`, nuestra llamada inicial al método, es $3*2*1$.

Como cada vez que se invoca al un método recursivamente se crean ámbitos nuevos, hay que tener un especial cuidado en que en algún momento de todas las llamadas, la recursividad ya no se vuelva a producir.

En el caso del ejemplo, se puede observar que cada recursiva recibe un número menor y positivo, por lo que en algún momento dicho número valdrá 1. En ese momento ya no se volverá a llamar recursivamente, terminando la recursión.

Si la recursión no termina, es como un bucle infinito (que no acaba nunca). La diferencia es que la recursión consume recursos de máquina, pudiendo llegar a terminar el programa si los recursos son insuficientes. A esto se le llama desbordamiento (*stack overflow*).

TEMA 5c: EXCEPCIONES

Generalmente, cuando en tiempo de ejecución ocurre un **error grave**, el programa termina bruscamente.

Las **excepciones** son una herramienta que tiene el programador para tratar estos casos excepcionales en tiempo de ejecución y, si es posible, poder continuar la marcha normal del programa.

Ejemplo introductorio

Supongamos que tenemos las siguientes clases:

```
class Division{
    int x;
    int y;

    Division(int x, int y){
        this.x=x;
        this.y=y;
    }

    static double divide(int x, int y){
        return x/y;
    }
}

class PruebaDivision {
    public static void main(String[] args){
        int num = Integer.valueOf(args[0]);
        int den = Integer.valueOf(args[1]);
        double resultado = Division.divide(num, den);
        System.out.println("El resultado es " + resultado);
    }
}
```

Recuerda que la sentencia `int num=Integer.valueOf(args[0])` convierte el `String` contenido en `args[0]` en un entero, llamado `num`.

Ahora nos planteamos el siguiente **problema**: ¿Qué pasa si intento dividir por cero?.

En este caso, el programa acabaría de forma brusca, ya que está ocurriendo algo “extraño”. Sin embargo el programador puede prever estas situaciones excepcionales y ponerles remedio. Veamos ahora cómo podemos arreglar el programa para que no ocurra ningún error al intentar dividir por cero:

```
class Division{
    int x;
    int y;

    Division(int x, int y){
        this.x=x;
        this.y=y;
    }

    static double divide (int x, int y) throws Exception {
        if(y==0)
            throw new Exception("No se puede dividir por cero");
        return x/y;
    }
}

class PruebaDivision {
    public static void main(String[] args){
        int num = Integer.valueOf(args[0]);
        int den = Integer.valueOf(args[1]);
        double resultado = 0;
        try{
            double resultado=Division.divide(num, den);
            System.out.println("no hay excepción en este caso");
        }
        catch(Exception e){
            System.out.println("Se ha producido un error");
        }
        System.out.println("El resultado es " + resultado);
    }
}
```

Hay que tener mucho cuidado de que cuando se manejen las excepciones, y después de un bloque `try-catch-finally`, todas las variables que se usen SIEMPRE tengan valores predecibles.

Observa las nuevas líneas que hemos introducido:

- En ese caso estamos diciendo que, si intentamos dividir por cero, **se producirá una excepción** para indicarnos que algo extraño está ocurriendo. Nosotros le hemos añadido un **mensaje** para que nos informe algo más del **tipo de error** que se produce.
- Además, en la **cabecera del método** `divide` hemos añadido `throws Exception`. Lo que hace esto es indicarle a Java que durante la ejecución del método puede ocurrir algún error, se trata de una posibilidad pero no es seguro que se vaya a producir el error. Siempre que un método tiene la posibilidad de lanzar excepciones hay que declararlo obligatoriamente en la cabecera (tras los parámetros de entrada).

¿Y en el método `main` de la clase `PruebaDivision` qué ocurre?

- En principio seguiría la ejecución normal del programa. Al llegar a la línea que está dentro de `try`, intentaría hacer la división. Si es por cero, como nosotros le hemos dicho que en este caso se produce una excepción, lo que hace el programa es saltar automáticamente al `catch`, y ejecutar el `System.out.println`.
- En el caso de dividir entre cero no se mostraría el mensaje "no hay excepción en este caso", ya que al producirse la excepción el programa ignora automáticamente el resto de las líneas del `try` y salta directamente al `catch`.

Manejo de Excepciones

Nuestro programa de ejemplo es muy sencillo y sólo lanza una posible excepción (dividir entre cero) pero en general un bloque `try` puede recoger varias excepciones distintas "lanzadas" por diversos métodos contenidos en ese bloque. En ese caso debe haber un bloque `catch` para tratar cada tipo de excepción.

El **esquema genérico** es el siguiente:

```
try {
    ... ..
} catch (ClaseException1 e) {
    ... ..
} catch (ClaseException2 e) {
    ... ..
} catch (ClaseException3 e) {
    ... ..
} finally {
    ... ..
}
```

- Los diferentes `catch` se intentan en el orden en que aparecen hasta que uno de ellos casa con la excepción lanzada; después de casar con uno, los demás se olvidan. (es decir, solo se ejecuta un bloque `catch`). Casar significa que la excepción a agarrar es de la clase indicada o de una clase extensión de ella.
- La sentencia `catch (Exception e)` recoge cualquier excepción, sea cual sea, y por eso se suele poner justo la última antes del `finally`.
- El bloque `finally` es opcional, sólo puede aparecer una vez, y se ejecuta siempre, aunque no se ejecute ningún `catch`.
- Una excepción es un objeto `throwable` (que se puede lanzar) y se puede hacer con él todo lo que se puede hacer con un objeto.
- Si ningún bloque `catch` coincide con la excepción lanzada, dicha excepción se lanza fuera de la estructura `try-catch-finally`. Si no se captura en el método, este debe lanzarla delegándola en la declaración o, si no, dará un error al compilar. Para delegar una excepción el método debe declararlo en su cabecera.

Si una estructura de este tipo está en un método y se devuelve un valor desde la parte `try`, el `finally` se ejecuta incluso en este caso.

El método `main()` nunca debería delegar excepciones. Debería capturar todas las que se produjesen y tratarlas apropiadamente.

TEMA 6: CLASES

Este tema es continuación directa del tema 2. Es importante que se recuerden bien todos los conceptos allí expuestos.

La programación orientada a objetos encapsula datos (atributos) y comportamientos (métodos).

Una **clase** es un programa java que agrupa una serie de datos (atributos) y de procedimientos (métodos) que tienen algo en común.

Un **objeto** es un caso particular de una clase.

En Java, la unidad de programación es la clase de la cual es algún momento se instancian (esto es, se crean) objetos. Veamos cual es el proceso de creación y vida de un objeto.

6.1 Ciclo de vida del objeto

Cuando los objetos se quedan sin referencias dejan de ser accesibles por el programa. A partir de ese momento, el sistema puede llamar a sus recursos (la memoria que ocupa). Por ello se habla del ciclo de vida de un objeto, que consta de las siguientes fases:

1 Definición.

Se trata de crear una **referencia** para el objeto (*declaración*).

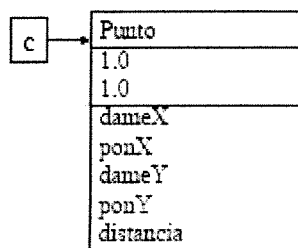
```
Punto c; // inicialmente a null
```



2 Creación.

Se trata de crear el objeto mediante el operador **new** (*ejemplarización*) y llamada a un método constructor (*inicialización*).

```
c = new Punto(1.0, 1.0);
```

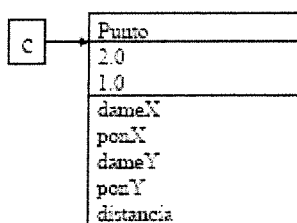


3 Uso.

Se trata de usar el objeto mediante su referencia: llamar a sus métodos, usar sus datos, pasarlo como parámetro, etc.

Por ejemplo, podemos cambiar el valor inicial de un atributo:

```
c.ponX (2.0);
```



Recuerda todo lo dicho acerca de la **referencia** de un objeto.

Cuando una referencia no se refiere a ningún objeto (por ejemplo, antes de crearlo) se dice que apunta a **null**.

Evidentemente, no es posible usar ningún atributo o método de una referencia que vale **null** porque no existe objeto del cual invocarlos. Este punto es importante, el programador tiene que asegurarse de que al usar un objeto, realmente existe (es decir, la referencia no contiene el valor **null**).

Si, por error, se usa un atributo o método de una referencia con valor **null**, el sistema lanzará una excepción **NullPointerException** y el programa acabará.

Afortunadamente, es posible comparar una referencia con el valor **null**, para saber si una referencia tiene asignada una instancia. Así, por ejemplo, podríamos comprobar:

```
P == null
P != null
```

Con estas expresiones determinaremos si es posible o no usar una referencia.

El sistema decide cuándo desaparecen los objetos que se han quedado sin referencias y decide cuándo reclamar los recursos que fueron asignados a la creación del objeto. El programador no tiene control de cuándo se liberarán dichos recursos.

