Lenguaje de Programación

Java

INIDICE

1. Introducción

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

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Desarrollan un código "neutro" que no depende del tipo de electrodoméstico, el cual se ejecuta sobre una "máquina hipotética o virtual" denominada **Java Virtual Machine** (**JVM**). Es la **JVM** quien interpreta el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: "**Write Once, Run Everywhere**".

Los programas desarrollados en Java presentan diversas ventajas frente a los desarrollados en otros lenguajes como C/C++. La ejecución de programas en Java tiene muchas posibilidades: ejecución como aplicación independiente **(Stand-alone Application**), ejecución como applet, ejecución como servlet, etc.. Un applet es una aplicación especial que se ejecuta dentro de un navegador o browser (por ejemplo Netscape Navigator o Internet Explorer) al cargar una página HTML desde un servidor Web. El **applet** se descarga desde el servidor y no requiere instalación en el ordenador donde se encuentra el browser. Un **servlet** es una aplicación sin interface gráfica que se ejecuta en un servidor de Internet. La ejecución como aplicación independiente es análoga a los programas desarrollados con otros lenguajes.

Java permite fácilmente el desarrollo tanto de arquitecturas cliente-servidor como de aplicaciones distribuidas, consistentes en crear aplicaciones capaces de conectarse a otros ordenadores y ejecutar tareas en varios ordenadores simultáneamente, repartiendo por lo tanto el trabajo. Aunque también otros lenguajes de programación permiten crear aplicaciones de este tipo, Java incorpora en su propio API estas funcionalidades.

2. Características de JAVA.

Las principales características de JAVA son: es simple, Orientado a Objetos, distribuido, robusto, seguro, portable, interpretado, multihebras, dinámico y de arquitectura neutral.

- **Simple**. Basado en C++ elimina: la aritmética de punteros, las referencias, las estructuras y uniones, las definiciones de tipos y macros y la necesidad de liberar memoria.
- **Orientado a Objetos**. Incluye:
  - Encapsulación, herencia y polimorfismo.
  - Interfaces para suplir la carencia de herencia múltiple.
  - Resolución dinámica de métodos.
- **Distribuido**.
  - Extensas capacidades de comunicaciones.
  - Permite actuar con http y ftp.
  - El lenguaje no es distribuido, pero incorpora facilidades para construir programas distribuidos.
  - Se están incorporando características para hecerlo distribuido.
- **Robusto**. Realiza continuos chequeos en tiempo de compilación y en tiempo de ejecución.
- **Seguro**.
  - No hay punteros
  - El cast hacia lo general es ímplicito.
  - Los bytecodes pasan varios test antes de ser ejecutados.
- **Portable**.
  - Mismo código en distintas arquitecturas.
  - Define la longitud de sus tipos independientemente de la plataforma.
  - Construye sus interfaces en base a un sistema abstracto de ventanas.
- **Interpretado**.
  - Para conseguir la independencia del S.O. genera bytecodes.
  - El intérprete toma cada bytecode y lo interpreta.
  - El mismo intérprete corre en distintas arquitecturas.
- **Multihebras**.
  - Permite crear tareas o hebras.
  - Sincronización de métodos.
  - Comunicación de tareas.
- **Dinámico**.
  - Conecta los módulos que intervienen en una aplicación en el momento de su ejecución
  - No hay necesidad de enlazar previamente.

2.1. El compilador de Java.

Se trata de una de las herramientas de desarrollo incluidas en el **JDK**. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de Java (con extensión ***.java**). Si no encuentra errores en el código genera los ficheros compilados (con extensión ***.class**). En otro caso muestra la línea o líneas erróneas. En el **JDK** de Sun dicho compilador se llama javac.exe. Tiene numerosas opciones, algunas de las cuales varían de una versión a otra. Se aconseja consultar la documentación de la versión del JDK utilizada para obtener una información detallada de las distintas posibilidades.

2.2. La Java Virtual Machine.

Tal y como se ha comentado al comienzo del capítulo, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de Sun a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se plantea la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código "neutro" el cual estuviera preparado para ser ejecutado sobre una "máquina hipotética o virtual", denominada **Java Virtual Machine** (**JVM**). Es esta JVM quien interpreta este código neutro convirtiéndolo a código particular de la CPU o chip utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La **JVM** es el intérprete de Java. Ejecuta los "**bytecodes**" (ficheros compilados con extensión ***.class**) creados por el compilador de Java (javac.exe). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado **JIT** (**Just-In-Time Compiler**), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

3. Nomenclatura habitual en la programación en JAVA.

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

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

4. Estructura general de un programa JAVA.

La estructura habitual de un programa realizado en cualquier lenguaje orientado a objetos u **OOP** (**Object Oriented Programming**), y en particular en el lenguaje Java es la siguiente: aparece una clase que contiene el programa principal (aquel que contiene la función ***main***()) y algunas clases de usuario (las específicas de la aplicación que se está desarrollando) que son utilizadas por el programa principal. Los ficheros fuente tienen la extensión *.java, mientras que los ficheros compilados tienen la extensión *.class.

Un fichero fuente (*.java) puede contener más de una clase, pero sólo una puede ser **public**. El nombre del fichero fuente debe coincidir con el de la clase **public** (con la extensión *.java). Si por ejemplo en un fichero aparece la declaración (**public class MiClase {...})** entonces el nombre del fichero deberá ser MiClase.java. Es importante que coincidan mayúsculas y minúsculas ya que MiClase.java y miclase.java serían clases diferentes para Java. Si la clase no es public, no es necesario que su nombre coincida con el del fichero. Una clase puede ser public o package (default), pero no **private** o **protected**. Estos conceptos se explican posteriormente.

De ordinario una aplicación está constituida por varios ficheros ***.class**. Cada clase realiza unas funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. La aplicación se ejecuta por medio del nombre de la clase que contiene la función main() (sin la extensión *.**class**). Las clases de Java se agrupan en packages, que son librerías de clases. Si las clases no se definen como pertenecientes a un **package**, se utiliza un package por defecto (default) que es el directorio activo. Los packages se estudian con más detenimiento en siguientes apartados.

```
// fichero EjCadena.java
public class EjCadena
{
  public static void main(String args[])
  {
    Cadena cad=new Cadena(args[0]);
    cad.invierteCadena();
    cad.visualizaCadena();
    cad.encriptaCadena();
    cad.visualizaCadena();
    cad.desencriptaCadena();
    cad.visualizaCadena();
  }
}
```

```
// fichero Cadena.java
public class Cadena
{
  private String cadena="";
  public Cadena(String cadena)
  {
    this.cadena=cadena;
  }
  public int ordinal(char c) {
    return ((int) c);
  }
  public char ascii(int i)
  {
    return ((char) i);
  }
  public void invierteCadena()
  {
    String cadena2="";
    for(int i=cadena.length()-1;i>=0;i--)
      cadena2=cadena2+cadena.charAt(i);
    cadena=cadena2;
  }
  public void encriptaCadena()
  {
    String cadena2="";
    char c;
    for(int i=0;i<cadena.length();i++)
    {
      c=cadena.charAt(i);
      cadena2=cadena2+ascii(ordinal(c)+3);
    }
    cadena=cadena2;
  }
  public void desencriptaCadena()
  {
    String cadena2="";
    char c;
    for(int i=0;i<cadena.length();i++)
    {
      c=cadena.charAt(i);
      cadena2=cadena2+ascii(ordinal(c)-3);
    }
    cadena=cadena2;
  }
  public void visualizaCadena()
  {
    System.out.println(cadena);
  }
}
```

5. Variables.

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

1. Variables de **tipos primitivos**. Están definidas mediante un valor único.
2. Variables **referencia**. Las variables referencia son referencias o nombres de una información más compleja: **arrays** u **objetos** de una determinada clase.

Desde el punto de vista de su papel en el programa, las variables pueden ser:

1. Variables **miembro** de una **clase**: Se definen en una clase, fuera de cualquier método; pueden ser tipos primitivos o referencias.
2. Variables **locales**: Se definen dentro de un método o más en general dentro de cualquier bloque entre llaves {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también tipos primitivos o referencias.

