UNIVERSIDAD CATOLICA DE COLOMBIA
FACULTAD DE INGENIERIA DE SISTEMAS

CURSO: JAVA BASICO
PROFESOR: EMERSON CASTAÑEDA SANABRIA

TEMA: Las clases en Java.

OBJETIVOS:

- Comprender el manejo de las clases con Java, junto con todas sus implicaciones: variables, métodos. paquetes, herencia, permisos de acceso, transformaciones de tipo y polimorfismo, para llevar a cabo tareas de codificación de manera mas apropiada.

CONTENIDO:

1. Conceptos básicos.
2. Ejemplo de definición de una clase.
3. Variables miembro.
4. Variables finales.
5. Métodos (funciones miembro).
6. Packages.
7. Herencia.
8. Clases y métodos finales.
9. Interfaces.
10. Clases internas.
11. Permisos de acceso en java.
12. Transformaciones de tipo.
13. Polimorfismo.

DESARROLLO:

1. CONCEPTOS BÁSICOS

Concepto de Clase

Una clase es una agrupación de ***datos*** (variables o campos) y de ***funciones*** (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

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

donde la palabra ***public*** es opcional: si no se pone, la clase tiene la visibilidad por defecto, esto es, sólo es visible para las demás clases del ***package***. Todos los métodos y variables deben ser definidos dentro del ***bloque*** {...} de la clase.

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 archivo se pueden definir varias clases, pero en un archivo no puede haber más que una clase ***public***. Este archivo se debe llamar como la clase ***public*** que contiene con extensión ****.java***. Con algunas excepciones, lo habitual es escribir una sola clase por archivo.
6. Si una clase contenida en un archivo no es ***public***, no es necesario que el archivo 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 archivo (***package packageName;***). Esta agrupación en ***packages*** está relacionada con la jerarquía de directorios y archivos en la que se guardan las clases.

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 alternativa a la herencia múltiple.

En algunos aspectos los nombres de las ***interfaces*** pueden utilizarse en lugar de las ***clases***. Por ejemplo, las ***interfaces*** sirven para definir ***referencias*** a cualquier objeto de cualquiera de las clases que implementan esa ***interface***. Con ese nombre o referencia, sin embargo, sólo se pueden utilizar los métodos de la interface. Éste 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 (a diferencia de las clases, las interfaces de ***Java*** sí tienen herencia múltiple).