Los tipos no primitivos (objetos) se declaran y se crean por separado pero se admite hacerlo en la misma línea. Los tipos primitivos se crean al declararse.

La definición de un método constructor sigue la forma de la definición de cualquier método, tal y como estudiamos en el tema 4.

Cabe destacar que si al llamar a un constructor, éste lanza una excepción, el objeto no se creará.

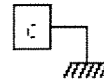
Al llamar al constructor por defecto no incluimos argumentos entre los paréntesis, ponemos sólo ().

A veces es necesario disponer de diferentes modos de construir objetos, por lo cual será necesario disponer de varios constructores.

4 Desaparición.

Consiste en olvidar el objeto cuando no haga falta; cuando no tiene referencias, el intérprete de Java lo destruye. Podemos forzar esa destrucción asignando el valor `null` a la referencia.

```
c = null;
```



Observaciones

- Como ya vimos en el tema 2, los pasos 1 (definición) y 2 (creación) pueden hacerse en uno solo:

```
Punto c = new Punto(1.0, 1.0);
```

6.2 Constructores

Ya se ha visto que para crear un objeto se usa la instrucción `new` seguida de una llamada al método constructor de clase (esto es, el nombre de la clase, y una serie de argumentos entre paréntesis).

El **constructor** es un método especial que tiene el mismo nombre que la clase y que sirve para inicializar los atributos de un determinado objeto cuando éste es creado.

Definir un método constructor

Para definir un constructor, se pone el tipo de acceso, el nombre de la clase, los parámetros que acepta, si lanza excepciones, y un cuerpo o bloque de código en el que realizaremos las inicializaciones de atributos de clase que queramos:

```
acceso nombreClase (parametros) excepciones {...}
```

Tal y como hacemos en el siguiente ejemplo:

```
public Punto (double x, double y) {
    this.x= x;
    this.y= y;
}
```

Constructor por defecto

Si programamos una clase sin constructor el compilador de java le pone uno por defecto. Este **constructor por defecto** no tiene parámetros de entrada y asigna a los atributos los siguientes valores, según su tipo:

```
Enteros → 0
Reales → 0.0
Char → '\0'
Boolean → false
Objetos → null
```

Observaciones

- Al añadir uno o varios constructores a una clase, el constructor que por defecto proporciona Java deja de ser accesible.
- El constructor puede estar sobrecargado, es decir, pueden existir varios constructores dentro de la misma clase con distintos parámetros de entrada.

6.3 Referencias y autoreferencia `this`

Desde fuera de un objeto, podemos usar sus elementos (atributos y métodos) mediante la referencia al objeto:

```
p.x= 3; c.ponX (2.0); z= p.distancia(q); ...
```

Desde dentro de un objeto (en la definición de la clase), podemos usar sus elementos sin referencia, o con `this` como referencia. Esto es útil cuando hay variables automáticas con el mismo nombre.

Ejemplos

```
public void mueve (float _x, float _y) {
    x= _x;
    y= _y;
}

public void mueve (float x, float y) {
    this.x= x;
    this.y= y;
}
```

Además de este uso de `this`, tenemos que destacar otro importante:

Cuando se tienen varios constructores, es muy común que parte de la construcción del objeto sea común a todos ellos. Por ello, se permite que los constructores se invoquen unos a otros, mediante la referencia `this`. Para ello, debemos seguir los siguientes pasos:

- El único requisito es que debe hacerse en la primera línea del constructor.
- Se pone la referencia `this()`, y entre paréntesis los parámetros del constructor que se desean invocar.

Téngase en cuenta que los constructores tendrán diferentes parámetros, por lo que según los parámetros que pongamos en el `this()`, estaremos invocando a uno u otro constructor, de entre los existentes.

6.4 Comparación de objetos

A la hora de comparar dos objetos, existe una importante diferencia:

- Con el operador `==` comparamos referencias y solo devuelve `true` cuando ambas referencias apuntan al mismo objeto.
- Con el método `equals()` lo que se compara es el contenido de dos objetos distintos. El método `equals()` lo tienen todas las clases java ya que está heredado de la clase `Object`. Este método devolverá `true` si los dos objetos son iguales (son del mismo tipo y tienen los mismos datos) y `false` en caso contrario. Para clases que no son estándar (`String`, por ejemplo, es estándar) normalmente redefiniremos el método `equals()` para que la comparación sea correcta.

6.5 Comando final

Se puede calificar como `final`:

- Una variable cuyo valor no queremos que se pueda modificar (es decir, convertimos una variable en una **constante**).
- Un atributo que no dejamos modificar.
- Un método que no queremos que se pueda sobrescribir (durante una extensión).
- Una clase que queramos que no se pueda extender.

Este uso de `this` como acceso a un constructor está restringido al cuerpo de los constructores, y debe aparecer como primera instrucción.

Estudiaremos la clase `Object`, y los conceptos de los que aquí hablamos, en el tema 7.

Extender y sobrescribir son conceptos del tema 7.

6.6 Control de acceso

Definimos **estado de un objeto** como el conjunto de los valores de sus atributos.

Una modificación arbitraria (intencionada o por error) de este estado puede dar lugar a comportamientos indeseados o inconsistencias semánticas en el objeto. Por ello es necesario controlar el acceso a los atributos y métodos de los objetos.

Como ya se dijo, para controlar el acceso en Java, se antepone a la declaración de atributos y métodos el tipo de acceso que se requiere. Los tipos de acceso son:

- `public`

Acceso público → El acceso no está restringido y se puede usar dicho elemento libremente.

- Aplicable a clases, atributos y métodos.
- Lo usaremos sobre todo para: métodos de acceso y modificación, y para constantes.

- `private`

Acceso privado → Los elementos privados sólo pueden ser usados dentro de la clase que los define, lo que impide su invocación exterior.

- Aplicable a atributos y métodos.
- Lo usaremos sobre todo para atributos.

- `protected`

Acceso protegido → Este tipo de acceso permite que los elementos protegidos sólo puedan ser usados dentro de la clase que los define, aquellas clases que la extiendan y cualquier clase en el mismo paquete.

- Sin acceso declarado (cuando no se pone nada)

Acceso de paquete → El acceso a estos componentes es libre dentro del paquete en el que se define la clase.

Accesibilidad atributos y métodos

Se recomienda que todos los **atributos** sean `private` de forma que sólo los métodos del objeto puedan tocarlos, y así el programador pueda responsabilizarse de su contenido en cada momento.

Los métodos, sin embargo, suelen declararse `public` para poder manejar los atributos de la clase y operar con ellos desde fuera de la clase pero sin acceder a ellos directamente.

Métodos get y set

Los métodos `get` y `set` también son llamados **métodos accesoros**. En español, se suelen definir como métodos *dame* y *pon*.

- Los métodos `get` (*dame*) sirven para acceder a (leer) los atributos privados de un objeto.
- Los métodos `set` (*pon*) sirven para modificar los atributos privados de un objeto.

Son métodos totalmente optativos (es decir, no es obligatorio que aparezcan en una clase y además pueden tener el nombre que nosotros queramos).

Es importante resaltar que el acceso a los atributos a través de métodos públicos permite al programador de la clase tener el control del estado del objeto. Así, el programador puede responsabilizarse de que dicho estado es siempre legal y consistente, soportado por dos mecanismos: las excepciones lanzadas por los métodos en caso de argumentos ilegales, y la restricción del acceso a los atributos.

Es la parte de la clase que "se ve" desde fuera de la propia clase.

Es la parte de la clase que "NO se ve" desde fuera de la propia clase.

Entenderemos más estos conceptos en el tema 7.

De momento entenderemos **paquete** como directorio.

El intento de cambiar cualquier atributo privado, desde fuera de la clase, dará lugar a un error de compilación.

En resumen:
Atributos = `private`
Métodos = `public`

6.7 Elementos de clase: **static**

Llamamos **elemento de clase** a cualquier elemento definido en una clase con la palabra **static**.

¿Cuándo definir elementos de clase?

Definiremos elementos de clase en los siguientes casos:

- Definición de constantes (`Math.PI`).
- Variables únicas y comunes para todos los objetos de esa clase.
- Métodos que necesitemos usar aunque no haya objetos (`Math.sqrt()`).
- Métodos que sólo usan elementos de clase.
- El método especial `main()`, pues el intérprete lo debe encontrar antes de construir objeto alguno.

Variables de clase

Los atributos **static**:

- Sólo existe uno para todos los objetos.
- Todos los objetos lo comparten.
- Si un objeto lo modifica, todos lo ven modificado.

Métodos de clase

Un método **static**:

- Se puede llamar referido a la clase, no hace falta crear un objeto para poder llamarlo.
- Sólo puede trabajar directamente sobre atributos **static** pues los atributos normales no están creados.

Cabe destacar que, indirectamente, un método **static** puede recibir objetos como parámetros o crear él mismo algún objeto, si fuera necesario.

6.8 Ámbitos de clase, de método, y de objeto

Ámbito de clase

En el bloque de una clase se pueden encontrar atributos y métodos. Se dice que los identificadores de los atributos, y las firmas de los métodos están en el ámbito de la clase. Este ámbito es visible desde los diferentes bloques internos que aparezcan en la clase, es decir, desde los bloques de código de los métodos.