5.1. Nombres de Variables.

Los nombres de variables en Java se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por Java como operadores o separadores (,.+-*/ etc.).

Existe una serie de palabras reservadas las cuales tienen un significado especial para Java y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

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

(*) son palabras reservadas, pero no se utilizan en la actual implementación del lenguaje Java.

5.2. Tipos Primitivos de Variables.

Se llaman tipos primitivos de variables de Java a aquellas variables sencillas que contienen los tipos de información más habituales: valores **boolean**, **caracteres** y valores **numéricos** enteros o de punto flotante.

Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores **true** y **false** (**boolean**); un tipo para almacenar **caracteres** (**char**), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (**byte**, **short**, **int** y **long**) y dos para valores reales de punto flotante (float y double). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la siguiente tabla:

Tipo de variable	Descripción
boolean	1 byte. Valores true y false
char	2 bytes. Unicode. Comprende el código ASCII
byte	1 byte. Valor entero entre -128 y 127
short	2 bytes. Valor entero entre -32768 y 32767
int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
long	8 bytes. Valor entre –9.223.372.036.854.775.808 7 9.223.372.036.854.775.807
float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Los tipos primitivos de Java tienen algunas características importantes que se resumen a continuación:

1. El tipo **boolean** no es un valor numérico: sólo admite los valores true o false. El tipo **boolean** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser boolean.
2. El tipo **char** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.
3. Los tipos **byte**, **short**, **int** y **long** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en Java no hay enteros unsigned.
4. Los tipos **float** y **double** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.
5. Se utiliza la palabra **void** para indicar la ausencia de un tipo de variable determinado.
6. A diferencia de C/C++, los tipos de variables en Java están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, **un** int ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador.
7. **Extensiones** de Java aprovechar la arquitectura de los procesadores Intel, que permiten realizar operaciones con una precisión extendida de 80 bits.

5.3. Cómo se definen e inicializan las variables.

Una variable se define especificando el tipo y el nombre de la variable. Estas variables pueden ser tanto de tipos **primitivos** como **referencias** a objetos de alguna clase perteneciente al **API** de Java o generada por el usuario. Las variable primitivas se inicializan a cero (salvo **boolean** y **char**, que se inicializan a false y '**\0**') si no se especifica un valor en su declaración. Análogamente las variables de tipo referencia son inicializadas por defecto a un valor especial: **null**.

Es importante distinguir entre la referencia a un **objeto** y el objeto mismo. Una referencia es una variable que indica dónde está en la memoria del ordenador un objeto. Al declarar una referencia todavía no se encuentra "**apuntando**" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración) luego se le asigna el valor **null**. Si se desea que esta referencia apunte a un nuevo objeto es necesario utilizar el operador new. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones). También es posible igualar la referencia declarada a un objeto existente previamente.

Un tipo particular de referencias son los **arrays** o vectores, sean estos de variables primitivas (por ejemplo vector de enteros) o de objetos. En la declaración de una referencia de tipo **array** hay que incluir los corchetes []. En los siguientes ejemplos aparece cómo crear un vector de 10 números enteros y cómo crear un vector de elementos **MyClass**. A su vez se garantiza que los elementos del vector son inicializados a **null** o a cero (según el tipo de dato) en caso de no indicar otro valor.

Ejemplos de declaración e inicialización de variables:

```
int x;                          // Declaración de la variable primitiva x. Se inicializa a 0
int y = 5;                      //Declaración de la variable primitiva y. Se inicializa a 5
MyClass unaRef;                 //Declaración de una referencia a un objeto MyClass.
                                // Se inicializa a null
unaRef = new MyClass();         //La referencia "apunta" al nuevo objeto creado
                                // Se ha utilizado el constructor por defecto
MyClass segundaRef = unaRef;    //Declaración de referencia a un objeto
                                // Se inicializa al mismo valor que unaRef
int [] vector;                   // Declaración de un array. Se inicializa a null
vector = new int[10];           // Vector de 10 enteros, inicializados a 0
double [] v = {1.0, 2.65, 3.1}; // Declaración e inicialización de un
                                // vector de 3 elementos con los valores entre llaves
MyClass [] lista=new MyClass[5]; //Crea un vector de 5 referencias a objetos
                                // Las 5 referencias son inicializadas a null
lista[0] = unaRef;              // Se asigna a lista[0] el mismo valor que unaRef
lista[1] = new MyClass();       // Se asigna a lista[1] la referencia al
                                // nuevo objeto El resto (lista[2]…lista[4] siguen con valor null
```

En el ejemplo mostrado las referencias unaRef, segundaRef y lista[0] actuarán sobre el mismo objeto. Es equivalente utilizar cualquiera de las referencias ya que el objeto al que se refieren es el mismo.

5.4. Visibilidad y vida de las variables.

Se entiende por **visibilidad**, **ámbito** o **scope** de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en cualquier expresión. En Java todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves {}, es decir dentro de un bloque, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque **if** no serán válidas al finalizar las sentencias correspondientes a dicho **if** y las

variables miembro de una clase (es decir declaradas entre las llaves {} de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

Las variables miembro de una clase declaradas como **public** son accesibles a través de una referencia a un objeto de dicha clase utilizando el operador punto (.). Las variables miembro declaradas como **private** no son accesibles directamente desde otras clases. Las funciones miembro de una clase tienen acceso directo a todas las variables miembro de la clase sin necesidad de anteponer el nombre de un objeto de la clase. Sin embargo las funciones miembro de una clase B derivada de otra A, tienen acceso a todas las variables miembro de A declaradas como **public** o **protected**, pero no a las declaradas como **private**. Una clase derivada sólo puede acceder directamente a las variables y funciones miembro de su clase base declaradas como **public** o **protected**. Otra característica del lenguaje es que es posible declarar una variable dentro de un bloque con el mismo nombre que una variable miembro, pero no con el nombre de otra variable local. La variable declarada dentro del bloque oculta a la variable miembro en ese bloque. Para acceder a la variable miembro oculta será preciso utilizar el operador **this**.

Uno de los aspectos más importantes en la programación orientada a objetos (OOP) es la forma en la cual son creados y eliminados los objetos. La forma de crear nuevos objetos es utilizar el operador new. Cuando se utiliza el operador new, la variable de tipo referencia guarda la posición de memoria donde está almacenado este nuevo objeto. Para cada objeto se lleva cuenta de por cuántas variables de tipo referencia es apuntado. La eliminación de los objetos la realiza el denominado **garbage collector**, quien automáticamente libera o borra la memoria ocupada por un objeto cuando no existe ninguna referencia apuntando a ese objeto. Lo anterior significa que aunque una variable de tipo referencia deje de existir, el objeto al cual apunta no es eliminado si hay otras referencias apuntando a ese mismo objeto.

6. Operadores de JAVA.

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

6.1. Operadores aritméticos.

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

6.2. Operadores de asignación.

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

```
variable = expression;
```

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

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

6.3. Operadores unarios.

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

6.4. Operador instanceof.

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

```
objectName instanceof ClassName
```

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

6.5. Operador condicional ?.

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

```
booleanExpression ? res1 : res2
```

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

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

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

6.6. Operadores incrementales.

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

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

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

6.7. Operadores relacionales.

Los operadores relacionales sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor **boolean** (**true** o **false**) según se cumpla o no la relación considerada. La siguiente tabla muestra los operadores relacionales de **Java**.

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

Estos operadores se utilizan con mucha frecuencia en las **bifurcaciones** y en los **bucles**.

6.8. Operadores lógicos.

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

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	NOT	! op	true si op es false y false si es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalua op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

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

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

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

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

6.10. Operadores que actúan a nivel de bits.

Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o **flags**, esto es, variables de tipo entero en las que cada uno de sus bits indican si una opción está activada o no.

La Tabla siguiente muestra los operadores de Java que actúan a nivel de bits.