2. EJEMPLO DE DEFINICIÓN DE UNA CLASE

A continuación se reproduce como ejemplo la clase ***Circulo.***

```
// archivo 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 se definen las variables miembro y los métodos (cuyos nombres

se han resaltado en negrita) dentro de la clase. Dichas variables y métodos pueden ser ***de objeto*** o ***de clase*** (***static***). Se puede ver también cómo el nombre del archivo coincide con el de la clase ***public*** con la extensión ****.java***.

3. 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 está constituida por unos ***datos*** y unos ***métodos*** que operan sobre esos datos.

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***, el carácter nulo para ***char*** y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas también en el constructor.

Las variables miembro pueden también 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***.

Los ***métodos de objeto*** se aplican a un objeto concreto poniendo el nombre del objeto y luego el nombre del método, separados por un punto. A este objeto se le llama ***argumento implícito***. Por ejemplo, para calcular el área de un objeto de la clase ***Circulo*** llamado ***c1*** se escribirá: ***c1.area();***. Las variables miembro del argumento implícito se acceden directamente o precedidas por la palabra ***this*** y el operador punto.

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. Existen otros dos modificadores (no de acceso) para las variables miembro:

1. ***transient***: indica que esta variable miembro no forma parte de la ***persistencia*** (capacidad de los objetos de mantener su valor cuando termina la ejecución de un programa) de un objeto y por tanto no debe ser ***serializada*** (convertida en flujo de caracteres para poder ser almacenada en disco o en una base de datos) con el resto del objeto.
2. ***volatile***: indica que esta variable puede ser utilizada por distintas ***threads*** sincronizadas y que el compilador no debe realizar optimizaciones con esta variable.

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 para los tipos primitivos (***false*** para ***boolean***, el carácter nulo para ***char*** y cero para los tipos numéricos), 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.

4. 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. Además de las variables miembro, también las variables locales y los propios argumentos de un método pueden ser declarados ***final***.

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

5. MÉTODOS (FUNCIONES MIEMBRO)

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, separados 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.

Si en el ***header*** del método se incluye la palabra ***native*** (Ej: public native void miMetodo();) no hay que incluir el código o implementación del método. Este código deberá estar en una librería dinámica (*Dynamic Link Library* o DLL). Estas librerías son archivos de funciones compiladas normalmente en lenguajes distintos de ***Java*** (C, C++, Fortran, etc.). Es la forma de poder utilizar conjuntamente funciones realizadas en otros lenguajes desde código escrito en ***Java***.

Un método también puede declararse como ***synchronized*** (Ej: public synchronized double miMetodoSynch(){...}). Estos métodos tienen la particularidad de que sobre un objeto no pueden ejecutarse simultáneamente dos métodos que estén sincronizados.

Métodos sobrecargados (overloaded)

Al igual que C++, ***Java*** permite métodos ***sobrecargados*** (***overloaded***), es decir, métodos distintos que tienen ***el mismo nombre***, pero que se diferencian por el número y/o tipo de los argumentos. El ejemplo de la clase ***Circulo*** del Apartado 3.2 presenta dos casos de métodos sobrecargados: los cuatro constructores y los dos métodos llamados ***elMayor()***.

A la hora de llamar a un método sobrecargado, ***Java*** sigue unas reglas para determinar el método concreto que debe llamar:
1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
2. 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.
3. Si sólo existen métodos con argumentos de un tipo más restringido (por ejemplo, *int* en vez de *long*), el programador debe hacer un ***cast*** explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
4. 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***.

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. La forma de modificar dentro de un método una variable de un tipo primitivo es incluirla como variable miembro en una clase y pasar como argumento una referencia a un objeto de dicha clase. 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 punteros a función como argumentos). 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 método1. 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.

Si un método devuelve ***this*** (es decir, un objeto de la clase) o una referencia a otro objeto, ese objeto puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así
1 En ***Java*** no hay variables locales ***static***, que en C/C++ y Visual Basic son variables locales que conservan su valor entre las distintas llamadas a un método. sucesivamente. En este caso aparecerán varios métodos en la misma sentencia unidos por el operador punto (.), por ejemplo,

```
String numeroComoString = ”8.978”;
```

```
float p = Float.valueOf(numeroComoString).floatValue();
```

donde el método ***valueOf(String)*** de la clase ***java.lang.Float*** devuelve un objeto de la clase ***Float*** sobre el que se aplica el método ***floatValue()***, que finalmente devuelve una variable primitiva de tipo ***float***. El ejemplo anterior se podía desdoblar en las siguientes sentencias:

```
String numeroComoString = ”8.978”;
Float f = Float.valueOf(numeroComoString);
float p = f.floatValue();
```

Obsérvese que se pueden encadenar varias llamadas a métodos por medio del operador punto (.) que, como todos los operadores de ***Java*** excepto los de asignación, se ejecuta de izquierda a derecha.

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.

Constructores

Un punto clave de la *Programación Orientada Objetos* es el evitar información incorrecta por no haber sido correctamente inicializadas las variables. ***Java*** no permite que haya variables miembro que no estén inicializadas2. Ya se ha dicho que ***Java*** inicializa siempre con ***valores por defecto*** las variables miembro de clases y objetos. El segundo paso en la inicialización correcta de objetos es el uso de ***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*** 2 Sí puede haber variables locales de métodos sin inicializar, pero el compilador da un error si se intentan utilizar sin asignarles previamente un valor.

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, y los ***Strings*** y las demás ***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.

Inicializadores

Por motivos que se verán más adelante, ***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***.

Inicializadores static
Un ***inicializador static*** es un algo parecido a un método (un bloque {…} de código, sin nombre y sin argumentos, precedido por la palabra ***static***) que se llama automáticamente al crear la clase (al utilizarla por primera vez). También se diferencia del ***constructor*** en que no es llamado para cada objeto, sino una sola vez para toda la clase.

Los tipos primitivos pueden inicializarse directamente con asignaciones en la clase o en el constructor, pero para inicializar objetos o elementos más complicados es bueno utilizar un ***inicializador*** (un bloque de código {…}), ya que permite gestionar ***excepciones3*** con ***try***…***catch***. Los ***inicializadores static*** se crean dentro de la clase, como métodos sin nombre, sin argumentos 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");
}
```

Inicializadores de objeto
A partir de ***Java 1.1*** existen también ***inicializadores de objeto***, que no llevan la palabra ***static***. Se utilizan para las ***clases anónimas***, que por no tener nombre no pueden tener constructor. En este caso, los inicializadores de objeto se llaman cada vez que se crea un objeto de la clase anónima.