Por ello, desde dichos métodos se pueden usar los elementos del ámbito de clase, esto es, los atributos de la clase, y los otros métodos de la clase.

Ámbito de método

En un método, un identificador (de variable local o parámetro) puede usarse sólo desde que se ha declarado hasta el final del bloque del método.

Ámbito de objeto

Cuando creamos objetos, cada uno de ellos tiene sus copias locales de datos y métodos, salvo los elementos de clase, únicos y comunes a todos los objetos de una misma clase, como se ha visto.

Se debe limitar al máximo el uso de los elementos de clase.

Un fallo muy común (provoca un error de compilación) es que un método de clase intente usar un atributo que no sea de clase.

IMPORTANTE: Será así aun cuando los métodos se definan antes que los atributos.

En este tema empezaremos a ver la verdadera potencia de la Programación Orientada a Objetos (POO), frente a otros lenguajes tradicionales.

Como ves, el concepto de encapsulación no es nada nuevo a estas alturas de curso.

En el tema anterior, cuando estudiamos el control de acceso (`public`, `private`...) dijimos que cuando no ponemos palabra reservada a tal efecto, el elemento en cuestión tiene "accesibilidad de paquete"

Un ejemplo de paquete es el paquete `java`, que engloba una serie de paquetes con utilidades de soporte al desarrollo y ejecución de una aplicación. Ejemplos de subpaquetes de éste son `util` o `lang`.

Si un paquete contiene clases sin relación, será difícil figurarse qué contiene y para qué sirve.

Existen otros conceptos relativos a encapsulación que no se estudiarán en este curso. Es el caso de las **clases internas**, las **clases locales** y las **clases anónimas**.

TEMA 7: RELACIONES ENTRE CLASES

7.1 Encapsulación

Como ya hemos dicho en muchas ocasiones, la **clase** es la unidad básica de **encapsulación** en Java. Se maneja como un elemento, aunque está compuesta por varios. La encapsulación es la agrupación de una serie de datos (atributos) junto con las funciones que los manejan (métodos).

Además, como también hemos visto, un objeto puede usar la parte pública de otro. Diferenciamos así, en POO, entre clase servidora (la que define un método determinado), y clase cliente (la que invoca a ese método).

7.2 Paquetes

Como ya se mencionó en el tema 6, las clases que contiene un directorio (una carpeta en nuestro ordenador), formarán, por defecto, un paquete. Pero si no lo declaramos como tal, este paquete formado por omisión, carecerá de una serie de posibilidades adicionales de Java, muy útiles. Así, con Java podemos crear un paquete con un determinado nombre, formado por ficheros contenidos SIEMPRE en un mismo directorio (el directorio y el paquete tendrán el mismo nombre), y cuyo contenido será fácilmente accesible y manejable si así lo queremos.

Los paquetes son agrupaciones de clases, interfaces, y otros paquetes (subpaquetes), normalmente relacionados entre sí. Los paquetes proporcionan un mecanismo de encapsulación de mayor nivel que las clases.

Para construir un paquete, se añade a cada fichero que forma parte del mismo una declaración de paquete:

```
package nombrePaquete;
```

Así, todos los elementos en el fichero en el que aparece tal declaración formarán parte del paquete `nombrePaquete`.

Observaciones

- La declaración de paquete debe aparecer una sola vez, por lo que no es posible que una clase o interfaz formen parte de dos paquetes distintos simultáneamente.
- Los paquetes han de construirse de forma que contengan clases e interfaces funcionalmente relacionados, de forma que proporcionen agrupaciones lógicas para los programadores que los usen.

Uso de paquetes

Cuando se requiere usar algún componente de un paquete, se añade una declaración de importación, que puede tener las siguientes formas:

- Importación de un solo elemento (clase o interfaz) → `import nombrePaquete.elemento;`
Para usar lo importado, simplemente se pone el nombre, libremente.
- Importación de todo el paquete → `import nombrePaquete;`
Para usar cualquier elemento del paquete, hay que anteponer a su nombre el nombre del paquete, seguido de un punto.
- Importación de todos los elementos sin uso cualificado → `import nombrePaquete.*;`
Para usar cualquier elemento del paquete, sólo usaremos su nombre, como en el primer caso.

7.3 Composición

Se dice que existe **composición** cuando un objeto incorpora otro objeto (u objetos) como atributo.

Normalmente estos objetos internos son `private` y se inicializan a través del constructor.

La composición es una relación “**tiene-un**”, en la que una clase (**clase contenedora**) “tiene un” objeto (o varios) de otra clase (**clase contenida**) como atributo.

IMPORTANTE: un objeto de la clase contenedora puede usar la parte pública de la contenida.

Ejemplos

Ejemplo básico:

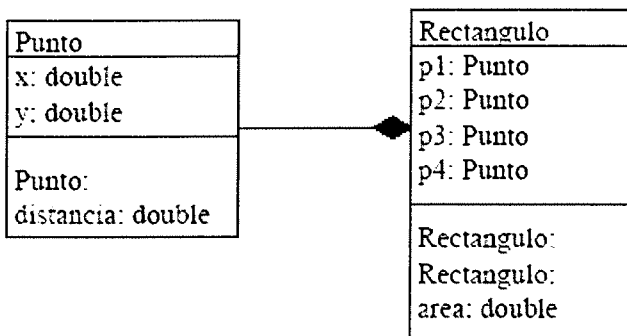
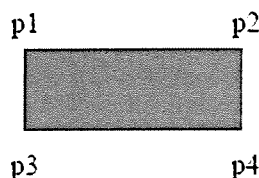
```
class A { ... }
class B { A varA; ... }
```

En este caso se dice que los objetos B **tienen un** objeto A.

Ojo con esto pues es muy importante.

Otro ejemplo:

Un objeto Rectángulo **tiene** cuatro objetos Punto



7.4 Extensión. Herencia

La programación orientada a objetos introduce la capacidad de extender clases, produciendo nuevas definiciones de clases que heredan todo el comportamiento y código de la clase extendida.

- La clase original se denomina clase padre, clase base, o **superclase**.
- La nueva clase definida como una extensión se denomina clase hija, derivada, o **subclase**.

La extensión de una clase se denomina herencia porque la nueva clase hija hereda todos los atributos y métodos de la clase padre.

La herencia es una relación “**es-un**” entre clases. Así, la clase derivada “es una” clase base (enriquecida).

Para extender una clase, la notación es:

```
class B extends A { ... }
```

Dentro de las llaves se escribirán las definiciones necesarias para modificar o añadir comportamientos a la nueva clase B.

La herencia es un mecanismo importante de extensión que permite la reutilización de código existente.

Donde B es la clase derivada (subclase) y A es la clase base (superclase).

Recuerda así mismo lo estudiado en el tema 6 respecto al comando `final`: usado en la cabecera de una clase, impedirá que se pueda extender.

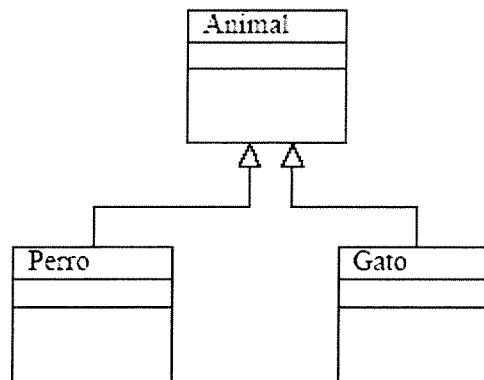
Observaciones

- Usamos herencia cuando necesitamos aumentar la funcionalidad de una clase.
- Una clase solo puede heredar de otra clase (no de varias).
- B hereda de A todos los componentes (todos los atributos y todos los métodos) y en particular, la interfaz (métodos públicos). Tiene que quedar claro que, cuando hacemos una extensión, los métodos de la clase “padre” también los tiene la clase “hija” aunque no aparezcan explícitamente en el código de la clase “hija”.
- B puede redefinir métodos de A (también se denomina sobrescribir o sombrear métodos). En este caso se dice que los métodos de A quedan ocultos. Aún así se puede acceder a los métodos ocultos:

```
super.metodoOculto ()
```
- La extensión jamás provocará la modificación de la clase base.

Ejemplo

Las clases Perro y Gato heredan de la clase Animal. Por tanto la clase Perro es una clase Animal.



Compatibilidad de tipos

Como ya se ha visto, la herencia tiene la siguiente propiedad: es una relación “es-un”. En nuestro ejemplo, decíamos que los objetos de la clase Perro pueden ser considerados como Animales. Esto quiere decir que una referencia de la clase Animal puede contener una instancia de la clase Animal o de cualquier clase derivada, puesto que siguen siendo Animal. En particular, una referencia de la clase Animal puede contener una instancia de la clase Perro ó de la clase Gato.

En definitiva, la clase base es compatible con el tipo derivado, pero no al revés. Un Perro es un Animal, pero no cualquier Animal será un Perro (puede ser también un Gato).