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2(positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits
~	op1 ~ op2	Operador complemento

En binario, las potencias de dos se representan con un único bit activado. Por ejemplo, los números (1, 2, 4, 8, 16, 32, 64, 128) se representan respectivamente de modo binario en la forma (00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000), utilizando sólo 8 bits. La suma de estos números permite construir una variable **flags** con los bits activados que se deseen. Por ejemplo, para construir una variable **flags** que sea 00010010 bastaría hacer **flags**=2+16. Para saber si el segundo bit por la derecha está o no activado bastaría utilizar la sentencia,

```
if (flags & 2 == 2) {...}
```

6.11. Precedencia de operadores.

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

operadores postfijos	[] . (params) expr++ expr--
operadores unarios	++expr --expr +expr -expr ~ !
creacion o cast	new (type)expr
multiplicación / división	* / %
suma / resta	+ -
shift	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
lógico AND	&&
lógico OR	||
condicional	? :
asignación	= += -= *= /= %= &= ^= |= <<= >>= >>>=

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

7. Estructuras de programación.

Las estructuras de programación o estructuras de control permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados bifurcaciones y bucles. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a concepto, aunque su sintaxis varía de un lenguaje a otro. La sintaxis de Java coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no suponga ninguna dificultad adicional.

7.1. Sentencias o expresiones.

Una expresión es un conjunto variables unidos por operadores. Son órdenes que se le dan al computador para que realice una tarea determinada.

Una sentencia es una expresión que acaba en punto y coma (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:

```
i = 0; j = 5; x = i + j;// Línea compuesta de tres sentencias
```

7.2. Comentarios.

Existen dos formas diferentes de introducir comentarios entre el código de Java (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras "//" en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos **/*…*/.** Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una línea utilizando
//las dos barras. Requiere incluir dos barras al comienzo de línea
/* Esta segunda forma es mucho más cómoda para comentar un número elevado de líneas ya que sólo
requiere modificar el comienzo y el final. */
```

En Java existe además una forma especial de introducir los comentarios (utilizando **/**…*/** más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las clases y packages desarrollados por el programador. Una vez introducidos los comentarios, el programa **javadoc.exe** (incluido en el JDK) genera de

forma automática la información de forma similar a la presentada en la propia documentación del JDK. La sintaxis de estos comentarios y la forma de utilizar el programa javadoc.exe se puede encontrar en la información que viene con el JDK.

7.3. Bifurcaciones.

Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el flujo de ejecución de un programa. Existen dos bifurcaciones diferentes: **if** y **switch**.

7.3.1.- Bifurcación if.

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor true). Tiene la forma siguiente:

```
if (booleanExpression)
{
  instrucciones;
}
```

Las llaves {} sirven para agrupar en un bloque las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del if.

7.3.2. Bifurcación if else.

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el else se ejecutan en el caso de no cumplirse la expresión de comparación (false),

```
if (booleanExpression)
{
  instrucciones1;
}
else
{
  instrucciones2;
}
```

7.3.3. Sentencia switch.

Se trata de una alternativa a la bifurcación **if elseif else** cuando se compara la misma expresión con distintos valores. Su forma general es la siguiente:

```
switch (expression)
{
  case value1: statements1; break;
  case value2: statements2; break;
  case value3: statements3; break;
  case value4: statements4; break;
  case value5: statements5; break;
  case value6: statements6; break;
  [default: statements7;]
}
```

Las características más relevantes de switch son las siguientes:

1. Cada sentencia **case** se corresponde con un único valor de **expression**. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos.
2. Los valores no comprendidos en ninguna sentencia case se pueden gestionar en **default**, que es opcional.
3. En ausencia de **break**, cuando se ejecuta una sentencia case se ejecutan también todas las que van a continuación, hasta que se llega a un break o hasta que se termina el **switch**.

Ejemplo:

```
char c = (char)(Math.random()*26+'a');
                          // Generación aleatoria de letras minúsculas
System.out.println("La letra " + c );
switch (c)
{
  case 'a': // Se compara con la letra a
  case 'e': // Se compara con la letra e
  case 'i': // Se compara con la letra i
  case 'o': // Se compara con la letra o
  case 'u': // Se compara con la letra u
    System.out.println(" Es una vocal ");
    break;
  default:
    System.out.println(" Es una consonante ");
}
```

7.4. Bucles.

Un bucle se utiliza para realizar un proceso repetidas veces. Se denomina también **lazo** o **loop**. El código incluido entre las llaves {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (**booleanExpression**) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

7.4.1. Bucle while.

Las sentencias **statements** se ejecutan mientras **Expression** sea **true**.

```
while (Expression)
{
  statements;
}
```

7.4.2. Bucle for.

La forma general del bucle **for** es la siguiente:

```
for (initialization; Expression; increment)
{
  statements;
}
```

que es equivalente a utilizar **while** en la siguiente forma,

```
initialization;
while (Epression)
{
  statements;
  increment;
}
```

La sentencia o sentencias **initialization** se ejecuta al comienzo del **for**, e **increment** después de statements. La booleanExpression se evalúa al comienzo de cada iteración; el bucle termina cuando la expresión de comparación toma el valor false. Cualquiera de las tres partes puede estar vacía. La initialization y el increment pueden tener varias expresiones separadas por comas.

7.4.3. Bucle do while.

Es similar al bucle while pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados los **statements**, se evalúa la condición: si resulta true se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a false finaliza el bucle.

```
do
{
  statements
} while (Expression);
```

7.4.4. Sentencias break y continue.

La sentencia **break** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando sin finalizar el resto de las sentencias.

La sentencia **continue** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración "i" que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración (i+1).

7.4.5. Sentencias break y continue con etiquetas.

Las etiquetas permiten indicar un lugar donde continuar la ejecución de un programa después de un break o **continue**. El único lugar donde se pueden incluir etiquetas es justo delante de un bloque de código entre llaves {} (**if, switch, do...while, while, for**) y sólo se deben utilizar cuando se tiene uno o más bucles (o bloques) dentro de otro bucle y se desea salir (break) o continuar con la siguiente iteración (**continue**) de un bucle que no es el actual.

La sentencia **break** labelName por lo tanto finaliza el bloque que se encuentre a continuación de **labelName**. Por ejemplo, en las sentencias,

```
bucleI:
// etiqueta o label
for( int i = 0, j = 0; i < 100 ; i++)
{
  while ( true )
  {
    if( (++j) > 5)
    {
      break bucleI;
    }    // Finaliza ambos bucles
    else
    {
      break;
    } // Finaliza el bucle interior ( while)
  }
}
```

la expresión break **buclel**; finaliza los dos bucles simultáneamente, mientras que la expresión break; sale del bucle **while** interior y seguiría con el bucle for en i. Con los valores presentados ambos bucles finalizarán con i = 5 y j = 6 (se invita al lector a comprobarlo).

La sentencia **continue** (siempre dentro de al menos un bucle) permite transferir el control a un bucle con nombre o etiqueta. Por ejemplo, la sentencia,

```
continue bucle1;
```

transfiere el control al bucle for que comienza después de la etiqueta bucle1: para que realice una nueva iteración:

```
bucle1:
for (int i=0; i<n; i++)
{
  bucle2:
  for (int j=0; j<m; j++)
  {
    ...
    if (expression)
      continue bucle1; then continue bucle2;
       ...
  }
}
```

7.4.6. Sentencia return.

Otra forma de salir de un bucle (y de un método) es utilizar la sentencia **return**. A diferencia de **continue** o break, la sentencia return sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del **return** (**return value**;).

7.4.7. Bloque try {...} catch {...} finally {...}.

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

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

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

El código dentro del bloque try está “vigilado”: Si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque catch que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques catch como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque **finally**, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error.

En el caso en que el código de un método pueda generar una Exception y no se desee incluir en dicho método la gestión del error (es decir los bucles try/catch correspondientes), es necesario que el método pase la Exception al método desde el que ha sido llamado. Esto se consigue mediante la adición de la palabra throws seguida del nombre de la Exception concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques try/catch o volver a pasar la Exception. De esta forma se puede ir pasando la Exception de un método a otro hasta llegar al último método del programa, el método **main()**.

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

```
void metodo1()
{
  ...
  try
  {
    ...
  // Código que puede lanzar las excepciones IOException y MyException
  }
  catch (IOException e1)
  {
    // Se ocupa de IOException simplemente dando aviso
    System.out.println(e1.getMessage());
  }
  catch (MyException e2)
  {
    // Se ocupa de MyException dando un aviso y finalizando la función
    System.out.println(e2.getMessage());
    return;
  }
  [finally
  {
    // Sentencias que se ejecutarán en cualquier caso
    ...
  }]
  ...
} // Fin del metodo1

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

8. Programación Orientada a Objetos en Java.

Java es un lenguaje orientado a objetos. En este punto se presenta la manera con la que se implementan las clases, objetos, métodos y demás elementos que son necesarios, para construir aplicaciones orientadas a objetos en Java.