Así, decimos que la extensión produce compatibilidad ascendente, esto es, una referencia de la clase base puede contener un objeto de una clase derivada.

Conversión ascendente: Upcasting

Los objetos pueden ser asignados a referencias de diferentes clases, con la intención de tener distintas visiones del mismo. Un objeto se puede asignar a una referencia de su clase y el objeto se ve como de su clase, o bien se puede asignar a una referencia de una clase ascendente. Esto se denomina conversión ascendente (**upcasting**) y siempre es posible.

Así, supongamos:

```
class B extends A { ... }
```

Importante recordar:
Una variable puede referenciar objetos de su propia clase, o de cualquier clase extendida a partir de ella.

- Las variables de clase B sólo pueden referirse a objetos de clase B.
- Las variables de clase A pueden referenciar objetos de clase A (lo normal) y también pueden referenciar objetos de clase B (upcasting).
- El upcasting es 100% seguro y siempre se puede aplicar: `varA = varB;` Esto es debido a que un objeto de clase B ha heredado todos los atributos y métodos de la clase A.

Conversión descendente: Downcasting

También es posible asignar un objeto a una clase derivada, y el objeto se ve como si fuera de la clase derivada. Esto se denomina conversión descendente (**downcasting**) y no siempre es legal (la clase derivada podría no ser compatible con el objeto).

Por esta razón, este tipo de conversión ha de ser siempre explícito, anteponiendo a la referencia el nombre de la clase (entre paréntesis):

```
refSubClase = (identificadorSubClase) refSuperClase;
```

Así, supongamos: `class B extends A { ... }`

- Sólo se puede utilizar si estamos invirtiendo un upcasting: `varB = (B) varA;`
- Se lanzará una excepción (error) si el objeto referenciado no es realmente de clase B.
- Se puede chequear previamente con el comando `instanceof`:

```
if (varA instanceof B) varB = (B) varA;
```

Reescritura

Es posible modificar la parte que se hereda en la clase derivada. A esto se le llama **reescribir** los elementos de la clase base.

La reescritura es la capacidad que tiene una clase derivada para redefinir un elemento (no estático) de la clase base.

Para ello, la clase derivada puede definir:

- Un **atributo** con el mismo nombre que uno de la clase base.
- Un **método** con la misma signatura que uno de la clase base.

Observaciones

- Los elementos de clase (static) no se pueden reescribir, puesto que no van a formar parte de las instancias de la clase, y quedarían inaccesibles a los objetos de las clases derivadas.
- En la redefinición se puede ampliar el nivel de acceso, haciéndolo más público, pero no más privado.
- Como ya se vio, al reescribir métodos en una clase derivada, el de la clase base queda oculto, pero podemos acceder al "original" con la sentencia:

```
super.metodoOculto ()
```

- Aunque la herencia establece una jerarquía que puede ser de muchos niveles, una clase sólo dispone de sus elementos y de los elementos de la clase base. Así, una clase derivada no tiene acceso a los elementos de las clases base de su clase padre directa. Por ello, un intento de acceso `super.super.elemento` **no es posible**.

A una referencia de una clase derivada se le asigna un objeto contenido en una referencia a una clase base.

La excepción será tipo `ClassCastException`.

Ahora bien, el uso intensivo de `instanceof` denota un diseño incorrecto y no es aconsejable.

El uso de reescritura es muy útil cuando la clase derivada amplía la funcionalidad de una clase, así como cuando la clase derivada particulariza el comportamiento de la clase base.

En cualquier caso, recuerda de nuevo lo estudiado respecto a **final**: usado con métodos o atributos, impide que se puedan sobrescribir. Por estas y otras razones, el compilador puede optimizar el código de elementos final, ejecutándose más rápido.

Esto es, en java no existe el concepto de "abuelo".

Clase Object

Todas las clases de Java son extensiones de la clase Object (clase predefinida).

```
Object x= new CualquierClase (...);
```

Esta clase proporciona métodos muy útiles, que podremos aplicar a un objeto de cualquier clase. Entre estos, destacamos dos:

```
- public String toString(){...}
```

Este método devuelve el contenido del objeto en forma de cadena (String), permitiendo así la escritura directa por pantalla de un objeto utilizando el método `System.out.println()`, metiendo entre los paréntesis la referencia del objeto, como si de una cadena se tratase.

Para un correcto funcionamiento de este método en nuestra clase, ésta puede particularizarlo. Reescribiéndolo conseguiremos que la cadena devuelta por el mismo esté en el formato que queramos.

```
- public boolean equals(Object obj) {...}
```

Este método compara el contenido de dos objetos distintos. Como ya comentamos en el tema 6, devolverá `true` si los dos objetos son iguales (son del mismo tipo y tienen los mismos datos) y `false` en caso contrario.

Como también dijimos, para clases que no son estándar deberemos redefinir el método `equals()` para que la comparación sea correcta.

Constructores

Crear objetos de una clase derivada puede tener cierta dificultad si no se conoce bien cuál es el proceso de construcción de instancias, que empezamos a ver en los temas 2 y 6, y que completamos en este.

Al crear un objeto de una clase derivada, primero se crea su parte base y después se construye la parte de la clase derivada. Si la base es, a su vez, derivada de otra, se aplica el mismo orden, hasta llegar al constructor de Object.

De esta forma, cuando se instancia la clase Perro, primero se construye un Object, luego se añade la construcción de la parte de la clase Animal y, por último, la parte de la clase Perro.

El proceso se complica un poco cuando se añaden constructores a las clases. Cuando una clase tiene un constructor (distinto del asignado por defecto), primero se construye la clase base, luego la clase derivada y entonces se ejecuta el código del constructor.

También existe la posibilidad de que la clase base tenga uno o varios constructores definidos. Cuando se construye la clase base, Java tiende a utilizar el constructor por defecto (un constructor sin parámetros). Con esto, si hemos añadido constructores a esta clase base (y por tanto su constructor por defecto ha desaparecido), y todos tienen parámetros, la clase derivada tendrá que invocar explícitamente a uno de ellos. Para ello, añadiremos en la clase derivada un constructor en el que la primera sentencia sea:

```
super(parametros);
```

Observaciones

- Un constructor puede invocar, en la primera sentencia de su cuerpo, a `this()` ó a `super()`, pero sólo a uno de ellos.

Si no se realiza esta invocación mediante `super()` cuando se debe, habrá un error de compilación: No constructor matching `A()` found in class `A`, que quiere decir que no hay un constructor sin parámetros en la clase base.

Recuerda el uso de `this()` para invocar a otros constructores de la misma clase.

Resumen de las fases de construcción de un objeto

1. Se asigna el valor por defecto a los atributos (0, 0.0, false, \u0000, null).
2. Se invoca al constructor de la clase base:
 - Explícitamente invocado como primera sentencia `super(parametros).`
 - O no se pone nada, con lo que se invoca al constructor sin parámetros de la clase base.
 - Excepto que se llame a `this()`.
 - La cadena de constructores sigue hasta la clase `Object`.
3. Se asignan los valores iniciales según las sentencias de inicialización.
4. Se ejecuta el cuerpo del constructor (si tiene sentencias a parte de la inicialización de atributos).

7.5 Polimorfismo

A parte de explicar el concepto de polimorfismo, repasamos un poco.

Se ha explicado que en una jerarquía de herencia existe compatibilidad ascendente. Por ello, a una referencia se le puede asignar un objeto de la clase declarada o de cualquier clase derivada de ella.

Por ejemplo, podríamos tener(suponiendo que no hemos añadido constructores):

```
Animal a = new Gato();
```

También se ha dicho que es posible invocar a un método del objeto y que se ejecutará el método correspondiente a la clase a la que pertenece el objeto. Así, suponiendo que existe el método `dice()` en la clase `Animal`, y que se ha reescrito en la clase `Gato`, se puede escribir:

```
Animal a = new Gato();
Gato g = new Gato();

a.dice();
g.dice();
```

Ambas invocaciones del método `dice()` ejecutarán el código de la clase `Gato`, pues las referencias `a` y `g` contienen instancias de la clase `Gato`, aunque `a` esté declarado como `Animal`.

Ahora bien, el compilador comprueba que el método `dice()` está disponible en la clase de la que se declara la referencia. Así, el siguiente código daría un error de compilación:

```
Object obj = new Gato();

obj.dice();
```

La referencia `obj`, declarada como de la clase `Object`, puede contener un `Gato`, pues es una clase derivada de `Object` (como todas). Sin embargo el compilador no permite invocar `dice()` con la referencia `obj`, puesto que en la definición de `Object` no existe tal método. Hay que tener en cuenta que en este caso se ve claramente que la referencia `Object` tiene una instancia de `Gato`, pero no siempre se puede determinar.

La conclusión es que:

- Los métodos disponibles a la hora de compilar vienen determinados por las declaraciones del programa.
- Los métodos a invocar durante la ejecución vienen determinados por la clase real a la que pertenece el objeto.

En esto consiste el polimorfismo: un objeto puede ser asignado a una referencia declarada de la misma clase o de cualquiera de las clases base, es decir, puede ser visto de muchas formas distintas. Pero su comportamiento viene determinado por la clase que se instancia (la generada por la sentencia `new`), que es independiente de la referencia usada.

Esto es, un `Gato` puede ser visto como `Gato` o como `Animal`, pero se comportará como un `Gato`.

7.6 Herencia forzada: Clases abstractas

La herencia puede forzarse a través de las clases abstractas.

Si se intenta crear un objeto de una clase abstracta, se producirá un error de compilación.

Siguiendo con nuestro ejemplo, puede no tener sentido que existan objetos `Animal` si se desea que existan perros y gatos concretos. El mecanismo para forzar esto es declarando abstracta la clase `Animal`, puesto que una clase abstracta no se puede instanciar, esto es, no se pueden crear objetos de clases abstractas.

Para declarar una clase abstracta usamos la siguiente notación en la cabecera de la clase:

```
abstract class
```

Cabe destacar que, no obstante, sigue siendo posible tener la visión de instancias de clases derivadas como referencias a su clase base abstracta, como denotamos en el siguiente ejemplo:

```
Animal a = new Gato();
```

Aquí, el objeto `a` tendrá una referencia de `Animal`, pero se crea como un `Gato`, por eso funciona.

Aquí, el compilador daría error si se intentara instanciar el objeto como `new Animal()`;

Observaciones

- Las clases abstractas están a medio camino entre los interfaces y las clases “normales”.
- Una clase abstracta puede tener uno o más métodos sin detallar (es decir sólo con la cabecera, sin el bloque ni las llaves). A estos métodos sin detallar se les llama **métodos abstractos**, se les declara en su cabecera como `abstract` y se supone que serán sobrescritos por las clases que hereden de la clase abstracta..
- Si una clase derivada no redefine todos los métodos abstractos, se considera abstracta y debe declararse como tal; en caso contrario, el compilador lanzará un error.

Así, si se desea que un método continúe siendo abstracto en una clase derivada, se puede hacer explícitamente declarando dicha clase derivada como abstracta y reescribiendo el método sin cuerpo, de forma que sigue siendo abstracto. Si no, se debe proporcionar una implementación a dicho método.

Con esto se consigue tener relaciones parciales de objetos cuya codificación no existe. Lo que se está obligando es que todas las clases derivadas tengan una parte común en su interfaz.

7.7 Interfaces

Como hemos visto, el polimorfismo permite manejar ejemplares de distintas clases de forma homogénea. Para ello, hemos estudiado como estos objetos deben ser ejemplares de clases relacionadas mediante herencia. No obstante, es muy común que surja la necesidad de tener objetos que comparten una interfaz pública pero que no pertenecen a la misma jerarquía de herencia (esto es, cuando no existe una relación “es – un” entre tales objetos).

En esta tónica, definimos interfaz como: declaración de métodos no estáticos y campos estáticos finales cuyo objetivo es que sean implementados por varias clases de forma que sus instancias sean polimórficas y presenten la misma interfaz pública.

Definición de una interfaz

- Los elementos de una interfaz son públicos por definición: no es necesario poner la palabra `public`, ni se pueden cambiar sus derechos de acceso.
En la declaración de los métodos se admite calificarlos como públicos, aunque es redundante.
- La declaración del método puede incluir el lanzamiento de excepciones.

La interfaz es el máximo nivel de abstracción en Java.

IMPORTANTE: reiteramos, las interfaces son un mecanismo de Java para obtener el polimorfismo entre clases no pertenecientes a la misma jerarquía de herencia.

Muchos textos presentan las interfaces como un caso particular de clases abstractas (se llega a decir que son las clases abstractas más puras), pero semánticamente son diferentes.

- Los métodos de una interfaz no pueden ser estáticos, porque los elementos estáticos son de clase y una interfaz no es una clase.
- Una interfaz indica de qué métodos disponemos pero sin implementarlos. Es decir solo se escribe la cabecera pero no el bloque (de hecho, ni siquiera se ponen las llaves de comienzo y final).
- No se pueden instanciar objetos de un interfaz. Por ello, los interfaces nunca tienen constructores.

Sin embargo una referencia sí puede apuntar a una interfaz.

Ejemplos

```
interface Coordenada {
    double x ();
    double y ();
    double distancia (Coordenada q);
}

interface Poligono {
    int numero_vertices ();
    Coordenada vertice (int i);
    double perimetro ();
    double superficie ();
    void escala (double proporcion);
}
```

Implementación de una interfaz

Una vez que se ha definido una interfaz, cualquier clase puede presentar dicha interfaz mediante un mecanismo denominado implementación de interfaces. Así, sobre una implementación debemos saber que:

- Es aquella clase “normal” que hace todo lo que prometía la interfaz, Habitualmente, incluye constructores.
- La sintaxis es: `nombreClase implements nombreInterface {...}`
- Cuando una clase implementa una interfaz, tendrá que sobrecargar sus métodos con acceso público. Con otro tipo de acceso, el compilador lanzará un error.
- Los atributos son finales, sin necesidad del calificativo final, y deben cargarse con un valor inicial.
- Se pueden añadir atributos y métodos que no estaban en la interfaz.

```
class Cartesiana implements Coordenada {
    private double x, y;
    Cartesiana (double __x, double __y) {
        x= __x; y= __y;
    }
    public double x () { return x; }
    public double distancia (Coordenada q) {
        double dx= q.x()-x; double dy= q.y()-y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

- Una misma clase puede implementar varias interfaces. Para ello empleamos la notación:

```
class ClaseA implements InterfaceB, InterfaceC {...}
```

Ejemplo

```
interface Mamifero { void amamanta (); }
interface Oviparo { void ponHuevos (); }

class Ornitorrinco implements Mamifero, Oviparo { ... }

Ornitorrinco juliana= new Ornitorrinco();
juliana.ponHuevos();
juliana.amamanta();
```

De esta forma quedará patente que se está definiendo una implementación parcial.

- Es importante notar que si alguno de los métodos quedase sin implementar, la clase será abstracta. En ese caso, es necesario definirlo de forma explícita, añadiendo el calificativo **abstract** a la clase y añadiendo la cabecera del método que se queda sin implementar anteponiéndole a su vez también el calificativo **abstract**.

Evidentemente, es posible que una clase proporcione cuerpo a todos los métodos de las interfaces que implementa y aun así sea abstracta añadiendo otros métodos abstractos.

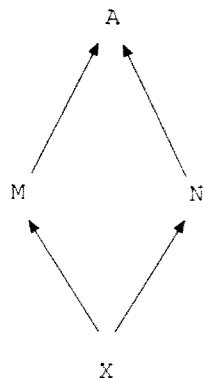
Jerarquía de interfaces

Se pueden construir nuevas interfaces como extensiones de otras interfaces ya existentes, obteniéndose una jerarquía de interfaces. Es lo que se conoce como **Herencia Múltiple**.

A diferencia de la jerarquía de clases, que es única en Java (todas las clases derivan de la clase **Object**), las interfaces no tienen una interfaz común predefinida.

Ejemplo

Se puede dar el caso de que varios ancestros tengan a su vez un ancestro común, representado por un diagrama de diamante:



```
interface A {...}
interface M extends A {...}
interface N extends A {...}
interface X extends M,N {...}
```

Observaciones

- En algunos textos a esta jerarquía de interfaces se la denomina **Implementación Múltiple**, entendiendo como herencia múltiple al intento de realizar diagramas como este usando únicamente herencia (caso prohibido en Java).

Aunque en Herencia no vimos un punto análogo a este, durante toda la explicación desarrollamos el concepto de jerarquía de herencia:

- La relación de herencia es transitiva y define una jerarquía (de herencia).
- Cualquier clase en java puede servir como clase base para ser extendida.
- En Java, todas las clases están relacionadas en una única jerarquía de herencia, puesto que toda clase o bien hereda explícitamente de otra o bien hereda implícitamente de la clase **Object** predefinida.

TEMA 8: COLECCIONES DE DATOS

Normalmente los programas necesitan manejar grandes cantidades de datos guardados en muchas variables. En este tema vamos a tratar las estructuras que van a permitir a un programa mantener la información de tantos datos como desee, y cómo se manejan estas estructuras de datos.

8.1 Arrays

A las estructuras estáticas de datos las llamaremos a partir de ahora array (arreglo) ó matriz. Un array es un conjunto homogéneo indexado de elementos:

- Homogéneo → todos los elementos deben ser del mismo tipo.
- Indexado → cada elemento ocupa una posición conocida del array.

Declaración y creación de arrays

La **declaración** de cualquier array se hace de las siguientes formas:

```
tipoBase[] nombreArray;
tipoBase nombreArray[];
```

Ambas declaraciones son equivalentes, y declaran un array de elementos de tipo `tipoBase` dándole el nombre `nombreArray`. Es decir, `nombreArray` es una referencia a una estructura que puede guardar muchos valores de tipo `tipoBase`.

Ejemplos

```
int[] pixel;
char[] nombre;
float notas[];
```

Observaciones

- No existe ninguna restricción al tipo base de un array.
- Al declarar un array no se dice el número de elementos que va a almacenar.
- En Java, los arrays son objetos, por lo que para poder utilizarlos necesitan ser creados antes.
- Como cualquier otro objeto, el valor inicial por defecto es null.

Al igual que cualquier otro objeto, los arrays se crean con `new`. En la **creación** de arrays, se fija el número de elementos almacenados (tamaño del array).