8.1. Concepto de Clase.

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos.

```
[public] class Classname {
// definición de variables y métodos
...
}
```

donde la palabra **public** es opcional: si no se pone, la **clase** sólo es visible para las demás clases del **package**. Todos los métodos y variables deben ser definidos dentro del bloque {...} de la clase. En definitiva una clase es una plantilla para crear objetos que tienen los mismos atributos y responden a los mismos mensajes.

Un **objeto** (en inglés, **instance**) es un ejemplar concreto de una **clase**. Las clases son como tipos de variables, mientras que los objetos son como variables concretas de un tipo determinado.

```
Classname unObjeto;
Classname otroObjeto;
```

A continuación se enumeran algunas características importantes de las clases:

1. Todas las **variables** y **funciones** de Java deben pertenecer a una **clase**. No hay variables y funciones globales.
2. Si una clase **deriva** de otra (**extends**), **hereda** todas sus **variables** y **métodos**.
3. Java tiene una **jerarquía** de **clases** estándar de la que pueden derivar las clases que crean los usuarios.
4. Una clase sólo puede heredar de una única clase (en Java no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de **Object**. La clase **Object** es la base de toda la jerarquía de clases de Java.
5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase **public**. Este fichero se debe llamar como la clase **public** que contiene con extensión ***.java**. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.
6. Si una clase contenida en un fichero no es **public**, no es necesario que el fichero se llame como la clase.
7. Los métodos de una clase pueden referirse de modo global al objeto de esa clase al que se aplican por medio de la referencia **this**.

8. Las clases se pueden agrupar en **packages**, introduciendo una línea al comienzo del fichero (**package packageName**;). Esta agrupación en **packages** está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

8.2. Concepto de Interface.

Una **interface** es un conjunto de declaraciones de funciones. Si una clase implementa (**implements**) una interface, debe definir todas las funciones especificadas por la interface. Las interfaces pueden definir también variables finales (constantes). Una clase puede implementar más de una interface, representando una forma alternativa de la herencia múltiple.

En algunos aspectos las interfaces pueden utilizarse en lugar de las clases. Por ejemplo, las interfaces sirven para definir referencias a cualquier objeto de las clases que implementan esa interface. Este es un aspecto importante del polimorfismo.

Una interface puede derivar de otra o incluso de varias interfaces, en cuyo caso incorpora las declaraciones de todos los métodos de las interfaces de las que deriva.

9. Ejemplo de definición de una clase.

```
// fichero Circulo.java
public class Circulo extends geometria
{
  static int numCirculos = 0;
  public static final double PI=3.14159265358979323846;
  public double x, y, r;
  public Circulo(double x, double y, double r)
  {
    this.x=x;
    this.y=y;
    this.r=r;
    numCirculos++;
  }
  public Circulo(double r) { this(0.0, 0.0, r); }
  public Circulo(Circulo c) { this(c.x, c.y, c.r); }
  public Circulo() { this(0.0, 0.0, 1.0); }
  public double perimetro() { return 2.0 * PI * r; }
  public double area() { return PI * r * r; }
  // método de objeto para  comparar círculos
  public Circulo elMayor(Circulo c)
  {
    if (this.r>=c.r)
      return this;
    else
      return c;
  }
  // método de clase para comparar círculos
  public static Circulo elMayor(Circulo c, Circulo d)
  {
    if (c.r>=d.r)
      return c;
    else
      return d;
    }
} // fin de la clase Circulo
```

En este ejemplo se ve cómo dentro de la clase se definen las **variables miembro** y los métodos, que pueden ser de objeto o de clase (**static**). Se puede ver también cómo el nombre del fichero coincide con el de la clase **public** con la extensión ***.java**.

10. Variables miembro.

A diferencia de la programación algorítmica clásica, que estaba centrada en las funciones, la programación orientada a objetos está centrada en los datos. Una clase son unos datos y unos métodos que operan sobre esos datos.

10.1. Variables miembro de objeto.

Cada objeto, es decir cada ejemplar concreto de la clase, tiene su propia copia de las variables miembro. Las variables miembro de una clase (también llamadas campos) pueden ser de tipos **primitivos** (**boolean**, **int**, **long**, **double**, …) o referencias a objetos de otra clase (composición).

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembro de tipos primitivos se inicializan siempre de modo automático, incluso antes de llamar al **constructor** (false para **boolean**, la cadena vacía para **char** y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas en el constructor.

También pueden inicializarse explícitamente en la declaración, como las variables locales, por medio de constantes o llamadas a métodos (esta inicialización no está permitida en C++). Por ejemplo,

```
long nDatos = 100;
```

Las variables miembro se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada objeto que se crea de una clase tiene su propia copia de las variables miembro. Por ejemplo, cada objeto de la clase Circulo tiene sus propias coordenadas del centro x e y, y su propio valor del radio r. Se puede aplicar un método a un objeto concreto poniendo el nombre del objeto y luego el nombre del método separados por un punto. Por ejemplo, para calcular el área de un objeto de la clase Circulo llamado c1 se escribe: c1.area();

Las variables miembro pueden ir precedidas en su declaración por uno de los modificadores de acceso: **public**, **private**, **protected** y **package** (que es el valor por defecto y puede omitirse). Junto con los modificadores de acceso de la clase (**public** y **package**), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables miembro.

10.2. Variables miembro de clase (static).

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama **variables de clase** o variables **static**. Las variables **static** se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo PI en

la clase Circulo) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como numCirculos en la clase Circulo).

Las variables de clase son lo más parecido que Java tiene a las variables globales de C/C++.

Las **variables de clase** se crean anteponiendo la palabra **static** a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, Circulo.numCirculos es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembro **static** se inicializan con los valores por defecto (false para **boolean**, la cadena vacía para **char** y cero para los tipos numéricos) para los tipos primitivos, y con **null** si es una referencia.

Las variables miembro static se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método static o en cuanto se utiliza una variable static de dicha clase. Lo importante es que las variables miembro static se inicializan siempre antes que cualquier objeto de la clase.

11. Variables finales.

Una variable de un tipo primitivo declarada como final no puede cambiar su valor a lo largo de la ejecución del programa. Puede ser considerada como una constante, y equivale a la palabra **const** de C/C++.

Java permite separar la definición de la inicialización de una variable final. La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos. La variable final así definida es constante (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Pueden ser final tanto las **variables miembro**, como las **variables locales** o los **propios argumentos** de un **método**.

Declarar como final un objeto miembro de una clase hace constante la referencia, pero no el propio objeto, que puede ser modificado. En Java no es posible hacer que un objeto sea constante.

12. Métodos (funciones miembro).

12.1. Métodos de objeto.

Los métodos son funciones definidas dentro de una clase. Salvo los métodos static o de clase, se aplican siempre a un objeto de la clase por medio del operador punto (.). Dicho objeto es su argumento implícito. Los métodos pueden además tener otros argumentos explícitos que van entre paréntesis, a continuación del nombre del método.