```
pixel = new int[3];
nombre = new char[30];
notas = new float[24];
```

Es habitual hacer la declaración y la creación en la misma línea:

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

Observaciones

- En todos los casos anteriores, los arrays empiezan vacíos (cada elemento del array se inicializa al valor por defecto de su tipo).
- También se puede crear el array lleno desde el principio:

```
boolean []tabla = {true, false, true, false};
int b[] = {3,4,5};
```

En muchos programas existe la necesidad básica de disponer de muchas variables u objetos del mismo tipo o de la misma clase. En este caso, es necesario poder llamar a todo el grupo de variables o de objetos con un único nombre para todos ellos. Además, debe ser posible manejar cada uno de ellos de forma individual como se ha descrito en temas anteriores.

Se denomina **tipo base** de un array al tipo de elementos que se pueden guardar en cada una de las posiciones del array.

Da lo mismo donde se pongan los corchetes.

De esta forma se indica, por ejemplo, que el array `pixel` puede contener 3 números enteros.

En este caso no hace falta usar el comando `new`.

Hasta ahora sólo habíamos visto ejemplos con arrays de una dimensión.

Arrays multidimensionales

Se puede disponer de arrays de más de una dimensión.

Un array declarado de la siguiente manera sería una matriz de dos dimensiones:

```
tipoBase[][] nombreArray;
```

Donde se indica que para elegir uno de los elementos del array hay que usar dos índices, esto es, hay que elegir el elemento por su fila y su columna.

Para crear un array de dos dimensiones hay que crear el objeto array de forma similar a como indicamos en la página anterior, solo que en este caso hay que indicar los tamaños de las dos dimensiones.

Ejemplos

```
int[][] tabla = new int[4][7];
int b[][] = {{1,2}, {2,3,4}};
boolean[][] quiniela = new boolean[3][15];
```

Observaciones

- Una matriz bidimensional no tiene por qué ser cuadrada, es decir, cada fila puede tener longitud distinta!!!
- Por supuesto, pueden existir matrices tridimensionales...etc.

Acceso a los elementos

Los elementos se indexan en el array, en el rango [0 ... N-1], donde N es el tamaño del array.

Así, para referirse a cada una de las variables u objetos de un array ya creado, se utiliza el número, denominado **índice**, que indica la posición del elemento que se quiere

Cualquier intento de hacer referencia a un elemento cuyo índice no esté en el rango del array generará la **excepción** `ArrayIndexOutOfBoundsException`.

Ejemplos

```
char vocales [] = {'a', 'e', 'i', 'o', 'u'};
vocales[0] == 'a';
i=2;
vocales[i+1] == 'o';

int b [] = new int [5];
b[0] = 0;
b[1] = Integer.valueOf(args[0]).intValue();
b[2] = b[0]*b[1];
b[3]++; //valdrá 1
```

En el primero de estos ejemplos se crea una tabla de 4 por 7 elementos, en el segundo, inicializamos cada una de las filas desde el principio.

MUY IMPORTANTE

OJO! En Java el índice de un array siempre empieza en 0!

Longitud de un array: length

El atributo `length` de un array se puede utilizar para consultar su valor pero no se puede modificar.

MUY IMPORTANTE

Cuando tenemos un array y queremos saber su longitud utilizamos el comando `length`:

```
nombreArray.length
```

Este comando devuelve un entero que es la longitud total del array independientemente de que las posiciones estén vacías o no.

No se debe confundir el comando `length`, que devuelve la longitud de un array, con el método `length()` que devuelve la longitud de una cadena (`String`).

Ejemplo

```
class Longitud {
    public static void main (String[]args) {
        String[] c = {television, mesa, silla};
        System.out.println("longitud del array: " + c.length);
        System.out.println("longitud del c[0]: " + c[0].length());
        System.out.println("longitud del c[1]: " + c[1].length());
        System.out.println("longitud del c[2]: " + c[2].length());
    }
}
```

Si ejecutamos la anterior clase veremos por pantalla lo siguiente:

```
longitud del array: 3
longitud del c[0]: 10
longitud del c[1]: 4
longitud del c[2]: 5
```

Observaciones

Gran parte de los errores con arrays están relacionados con sus límites.

- Según lo estudiado en la página anterior, la primera posición del array es 0 y la última es `length - 1`, siendo `length` el atributo del array que guarda su tamaño.

Hay que tener siempre en cuenta que un array bidimensional no tiene por qué ser cuadrado, y por tanto cada fila puede tener una longitud distinta.

En el caso de **arrays bidimensionales** el funcionamiento de `length` es algo distinto.

```
nombreArrayBidimensional.length
```

devuelve el número de filas del array, mientras que:

```
nombreArrayBidimensional[i].length
```

devuelve el número de columnas de la fila `i`.

Ejemplo

```
class Matriz {
    public static void main (String[]args) {
        int[][] m = {{1,2,3,4}, {4,5}, {6,7,8,9,0}};
        System.out.println("número de filas: " + m.length);
        System.out.println("longitud de la fila 1: " + m[0].length);
        System.out.println("longitud de la fila 2: " + m[1].length);
        System.out.println("longitud de la fila 3: " + m[2].length);
    }
}
```

Si ejecutamos la anterior clase veremos por pantalla lo siguiente:

```
número de filas: 3
longitud de la fila 1: 4
longitud de la fila 2: 2
longitud de la fila 3: 5
```


Otro ejemplo

```
int b[][]= {{1,2}, {3,4,5}};

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

Por pantalla veremos lo siguiente:

```
b[0][0]= 1
b[0][1]= 2
b[1][0]= 3
b[1][1]= 4
b[1][2]= 5
```

Un array especial: String[] args

Cuando ejecutamos una clase desde la línea de comandos lo que escribimos tras el nombre de la clase es entendido como un array de cadenas (String), que es el parámetro de entrada del método main. A este array se le suele llamar args pero este nombre no es obligatorio.

Por ejemplo, si escribimos la siguiente clase:

```
class Escribe {
    public static void main (String[]args){
        System.out.print(args[0] + " " + args[1] + " " +
            args[2]);
    }
}
```

y después ponemos lo siguiente en la línea de comandos:

```
>java Escribe hola casa liso6
               args[0]  args[1]  args[2]
```

veremos que en la pantalla aparece:

```
hola casa liso6
```

Sin embargo, si no sabemos de antemano cuántos argumentos vamos a poner en la línea de comandos, lo mejor es hacer un bucle:

```
class Escribe {
    public static void main (String[]args){
        for(int i = 0, i<args.length, i++)
            System.out.print(args[i] + " ");
    }
}
```

Java entiende todo como cadenas, ya sean letras ó números.

string [] args

- Al intentar ejecutar, cada cosa que yo escriba después de `java nombreClase` se almacena en un array de cadenas, llamado `args`.
- El separador entre Strings (cadenas) será el espacio.

Ejemplo:

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

lo que el ordenador "piensa" es:

HOLA CHAVALES QUE TAL
 args[0] args[1] args[2] args[3]

args.length → 4

Por Pantalla:

```
>java Ejecutable HOLA CHAVALES QUE TAL
```

- Para moverme por las diferentes palabras introducidas:
 - Me estaré moviendo por las diferentes posiciones de un array, llamado `args`.
 - Accederé a las posiciones usando `args[posicion]`.
 - Veré su tamaño poniendo `args.length`.
- Para moverme por los caracteres de una determinada palabra introducida (en la posición `i`):
 - Me estaré moviendo por el String contenido en `args[i]`.
 - Accederé a los diferentes caracteres usando `args[i].charAt(posicion)`.
 - Veré su tamaño poniendo `args[i].length()`.

IMPORTANTE: si no nos dan otros requisitos:

Cuando queramos usar este tipo de entrada, nos aseguraremos de que si `args.length==0` ó `args[i].length()==0` ó `args[i]==null` salte excepción.

Cuando queramos usar colecciones como las que vamos a estudiar ahora, debemos importar el paquete `util` de Java. Para ello pondremos `import java.util.*;` al principio de la clase con la que trabajemos.

Recuerda lo estudiado en el tema 7 acerca de las interfaces: se trata de **declaración** de métodos y atributos cuyo objetivo es que sean **implementados** por varias clases. Esto es, **no podemos crear objetos de una interfaz**, sino que debemos tener clases que la implementen.

En seguida veremos que, efectivamente, no usaremos el interfaz `List` como tal, sino clases que la implementan (también predefinidas en Java).

Esta interfaz, así como las clases que estudiaremos (que la implementan) **implementan a su vez la interfaz** `Collection<E>`, `Iterable<E>`, entre otras.