La primera línea de la definición de un método se llama **declaración** o **header**; el código comprendido entre las llaves {…} es el **cuerpo** o **body** del método. Considérese el siguiente método tomado de la **clase Circulo**:

```
public Circulo elMayor(Circulo c)
{ // header y comienzo del método
  if (this.r>=c.r) // body
    return this; // body
  else // body
    return c; // body
} // final del método
```

El **header** consta del **cualificador de acceso** (**public**, en este caso), del tipo del valor de retorno (Circulo en este ejemplo, **void** si no tiene), del nombre de la función y de una lista de argumentos explícitos entre paréntesis, separador por comas. Si no hay argumentos explícitos se dejan los paréntesis vacíos.

Los métodos tienen visibilidad directa de las variables miembro del objeto que es su argumento implícito, es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia **this**, de modo discrecional (como en el ejemplo anterior con **this.r**) o si alguna variable local o argumento las oculta.

El valor de retorno puede ser un valor de un tipo primitivo o una referencia. En cualquier caso no puede haber más que un único valor de retorno (que puede ser un objeto o un array). Se puede devolver también una referencia a un objeto por medio de un nombre de interface. El objeto devuelto debe pertenecer a una clase que implemente esa interface.

Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una **super-clase**.

Los métodos pueden definir variables locales. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

12.2. Métodos sobrecargados (overloaded).

Al igual que C++, Java permite métodos sobrecargados (**overloaded**), es decir métodos distintos con el mismo nombre que se diferencian por el número y/o tipo de los argumentos. El ejemplo de la clase Circulo presenta dos métodos sobrecargados: los cuatro constructores y los dos métodos **elMayor()**.

A la hora de llamar a un **método sobrecargado**, Java sigue unas reglas para determinar el método concreto que debe llamar:

Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.

1. Si no existe un método que se ajuste exactamente, se intenta **promover** los argumentos actuales al **tipo inmediatamente superior** (por ejemplo char a int, int a long, float a double, etc.) y se llama el método correspondiente.
2. Si sólo existen métodos con argumentos de un tipo más amplio (por ejemplo, long en vez de int), el programador debe hacer un **cast explícito** en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
3. El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la sobrecarga de métodos es la **redefinición**. Una clase puede **redefinir** (**override**) un método heredado de una superclase. Redefinir un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la herencia.

12.3. Paso de argumentos a métodos.

En Java los argumentos de los tipos primitivos se pasan siempre por valor. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. No hay otra forma de modificar un argumento de un tipo primitivo dentro de un método y que incluirlo en una clase como variable miembro. Las referencias se pasan también por valor, pero a través de ellas se pueden modificar los objetos referenciados.

En Java no se pueden pasar métodos como argumentos a otros métodos (en C/C++ se pueden pasar como argumentos punteros a función). Lo que se puede hacer en Java es pasar una referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.

Dentro de un método se pueden crear variables locales de los tipos primitivos o referencias. Estas variables locales dejan de existir al terminar la ejecución del método. Los argumentos formales de un método (las variables que aparecen en el header del método para recibir el valor de los argumentos actuales) tienen categoría de variables locales del método.

13.4. Métodos de clase (static).

Análogamente, puede también haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama métodos de **clase** o **static**. Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia **this**. Un ejemplo típico de métodos static son los métodos matemáticos de la clase **java.lang.Math** (sin(), cos(), exp(), pow(), etc.). De ordinario el argumento de estos métodos será de un tipo primitivo y se le pasará como argumento explícito. Estos métodos no tienen sentido como métodos de objeto.

Los métodos y variables de clase se crean anteponiendo la palabra **static**. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase (por ejemplo, **Math.sin(ang)**, para calcular el seno de un ángulo).

Los métodos y las variables de clase son lo más parecido que Java tiene a las funciones y variables globales de C/C++ o Visual Basic.

Ejemplo: Clase punto.

```
// fichero punto.java
import java.lang.Math.*
class Punto {
  double x, y;
  void valorInicial(double vx, double vy)
  {
    x=vx;
    y=vy;
  }
  double distancia(Punto q)
  {
    double dx=x-q.x;
    double dy=y-q.y;
    return Math.sqrt(dx*dx+dy*dy);
  }
}

// fichero usoPunto.java
class UsoPunto
{
  public static void main(String arg [])
  {
    Punto p, q;
    double d;
    p=new Punto();
    p.valorInicial(5, 4);
    q=new Punto();
    q.valorInicial(2,2);
    d=p.distancia(q);
    System.out.println("Distancia: " + d);
  }
}
```

13.5. Constructores.

Un **constructor** es un **método que se llama automáticamente** cada vez que se crea un objeto de una clase. La principal misión del constructor es reservar memoria e inicializar las variables miembro de la clase.

Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase. Su argumento implícito es el objeto que se está creando.

De ordinario una clase tiene varios constructores, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos sobrecargados). Se llama constructor por defecto al constructor que no tiene argumentos. El programador debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un constructor de una clase puede llamar a otro constructor previamente definido en la misma clase por medio de la palabra this. En este contexto, la palabra this sólo puede aparecer en la primera sentencia de un constructor.

El constructor de una sub-clase puede llamar al constructor de su **super-clase** por medio de la palabra **super**, seguida de los argumentos apropiados entre paréntesis. De esta forma, un constructor sólo tiene que inicializar por sí mismo las variables no heredadas.

El constructor es tan importante que, si el programador no prepara ningún constructor para una clase, el compilador crea un constructor por defecto, inicializando las variables de los tipos primitivos a su valor por defecto, los Strings a la cadena vacía y las referencias a objetos a null. Si hace falta, se llama al constructor de la super-clase para que inicialice las variables heredadas.

Al igual que los demás métodos de una clase, los constructores pueden tener también los modificadores de acceso public, private, protected y package. Si un constructor es private, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos public y static (factory methods) que llamen al constructor y devuelvan un objeto de esa clase.

Dentro de una clase, los constructores sólo pueden ser llamados por otros constructores o por métodos static. No pueden ser llamados por los métodos de objeto de la clase.

Ejemplo: Clase Punto.

```
// fichero punto.java
import java.lang.Math.*;
class Punto
{
  double x, y;
  Punto(double x, double y) {this.x=x; this.y=y;}
  void valorInicial(double vx, double vy)
  {
    x=vx;
    y=vy;
  }
  double distancia(Punto q)
  {
    double dx=x-q.x;
    double dy=y-q.y;
    return Math.sqrt(dx*dx+dy*dy);
  }
}

// fichero usoPunto.java
class UsoPunto
{
  public static void main(String arg [])
  {
    Punto p, q;
    double d;
    p=new Punto(5, 4);
    q=new Punto(2, 2):
    d=p.distancia(q);
    System.out.println("Distancia: " + d);
  }
}
```

13.6. Inicializadores.

Java todavía dispone de una tercera línea de actuación para evitar que haya variables sin inicializar correctamente. Son los inicializadores, que pueden ser static (para la clase) o de objeto.

13.6.1. Inicializadores static.

Un inicializador static es un método (un bloque {…} de código) que se llama automáticamente al crear la clase (al utilizarla por primera vez). Se diferencia del constructor en que no es llamado para cada objeto, sino una sola vez para toda la clase.

Los inicializadores static se crean dentro de la clase, como métodos sin nombre y sin valor de retorno, con tan sólo la palabra static y el código entre llaves {...}. En una clase pueden definirse varios inicializadores static, que se llamarán en el orden en que han sido definidos.

Los inicializadores static se pueden utilizar para dar valor a las variables static. Además se suelen utilizar para llamar a métodos nativos, esto es, a métodos escritos por ejemplo en C/C++ (llamando a los métodos System.load() o System.loadLibrary(), que leen las librerías nativas).

Por ejemplo:

```
static
{
  System.loadLibrary("MyNativeLibrary");
}
```

13.6.2. Inicializadores de objeto.

Existen también inicializadores de objeto, que no llevan la palabra static. Se utilizan para las clases anónimas, que por no tener nombre no tienen constructor. En este caso se llaman cada vez que se crea un objeto de la clase anónima.