En estos apuntes no explicamos TODOS los métodos de cada clase o interfaz. Al final de los apuntes puedes encontrar todas las APIs necesarias para consultar documentación completa.

Todas las clases que vemos, además, reescriben el método `clone()`, de `Object`.

8.2 Colecciones de tamaño dinámico

Vistas ya las los arrays, colecciones de datos cuyo tamaño no puede cambiar una vez creados, introducimos ahora el estudio de otras colecciones, cuyo tamaño puede variar según las utilizamos, y con las que podemos usar funciones prediseñadas que nos facilitan el trabajo (funciones para introducir elementos, borrarlos, obtener un elemento determinado..etc).

8.2.1 Interface List<E>

Con ella crearemos **listas de objetos de tipo E**. Es importante saber que usando esta interfaz:

- Se respeta el orden en el que se insertan los objetos.
- Admite duplicados.
- El tamaño se adapta dinámicamente a lo que sea necesario.

Se puede decir que es “similar” a un array unidimensional de tamaño dinámico.

Algunos de los métodos que incluye la interfaz, por lo tanto comunes a todas las clases que lo implementan, son:

- | | |
|---|---|
| • <code>boolean add(E elemento)</code> | → Añade el elemento al final de la lista. |
| • <code>void add(int posicion, E elemento)</code> | → Añade el elemento en la posición indicada. |
| • <code>void clear()</code> | → Elimina todos los elementos de la lista. |
| • <code>boolean contains(E elemento)</code> | → Devuelve <code>true</code> si la lista contiene el elemento. |
| • <code>boolean equals(Object x)</code> | → Compara el objeto indicado con la lista. |
| • <code>E get(int posicion)</code> | → Devuelve el elemento que está en la posición indicada. |
| • <code>int indexOf(E elemento)</code> | → Devuelve la posición del primer elemento que coincida con el especificado. Devuelve <code>-1</code> si el elemento no está. |
| • <code>boolean isEmpty()</code> | → Devuelve <code>true</code> si no hay elementos en la lista. |
| • <code>Iterator<E> iterator()</code> | → Devuelve un iterator con los elementos de la lista. |
| • <code>E remove(int posicion)</code> | → Devuelve y elimina el elemento de la posición especificada. |
| • <code>boolean remove(E elemento)</code> | → Elimina el primer objeto que coincida con el especificado. |
| • <code>E set(int posicion, E elemento)</code> | → Reemplaza el elemento de la posición indicada por este. |
| • <code>int size()</code> | → Devuelve el número de elementos de la lista. |
| • <code>Object[] toArray()</code> | → Devuelve un array unidimensional con todos los elementos de la lista. |

Clases que implementan la interfaz List

Las clases que implementan listas, y por lo tanto aquellas a las que llamaremos para crear colecciones de este tipo, son:

- ArrayList<E>

Se trata de un array dinámico, esto es, igual que un array unidimensional pero su tamaño se adapta dinámicamente al número de elementos que contiene.

Es una colección con gran economía en memoria.

Aparte de los métodos de la interfaz, esta clase tiene algunos importantes:

- | | |
|---|--|
| <code>ArrayList()</code> | → Constructor que crea una lista vacía con capacidad inicial 10 |
| <code>ArrayList(Collection<? extends E> c)</code> | → Constructor que crea una lista que contiene los elementos de la colección especificada. |
| <code>ArrayList(int tamaño)</code> | → Constructor que crea una lista del tamaño inicial indicado. |
| <code>void ensureCapacity(int minCapacidad)</code> | → Incrementa el tamaño de la lista, si es necesario, hasta el mínimo que le pasamos como parámetro. |
| <code>void removeRange(int from, int to)</code> | → Elimina todos los elementos de la lista cuyo índice esté entre <code>from</code> y <code>to</code> . |

Una lista encadenada se compone de nodos internamente ligados con el nodo siguiente y el anterior. Se presta especial atención al nodo primero y al último, y se respeta el orden de inserción, ya sea al principio o al final de la lista.

- LinkedList<E>

Es una lista encadenada.

Las inserciones y eliminaciones internas son rápidas.

Aparte de los métodos de la interfaz, esta clase tiene algunos importantes:

LinkedList()	→ Constructor que crea una lista vacía.
LinkedList(Collection<? extends E> c)	→ Constructor que crea una lista que contiene los elementos de la colección especificada.
void addFirst(E elemento)	→ Inserta el elemento especificado al principio de la lista.
void addLast(E elemento)	→ Inserta el elemento especificado al final de la lista.
E getFirst()	→ Devuelve el primer elemento de la lista.
E getLast()	→ Devuelve el último elemento de la lista.
E removeFirst()	→ Devuelve y elimina el primer elemento de la lista.
E removeLast()	→ Devuelve y elimina el último elemento de la lista.

Ejemplo

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

lista.add(1);
lista.add(9);
lista.add(1, 5);

System.out.println(lista.size());      // 3
System.out.println(lista.get(0));      // 1
System.out.println(lista.get(1));      // 5
System.out.println(lista.get(2));      // 9

for (int n: lista)
    System.out.print(n); // 1 5 9
```

Recorrido de listas

Debemos conocer tres formas distintas de recorrer una lista. Las vemos con un ejemplo:

```
List<X> lista= ...;

1. for (int i= 0; i < lista.size(); i++) {
    X dato= lista.get(i);
    procesa (dato);
}

2. for (X dato : lista)
    procesa (dato);

3. Iterator<X> it= lista.iterator();// versiones anteriores
   while ( it.hasNext() ) {
       X dato= it.next();
       procesa (dato);
   }
```

Lo más general.

Lo preferible, recordar lo visto respecto al nuevo bucle for.

Propio de versiones anteriores.

Esta interfaz, así como las clases que estudiaremos (que la implementan) implementan a su vez la interfaz `Collection<E>`, `e` `Iterable<E>`, entre otras.

Existen importantes conceptos acerca del sincronismo de estas colecciones, pero se escapan del temario de este curso. Para más información al respecto, así como completa referencia de los métodos de la interfaz y de las clases que la implementan, puedes consultar las APIs adjuntas al final de estos apuntes.

Todas las clases que vemos, además, reescriben el método `clone()`, de `Object`.

En clase veremos una breve explicación de cómo funciona el criterio de **ordenación natural**.

`SortedSet` es otra interfaz que implementan estas clases que estamos estudiando.

8.2.2 Interface Set<E>

Con ella crearemos **conjunto de objetos de tipo E**. Es importante saber que usando esta interfaz:

- No se respeta el orden de inserción.
- No admite duplicados.
- El tamaño se adapta dinámicamente a lo que sea necesario.

Algunos de los métodos que incluye la interfaz, por lo tanto comunes a todas las clases que lo implementan, son:

- | | |
|---|--|
| • <code>boolean add(E elemento)</code> | → Añade el elemento al conjunto, sólo si no estaba ya. |
| • <code>void clear()</code> | → Elimina todos los elementos del conjunto. |
| • <code>boolean contains(E elemento)</code> | → Devuelve true si el conjunto contiene el elemento. |
| • <code>boolean equals(Object x)</code> | → Compara el objeto indicado con el conjunto. |
| • <code>boolean isEmpty()</code> | → Devuelve true si no hay elementos en el conjunto. |
| • <code>Iterator<E> iterator()</code> | → Devuelve un iterator con los elementos del conjunto. |
| • <code>boolean remove(E elemento)</code> | → Elimina el objeto especificado si está contenido en el conjunto. |
| • <code>int size()</code> | → Devuelve el número de elementos del conjunto. |
| • <code>Object[] toArray()</code> | → Devuelve un array unidimensional con todos los elementos del conjunto. |

Clases que implementan la interfaz Set

Las clases que implementan conjuntos, y por lo tanto aquellas a las que llamaremos para crear colecciones de este tipo, son:

- HashSet<E>

Es una colección con gran economía en tiempo y memoria.

No ordena los elementos (**no emplea ordenación natural**).

Aparte de los métodos de la interfaz, esta clase tiene algunos importantes:

- | | |
|---|---|
| <code>HashSet()</code> | → Constructor que crea un conjunto vacío, de tamaño inicial 16 |
| <code>HashSet(Collection<? extends E> c)</code> | → Constructor que crea un conjunto que contiene los elementos de la colección especificada. |
| <code>HashSet(int tamaño)</code> | → Constructor que crea un conjunto del tamaño inicial indicado. |

- TreeSet<E>

Es una colección más lenta y voluminosa.

Coloca los elementos por **orden natural** (cuando se recorre con el iterator, los elementos salen ordenados).

Aparte de los métodos de la interfaz, esta clase tiene algunos importantes:

- | | |
|--|---|
| <code>TreeSet()</code> | → Constructor que crea un conjunto ordenado vacío. |
| <code>TreeSet(Collection<? extends E> c)</code> | → Constructor que crea un conjunto que contiene los elementos de la colección especificada, colocados por orden natural. |
| <code>E first()</code> | → Devuelve el primer elemento (el de orden más bajo). |
| <code>SortedSet<E> headSet(E toElement)</code> | → Devuelve un conjunto ordenado que contiene los elementos de menor orden que el indicado. |
| <code>E last()</code> | → Devuelve el último elemento (el de orden más alto). |
| <code>SortedSet<E> subSet(E fromElem, E toElem)</code> | → Devuelve un conjunto ordenado que contiene los elementos desde <code>fromElem</code> , incluido, hasta <code>toElem</code> , no incluida. |
| <code>SortedSet<E> tailSet(E toElement)</code> | → Devuelve un conjunto ordenado que contiene los elementos de mayor o igual orden que el indicado. |

Ejemplo

```
Set <Integer> conjunto= new HashSet <Integer> ();

conjunto.add(1);
conjunto.add(9);
conjunto.add(5);
conjunto.add(9);

System.out.println(conjunto.size()); // 3

for (int n: conjunto)
    System.out.println(n); // 9 1 5 (en cualquier orden)
```

8.2.3 interface Map<K, V>

Con ella crearemos diccionarios, en los que a cada valor (clave) de tipo K se le asocia un valor de tipo V. Es importante saber que usando esta interfaz:

- El respeto al orden varía según la clase que utilicemos (de entre las que implementan esta interfaz, lo vemos a continuación).
- Las claves (K) no pueden estar duplicadas.
- El tamaño se adapta dinámicamente a lo que sea necesario.

Algunos de los métodos que incluye la interfaz, por lo tanto comunes a todas las clases que lo implementan, son:

- | | |
|---------------------------------------|--|
| • void clear() | → Elimina todos los elementos del diccionario. |
| • boolean containsKey(Object clave) | → Devuelve true si el conjunto contiene un elemento con esa clave. |
| • boolean containsValue(Object value) | → Devuelve true si el conjunto contiene un elemento con ese valor. |
| • boolean equals(Object x) | → Compara el objeto indicado con el diccionario. |
| • V get(Object clave) | → Devuelve el valor asociado a la clave especificada. |
| • boolean isEmpty() | → Devuelve true si no hay elementos en el diccionario. |
| • Set<K> keySet() | → Devuelve un conjunto (Set) con las claves del diccionario. |
| • V put(K clave, V valor) | → Asocia el valor especificado con la clave especificada. Devuelve null si no existía un elemento con tal clave. |
| • V remove(Object clave) | → Elimina el elemento (clave y valor) cuya clave es la especificada. Devuelve el valor que tenía asociado, o null si no existía un elemento con tal clave. |
| • int size() | → Devuelve el número de elementos del diccionario. |
| • Collection<V> values() | → Devuelve un Collection con los valores del diccionario. |

Clases que implementan la interfaz Map

Las clases que implementan diccionarios, y por lo tanto aquellas a las que llamaremos para crear colecciones de este tipo, son:

- HashMap<K, V>

Es una colección con gran economía en tiempo y memoria.

No respeta el orden de inserción ni ordena los elementos por sus claves.

Aparte de los métodos de la interfaz, esta clase tiene algunos importantes:

- | | |
|---------------------|---|
| HashMap() | → Constructor que crea un diccionario vacío, de capacidad inicial 16. |
| HashMap(int tamaño) | → Constructor que crea un diccionario del tamaño inicial indicado. |

Destacamos, aquí cada elemento está formado por una clave, y un valor asociado a la misma. Ambos pueden ser de cualquier tipo o clase.

Podemos entender este tipo de colecciones como arrays bidimensionales de orden 2xn, de tamaño dinámico.

Esta interfaz, así como las clases que estudiaremos (que la implementan) NO implementan la interfaz Collection<E>, ni Iterable<E>.

Todas las clases que vemos, además, reescriben el método clone(), de Object.

El modo de ordenación de las dos clases siguientes se puede variar utilizando constructores no incluidos en estos apuntes. Consulta las APIs.

- **LinkedHashMap<K,V>**

Es una colección voluminosa.

Respeto el orden de inserción.

Aparte de los métodos de la interfaz, esta clase tiene algunos importantes:

`LinkedHashMap()`

→ Constructor que crea un diccionario ordenado vacío, de capacidad inicial 16.

`LinkedHashMap(int tamaño)`

→ Constructor que crea un diccionario ordenado del tamaño inicial indicado.

- **TreeMap<K,V>**

Es una colección lenta y voluminosa.

Ordena los elementos por claves, por **orden natural**.

Aparte de los métodos de la interfaz, esta clase tiene algunos importantes:

`TreeMap()`

→ Constructor que crea un diccionario ordenado vacío.

`K firstKey()`

→ Devuelve la primera clave del diccionario (la de orden más bajo).

`SortedMap<K,V> headMap(K toKey)`

→ Devuelve un diccionario ordenado que contiene los elementos cuyas claves son de menor orden que la indicada.

`K lastKey()`

→ Devuelve la última clave del diccionario (la de orden más alto).

`SortedMap<K,V> subMap(K fromKey, K toKey)`

→ Devuelve un diccionario ordenado que contiene los elementos cuyas claves van desde `fromKey`, incluida, hasta `toKey`, no incluida.

`SortedMap<K,V> tailMap(K fromKey)`

→ Devuelve un diccionario ordenado que contiene los elementos cuyas claves son de mayor o igual orden que la indicada.

SortedMap es otra interfaz que implementan estas clases que estamos estudiando.

Ejemplo

```
Map <String, String> mapa=new HashMap <String, String> () ;

mapa.put("uno", "one");
mapa.put("dos", "two");
mapa.put("tres", "three");
mapa.put("cuatro", "four");
mapa.put("tres", "33");

System.out.println(mapa.size());

for (String clave: mapa.keySet()) {
    String valor= mapa.get(clave);
    System.out.println(clave + " -> " + valor);
}
```

El resultado por pantalla es:

```
cuatro -> four
tres -> 33
uno -> one
dos -> two
```

Con la evolución del lenguaje, el uso de Iterator está cayendo. Emplearemos formas más rápidas y sencillas de recorrer colecciones de datos.

8.2.4 Interface Iterator<E>

Permite recorrer los elementos de una colección, siempre y cuando esta colección sea implementación de la interfaz Iterable.

Esto impone algo obvio, y es que para poder recorrer una colección usando Iterator, dicha colección debe contener un método iterator() que devuelva un Iterator de la misma.

Los métodos que incluye la interfaz Iterator son:

- boolean hasNext() → Devuelve true si la iteración tiene más elementos.
- E next() → Devuelve el siguiente elemento de la iteración.
- void remove() → Elimina, de la colección, el último elemento devuelto en la iteración.

Uso con while

```
List <E> lista = ...

Iterator <E> it = lista.iterator();

while ( it.hasNext() ) {
    E elemento= it.next();
    ... haz algo con elemento ...
}
```

Uso con for

```
List <E> lista = ...

for (Iterator <E> it = lista.iterator(); it.hasNext(); ) {
    E elemento= it . next ();
    ... haz algo con elemento ...
}
```

Ejemplo 1

```
List lista= new ArrayList () ;

lista.add(1);
lista.add(9);
lista.add(1, 5);

System.out.println(lista.size()); // 3
System.out.println(lista.get(0)); // 1
System.out.println(lista.get(1)); // 5
System.out.println(lista.get(2)); // 9

for (Iterator it= lista.iterator(); it.hasNext(); ) {
    int n= (Integer) it.next();
    System.out.print(n); // 1 5 9
}
```

Ejemplo 2

```
Set conjunto= new HashSet () ;

conjunto.add(1);
conjunto.add(9);
conjunto.add(5);
conjunto.add(9);

System.out.println(conjunto.size()); // 3

for (Iterator it= conjunto.iterator(); it.hasNext(); ) {
    int n= (Integer) it.next();
    System.out.println(n); // 9 1 5 (en cualquier orden)
}
```

Ejemplo del uso de Iterator con una lista (List).

Ejemplo del uso de Iterator con un conjunto (Set).

Como veremos en la práctica, estas operaciones las podemos realizar con el nuevo bucle for, sin necesidad de usar Iterator.

CUADRO DE COLECCIONES DINÁMICAS DE DATOS

INTERFAZ	SON...	CLASE	SON...	ORDEN DE LOS ELEMENTOS	¿ADMITE DUPLICADOS?
List<E>	Listas de objetos de tipo <i>E</i> (el que queramos)	ArrayList<E>	Arrays unidimensionales dinámicos.	Por inserción	SI
		LinkedList<E>	Listas encadenadas.		
Set<E>	Conjuntos de objetos de tipo <i>E</i> (el que queramos)	HashSet<E>	Conjuntos brutos.	NINGUNO	NO
		TreeSet<E>	Conjuntos ordenados.	Orden Natural	
Map<K, V>	Diccionarios: A cada valor de tipo <i>K</i> (clave) le hace corresponder uno de tipo <i>V</i> (valor).	HashMap<K, V>	Diccionarios desordenados.	NINGUNO	LAS CLAVES NO
		LinkedHashMap<K, V>	Diccionarios encadenados.	Por inserción	
		TreeMap<K, V>	Diccionarios ordenados	Orden natural (por claves)	