13.7. Resumen del proceso de creación de un objeto.

El proceso de creación de objetos de una clase es el siguiente:

1. Al crear el **primer objeto** de la **clase** o al utilizar el primer método o **variable static** se localiza la clase y se carga en memoria.
2. Se ejecutan los inicializadores **static** (sólo una vez).
3. Cada vez que se quiere crear un **nuevo objeto**:
   - se comienza reservando la memoria necesaria
   - se da valor por defecto a las variables miembro de los tipos primitivos
   - se ejecutan los inicializadores de objeto
   - se ejecutan los constructores

13.8. Destrucción de objetos (liberación de memoria).

En Java no hay destructores como en C++. El sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han perdido la referencia, esto es, objetos que ya no tienen ningún

nombre que permita acceder a ellos, por ejemplo, por haber llegado al final del bloque en el que habían sido definidos, porque a la referencia se le ha asignado el valor null o porque a la referencia se le ha asignado la dirección de otro objeto. A esta característica de Java se le llama garbage collection (recogida de basura).

En Java es normal que varias variables de tipo referencia apunten al mismo objeto. Java lleva internamente un contador de cuántas referencias hay sobre cada objeto. El objeto podrá ser borrado cuando el número de referencias sea cero. Como ya se ha dicho, una forma de hacer que un objeto quede sin referencia es cambiar ésta a **null**, haciendo por ejemplo:

```
ObjetoRef = null;
```

En Java no se sabe exactamente cuándo se va a activar el garbage collector. Si no falta memoria es posible que no se llegue a activar en ningún momento. No es pues conveniente confiar en él para la realización de otras tareas más criticas.

Se puede llamar explícitamente al garbage collector con el método **System.gc(),** aunque esto es considerado por el sistema sólo como una "sugerencia" a la **JVM**.

13.9. Finalizadores.

Los finalizadores son métodos que vienen a completar la labor del **garbage collector**. Un finalizador es un método que se llama automáticamente cuando se va a destruir un objeto (antes de que la memoria sea liberada de modo automático por el sistema). Se utilizan para ciertas operaciones de terminación distintas de liberar memoria (por ejemplo: cerrar ficheros, cerrar conexiones de red, liberar memoria reservada por funciones nativas, etc.). Hay que tener en cuenta que el **garbage collector** sólo libera la memoria reservada con new. Si por ejemplo se ha reservado memoria con funciones nativas en C (por ejemplo, utilizando la función malloc()), esta memoria hay que liberarla explícitamente con el método **finalize**().

Un finalizador es un método de objeto (no static), sin valor de retorno (void), sin argumentos y que siempre se llama finalize(). Los finalizadores se llaman de modo automático siempre que hayan sido definidos por el programador de la clase. Para realizar su tarea correctamente, un finalizador debería terminar siempre llamando al finalizador de su super-clase.

Tampoco se puede saber el momento preciso en que los finalizadores van a ser llamados. En muchas ocasiones será conveniente que el programador realice esas operaciones de finalización de modo explícito mediante otros métodos que él mismo llame.

El método **System.runFinalization()** "sugiere" a la JVM que ejecute los finalizadores de los objetos pendientes (que han perdido la referencia). Parece ser que para que este método se ejecute hay que llamar primero a gc() y luego a runFinalization().

13. Packages.

Un **package** es una agrupación de clases. Además el usuario puede crear sus propios packages. Para que una clase pase a formar parte de un package llamado **pkgName**, hay que introducir en ella la sentencia:

```
package pkgName;
```

que debe ser la primera sentencia del fichero sin contar comentarios y líneas en blanco.

Los nombres de los packages se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un package puede constar de varios nombres unidos por puntos (los propios **packages** de Java siguen esta norma, como por ejemplo **javax.swing.event**).

Todas las clases que forman parte de un package deben estar en el mismo directorio. Los nombres compuestos de los packages están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los nombres de las clases de Java sean únicos en Internet. Es el nombre del package lo que permite obtener esta característica. Una forma de conseguirlo es incluir el nombre del dominio (quitando quizás el país), como por ejemplo en el package siguiente:

```
es.proyectos.daw1.infor2.ordenar
```

Las clases de un package se almacenan en un directorio con el mismo nombre largo (**path**) que el package. Por ejemplo, la clase,

```
es.proyectos.daw1.infor2.ordenar.QuickSort.class
```

debería estar en el directorio,

```
CLASSPATH\es\proyectos\daw1\infor2\ordenar\QuickSort.class
```

donde **CLASSPATH** es una variable de entorno del PC que establece la posición absoluta de los directorios en los que hay clases de Java (clases del sistema o de usuario), en este caso la posición del directorio es en los discos locales del ordenador.

Los **packages** se utilizan con las finalidades siguientes:

1. Para agrupar clases relacionadas.
2. Para evitar conflictos de nombres (se recuerda que el dominio de nombres de Java es la Internet). En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del **package**.
3. Para ayudar en el **control** de la **accesibilidad** de **clases** y **miembros**.

13.1. Cómo funcionan los packages.

Con la sentencia **import packname**; se puede evitar tener que utilizar nombres muy largos, al mismo tiempo que se evitan los conflictos entre nombres. Si a pesar de todo hay conflicto entre nombres de clases, Java da un error y obliga a utilizar los nombres de las clases cualificados con el nombre del package.

El importar un package no hace que se carguen todas las clases del package: sólo se cargarán las clases public que se vayan a utilizar. Al importar un package no se importan los sub-packages. Éstos deben ser importados explícitamente, pues en realidad son packages distintos. Por ejemplo, al importar java.awt no se importa java.awt.event.

Es posible guardar en jerarquías de directorios diferentes los ficheros ***.class** y ***.java**, con objeto por ejemplo de no mostrar la situación del código fuente. Los **packages** hacen referencia a los ficheros compilados ***.class**.

En un programa de Java, una clase puede ser referida con su nombre completo (el nombre del **package** más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia **import** permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del **package** importado. Se importan por defecto el **package java.lang** y el **package** actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar **import**: para una clase y para todo un **package**:

```
import es.proyectos.daw1.infor2.ordenar.QuickSort.class;
import es.proyectos.daw1.infor2.ordenar.*;
```

que deberían estar en el directorio:

```
classpath\es\proyectos\daw1\infor2\ordenar
```

14. Herencia.

Se puede construir una clase a partir de otra mediante el mecanismo de la herencia. Para indicar que una clase deriva de otra se utiliza la palabra **extends**, como por ejemplo:

```
class CirculoGrafico extends Circulo {...}
```

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser redefinidas (**overridden**) en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la sub-clase (la clase derivada) "contuviera" un objeto de la super-clase; en realidad lo "amplía" con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de Java creadas por el programador tienen una super-clase. Cuando no se indica explícitamente una super-clase con la palabra **extends**, la clase deriva de **java.lang.Object**, que es la clase raíz de toda la jerarquía de clases de Java. Como consecuencia, todas las clases tienen algunos métodos que han heredado de Object.

La composición (el que una clase contenga un objeto de otra clase como variable miembro) se diferencia de la herencia en que incorpora los datos del objeto miembro, pero no sus métodos o interface (si dicha variable miembro se hace private).

Ejemplo: Clase Pixels.

```
// fichero pixels.java
class Pixels extends Punto
{
  byte color;
  Pixel(double vx, double vy, byte vc)
  {
    super(vx, vy); // debe ser la primera sentencia
    color=vc;
  }
  .......
}
```

14.1. La clase Object.

Como ya se ha dicho, la **clase Object** es la raíz de toda la jerarquía de clases de Java. Todas las clases de Java derivan de **Object**.

La clase **Object** tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase. Entre ellos se pueden citar los siguientes:

1. Métodos que pueden ser redefinidos por el programador:
   - **clone()** Crea un objeto a partir de otro objeto de la misma clase. El método original heredado de **Object** lanza una **CloneNotSupportedException**. Si se desea poder clonar una clase hay que implementar la **interface Cloneable** y redefinir el método clone(). Este método debe hacer una copia miembro a miembro del objeto original. No debería llamar al operador new ni a los constructores.
   - **equals()** Indica si **dos objetos son o no iguales**. Devuelve true si son iguales, tanto si son referencias al mismo objeto como si son objetos distintos con iguales valores de las variables miembro.
   - **toString()** Devuelve un **String** que contiene una representación del objeto como cadena de caracteres, por ejemplo para imprimirlo o exportarlo.
   - **finalize()** Este método finaliza la utilización del objeto.

2. Métodos que no pueden ser redefinidos (son métodos final):
   - **getClass()** Devuelve un objeto de la clase **Class**, al cual se le pueden aplicar métodos para determinar el nombre de la clase, su super-clase, las interfaces implementadas, etc. Se puede crear un objeto de la misma clase que otro sin saber de qué clase es.

   - **notify(), notifyAll() y wait()** Son métodos relacionados con las **threads**.

14.2. Redefinición de métodos heredados.

Una clase puede redefinir (volver a definir) cualquiera de los métodos heredados de su **super-clase** que no sean final. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Los métodos de la super-clase que han sido redefinidos pueden ser todavía accedidos por medio de la palabra super desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Los métodos redefinidos pueden ampliar los derechos de acceso de la super-clase (por ejemplo ser public, en vez de protected o package), pero nunca restringirlos.

Los métodos de clase o static no pueden ser redefinidos en las clases derivadas.

Ejemplo:

```
class Pixels extends Punto
{
  .........
  double distancia(double cx, double cy)
  {
    double dx=Math.abs(x-cx);
    double dy=Math.abs(y-cy);
    return dx+dy;            // distancia manhattan
  }
}
```

14.3. Clases y métodos abstractos.

Una clase abstracta (abstract) es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra abstract, como por ejemplo,

```
public abstract class Geometria { ... }
```

Una clase abstract puede tener métodos declarados como abstract, en cuyo caso no se da definición del método. Si una clase tiene algún método abstract es obligatorio que la clase sea abstract. En cualquier sub-clase este método deberá bien ser redefinido, bien volver a declararse como abstract (el método y la sub-clase).

Una clase **abstract** puede tener métodos que no son **abstract**. Aunque no se puedan crear objetos de esta clase, sus sub-clases heredarán el método completamente a punto para ser utilizado.

Como los métodos **static** no pueden ser redefinidos, un método abstract no puede ser static.

Ejemplo:

```
abstract class Figura
{
  abstract double perimetro();
  abstract double area();
}
class Cuadrado extends Figura
{
  double lado;
  cuadrado(double l) {lado=l;}
  double perimetro() {return lado*4;}
  double area() {return lado*lado;}
}
class Circulo extends Figura
{
  double radio;
  Circulo(double r) {radio=r;}
  double perimetro() {return 2*Math.PI*radio;}
  double area() {return Math.PI*radio*radio;}
}
```

```
class Ejherencia
{
  static public void main(String arg [])
  {
    Figura [] f=new Figura[2];
    F[0]=new Cuadrado(12);
    F[1]=new Circulo(7);
    for(int i=0;i<2;i++)
    {
      System.out.println("Figura "+i);
      System.out.println("\t"+f[i].perimetro());
      System.out.println("\t"+f[i].area());
    }
  }
}
```

14.4. Constructores en clases derivadas.

Ya se comentó que un constructor de una clase puede llamar por medio de la palabra **this** a otro constructor previamente definido en la misma clase. En este contexto, la palabra this sólo puede aparecer en la primera sentencia de un constructor.

De forma análoga el constructor de una clase derivada puede llamar al constructor de su super-clase por medio de la palabra super(), seguida entre paréntesis de los argumentos apropiados para uno de los constructores de la super-clase. De esta forma, un constructor sólo tiene que inicializar directamente las variables no heredadas.

La llamada al constructor de la super-clase debe ser la primera sentencia del **constructor** , excepto si se llama a otro constructor de la misma clase con **this**(). Si el programador no la incluye, Java incluye automáticamente una llamada al constructor por defecto de la super-clase, **super().** Esta llamada en cadena a los constructores de las super-clases llega hasta el origen de la jerarquía de clases, esto es al constructor de **Object**.

Como ya se ha dicho, si el programador no prepara un constructor por defecto, el compilador crea uno, inicializando las variables de los tipos primitivos a cero, los Strings a la cadena vacía y las referencias a objetos a **null**. Antes, incluirá una llamada al constructor de la super-clase.

En el proceso de finalización o de liberación de recursos (diferentes de la memoria reservada con new, de la que se encarga el **garbage collector**), es importante llamar a los finalizadores de las distintas clases, normalmente en orden inverso al de llamada de los constructores. Esto hace que el finalizador de la sub-clase deba realizar todas sus tareas primero y luego llamar al finalizador de la super-clase en la forma super.finalize(). Los métodos **finalize()** deben ser al menos protected, ya que el método finalize() de **Object** lo es, y no está permitido reducir los permisos de acceso en la herencia.

15. Clases y métodos finales.

Una **clase declarada final no puede tener clases derivadas.** Esto se puede hacer por motivos de seguridad y también por motivos de eficiencia, porque cuando el compilador sabe que los métodos no van a ser redefinidos puede hacer optimizaciones adicionales.

Análogamente, un método declarado como final no puede ser redefinido por una clase que derive de su propia clase.

16. Interfaces.

Una interface es un conjunto de declaraciones de **métodos (sin definición**). También puede definir constantes, que son implícitamente **public**, **static** y final, y deben siempre inicializarse en la declaración. Estos métodos definen un tipo **de** conducta. Todas las clases que implementan una determinada interface están obligadas a proporcionar una definición de los métodos de la interface, y en ese sentido adquieren una conducta o modo de funcionamiento.

Una clase puede implementar una o varias interfaces. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las interfaces, separados por comas, detrás de la palabra implements, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la super-clase en el caso de herencia. Por ejemplo,

```
public class CirculoGrafico extends Circulo implements Dibujable,
Cloneable
{
  ...
}
```

¿Qué diferencia hay entre una interface y una clase abstract? Ambas tienen en común que pueden contener varias declaraciones de métodos (la clase abstract puede además definirlos). A pesar de esta semejanza, que hace que en algunas ocasiones se pueda sustituir una por otra, existen también algunas diferencias importantes:

1. Una clase no puede heredar de dos clases **abstract**, pero sí puede heredar de una clase abstract e implementar una interface, o bien implementar dos o más interfaces.
2. Una clase no puede **heredar métodos** -definidos- de una interface, aunque sí constantes.
3. Las interfaces permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo **comportamiento**, independientemente de su situación en la jerarquía de clases de Java.
4. Las interfaces permiten “publicar” el comportamiento de una clase desvelando un mínimo de información.
5. Las interfaces tienen una jerarquía propia, independiente y más flexible que la de las clases, ya que tienen permitida la herencia múltiple.
6. De cara al polimorfismo, las referencias de un tipo interface se pueden utilizar de modo similar a las **clases abstract**.

16.1. Definición de interfaces.

Una interface se define de un modo muy similar a las clases. A modo de ejemplo se reproduce aquí la definición de la interface Dibujable:

```
// fichero Dibujable.java
import java.awt.Graphics;
public interface Dibujable
{
  public void setPosicion(double x, double y);
  public void dibujar(Graphics dw);
}
```

Cada interface **public** debe ser definida en un fichero ***.java** con el mismo nombre de la interface. Los nombres de las interfaces suelen comenzar también con mayúscula.

Las interfaces no admiten más que los modificadores de acceso **public** y **package**. Si la interface no es **public** no será accesible desde fuera del **package** (tendrá la accesibilidad por defecto, que es **package**). Los métodos declarados en una interface son siempre **public** y **abstract**, de modo implícito.

16.2. Herencia en interfaces.

Entre las interfaces existe una jerarquía (independiente de la de las clases) que permite herencia simple y múltiple. Cuando una interface deriva de otra, incluye todas sus constantes y declaraciones de métodos.

Una interface puede derivar de varias interfaces. Para la herencia de interfaces se utiliza asimismo la palabra **extends**, seguida por el nombre de las interfaces de las que deriva, separadas por comas.

Una interface puede ocultar una constante definida en una super-interface definiendo otra constante con el mismo nombre. De la misma forma puede ocultar, re-declarándolo de nuevo, la declaración de un método heredado de una super-interface.

Las interfaces no deberían ser modificadas más que en caso de extrema necesidad. Si se modifican, por ejemplo añadiendo alguna nueva declaración de un método, las clases que hayan implementado dicha interface dejarán de funcionar, a menos que implementen el nuevo método.

16.3. Utilización de interfaces.

Las constantes definidas en una interface se pueden utilizar en cualquier clase (aunque no implemente la interface) precediéndolas del nombre de la interface, como por ejemplo (suponiendo que PI hubiera sido definida en Dibujable):

```
area = 2.0*Dibujable.PI*r;
```

Sin embargo, en las clases que implementan la interface las constantes se pueden utilizar directamente, como si fueran constantes de la clase. A veces se crean interfaces

para agrupar constantes simbólicas relacionadas (en este sentido pueden en parte suplir las variables **enum** de C/C++).

De cara al polimorfismo, el nombre de una interface se puede utilizar como un nuevo tipo de referencia. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la interface. Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

Ejemplo:

```
interface Agrandable
{
  void zoom(int i);
}
class Circulo extends Figura implements Agrandable
{
  ........
  public void zoom(int i)
  {
    radio*=i;
  }
  .....
}
class Punto implements Agrandable
{
  .........
  public void zoom(int i)
  {
    x*=i;
    y*=i;
  }
  ..........
}
class EjInterfaz
{
  static public void main(String arg [])
  {
    Agrandable [] f=new Agrandable [2];
    f[0]=new Circulo(12);
    f[1]=new Punto(7, 4);
    for(int i=0;i<2;i++)
      f[i].zoom(3);
  }
}
```

17. Permisos de acceso en Java.

Una de las características de la **Programación Orientada a Objetos** es la encapsulación, que consiste básicamente en ocultar la información que no es pertinente o necesaria para realizar una determinada tarea. Los **permisos de acceso** de Java son una de las herramientas para conseguir esta finalidad.

17.1. Accesibilidad de los packages.

El primer tipo de accesibilidad hace referencia a la conexión física de los ordenadores y a los permisos de acceso entre ellos y en sus directorios y ficheros. En este sentido, un package es accesible si sus directorios y ficheros son accesibles (si están en un ordenador accesible y se tiene permiso de lectura). Además de la propia conexión física, serán accesibles aquellos packages que se encuentren en la variable CLASSPATH del sistema.

17.2. Accesibilidad de clases o interfaces.

En principio, cualquier clase o interface de un **package** es accesible para todas las demás clases del **package**, tanto si es **public** como si no lo es. Una clase public es accesible para cualquier otra clase siempre que su package sea accesible. Recuérdese que las clases e interfaces sólo pueden ser public o package (la opción por defecto cuando no se pone ningún modificador).

17.3. Accesibilidad de las variables y métodos miembros de una clase.

Desde dentro de la propia clase:

1. Todos los miembros de una clase son directamente accesibles (sin cualificar con ningún nombre o cualificando con la referencia **this**) desde dentro de la propia clase. Los métodos no necesitan que las variables miembro sean pasadas como argumento.
2. Los miembros **private** de una clase sólo son accesibles para la propia clase.
3. Si el constructor de una clase es **private**, sólo un método **static** de la propia clase puede crear objetos.

Desde una sub-clase:

1. Las sub-clases heredan los miembros **private** de su **super-clase**, pero sólo pueden acceder a ellos a través de métodos **public**, **protected** o **package** de la super-clase.

Desde otras clases del package:

1. Desde una clase de un **package** se tiene acceso a todos los miembros que no sean **private** de las demás clases del **package**.

Desde otras clases fuera del **package**:

1. Los métodos y variables son accesibles si la clase es **public** y el miembro es **public**.
2. También son accesibles si la clase que accede es una sub-clase y el miembro es **protected**.

La siguiente tabla muestra un resumen de los permisos de acceso de Java.

Visibilidad	**public**	**protected**	**private**	**default**
Desde la propia clase	SI	SI	SI	SI
Desde otra clase en el propio package	SI	SI	NO	SI
Desde otra clase fuera del package	SI	NO	NO	NO
Desde una sub-clase en el propio package	SI	SI	NO	SI
Desde una sub-clase fuera del propio package	SI	SI	NO	NO

18. Transformaciones de tipo: casting.

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de **int** a **double**, o de **float** a **long**. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia. En este apartado se explican brevemente estas transformaciones de tipo.

La conversión entre tipos primitivos es más sencilla. En Java se realizan de modo automático conversiones implícitas de un tipo a otro de más precisión, por ejemplo de int a long, de float a double, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto que el resultado de evaluar el miembro derecho.

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son conversiones inseguras que pueden dar lugar a errores (por ejemplo, para pasar a short un número almacenado como int, hay que estar seguro de que puede ser representado con el número de cifras binarias de short). A estas conversiones explícitas de tipo se les llama cast. El cast se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

A diferencia de C/C++, en Java no se puede convertir un tipo **numérico** a **boolean**.

19. Polimorfismo.

El polimorfismo tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama vinculación (**binding**). La vinculación puede ser temprana o estática (en tiempo de compilación) o tardía o dinámica (en tiempo de ejecución).

La vinculación temprana se realiza mediante funciones sobrecargadas. En el polimorfismo por sobrecarga los nombres de las funciones miembro de una clase pueden repetirse si varía el número o el tipo de los argumentos.

Ejemplo: Polimorfismo por vinculación temprana (por sobrecarga de funciones).

```
// fichero punto.java
import java.lang.Math.*;
class Punto
{
  double x, y;
  Punto(double x, double y) {this.x=x; this.y=y;}
  void valorInicial(double vx, double vy)
  {
    x=vx;
    y=vy;
  }
  double distancia(Punto q)
  {
    double dx=x-q.x;
    double dy=y-q.y;
    return Math.sqrt(dx*dx+dy*dy);
  }
  double distancia(double cx, double cy)
  {
    double dx=x-cx;
    double dy=y-cy;
    return Math.sqrt(dx*dx+dy*dy);
  }
}
```

Con funciones redefinidas en Java se utiliza siempre vinculación tardía, excepto

si el método es **final**. La vinculación tardía hace posible que, con un método declarado en una clase base (o en una interface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea el tipo de objeto y no el tipo de la referencia lo que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el polimorfismo necesita evaluación tardía.

El polimorfismo puede hacerse con referencias de s**uper-clases abstract, super-clases normales** e **interfaces**. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las interfaces permiten ampliar muchísimo las posibilidades del polimorfismo.

Ejemplo: Polimorfismo por vinculación tardía (vinculación dinámica).

```
class PruebaPixel2
{
  public static void main(String arg [])
  {
    Punto x=new Pixel(4, 3, (byte)2);
    Punto p=new Punto(4, 3);
    double d1, d2;
    d1=x.distancia(1, 1);        // método de Pixel
    d2=p.distancia(1, 1);        // método de Punto
    System.out.println("Distancia pixel: " + d1);
    System.out.println("Distancia punto: ") + d1);
  }
}
```

Si se desea acceder a algún método oculto de la clase base se ha de utilizar **super**.

Ejemplo: Acceso al método oculto de la class base.

```
class Punto
{
  .......
  void trasladar(double cx, double cy)
  {
    x+=cx;
    y+=cy;
  }
}

class Pixel extends Punto
{
  ........
  void trasladar(double cx, double cy)
  {
    super.trasladar(cx, cy);
    color=(byte)0;
  }
}
```