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

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*** (recolección 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 críticas.
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.

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 archivos, 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 utilizando 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, en Java 1.1 hay que llamar primero a ***gc()*** y luego a ***runFinalization()***.

6. PACKAGES

Qué es un package
Un ***package*** es una agrupación de clases. En la API de ***Java 1.1*** había 22 ***packages***; en ***Java 1.2*** hay 59 ***packages***, lo que da una idea del "crecimiento" experimentado por el lenguaje. 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 archivo 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 ***java.awt.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:

```
co.ucc.cursos.java.ordenar
```

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

```
co.ucc.cursos.java.ordenar.QuickSort.class
```

debería estar en el directorio,

CLASSPATH\co\ucc\cursos\java\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 **co** en los discos locales del equipo.

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.

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 archivos ****.class*** y ****.java***, con objeto por ejemplo de no mostrar la situación del código fuente. Los ***packages*** hacen referencia a los archivos 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 co.ucc.cursos.java.ordenar.QuickSort.class;
import co.ucc.cursos.java.ordenar.*;
```

que deberían estar en el directorio:

```
classpath\co\ucc\cursos\java\ordenar
```

7. HERENCIA

Concepto de 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***).

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 ya se ha visto al hablar de los ***finalizadores***.
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.

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. Las 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***), pero nunca restringirlos. Los ***métodos de clase*** o ***static*** no pueden ser redefinidos en las clases derivadas.

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***.

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 sus valores por defecto, y los ***Strings*** y demás ***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.

8. CLASES Y MÉTODOS FINALES

Recuérdese que las ***variables*** declaradas como ***final*** no pueden cambiar su valor una vez que han sido inicializadas. En este apartado se van a presentar otros dos usos de la palabra ***final***. 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.

9. INTERFACES

Concepto de interface

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, inpendientemente 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***.

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.***

```
 // archivo 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 archivo ****.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.

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.

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.

10. CLASES INTERNAS

Una ***clase interna*** es una clase definida dentro de otra clase, llamada ***clase contenedora***, en alguna variante de la siguiente forma general:

```
class ClaseContenedora {
      ...
      class ClaseInterna {
            ...
      }
      ...
}
```

Las ***clases internas*** fueron introducidas en la versión ***Java 1.1***. Además de su utilidad en sí, las ***clases internas*** se utilizan mucho en el nuevo ***modelo de eventos*** que se introdujo en dicha versión de ***Java***. Hay cuatro tipos de ***clases internas***:

1. Clases internas *static*.
2. Clases internas *miembro*.
3. Clases internas *locales*.
4. Clases *anónimas*.

En lo sucesivo se utilizará la terminología ***clase contenedora*** o ***clase global*** para hacer referencia a la clase que contiene a la ***clase interna***. Hay que señalar que la JVM (***Java Virtual Machine***) no sabe nada de la existencia de ***clases internas***. Por ello, el compilador convierte estas clases en ***clases globales***, contenidas en archivos ****.class*** cuyo nombre es ***ClaseContenedora$ClaseInterna.class***. Esta conversión inserta variables ocultas, métodos y argumentos en los constructores. De todas formas, lo que más afecta al programador de todo esto es lo referente al nombre de los archivos que aparecen en el directorio donde se realiza la compilación, que pueden resultar sorprendentes si no se conoce su origen.

Clases e interfaces internas static

Se conocen también con el nombre de ***clases anidadas*** (*nested classes*). Las clases e interfaces internas static sólo pueden ser creadas dentro de otra clase ***al máximo nivel***, es decir directamente en el bloque de definición de la ***clase contenedora*** y no en un bloque más interno. Es posible definir ***clases e interfaces internas static*** dentro de una ***interface contenedora***. Este tipo de clases internas se definen utilizando la palabra ***static***. Todas las ***interfaces internas*** son implícitamente ***static***. En cierta forma, las ***clases internas static*** se comportan como clases normales en un ***package***. Para utilizar su nombre desde fuera de la clase contenedora hay que precederlo por el ***nombre de la clase contenedora*** y el ***operador punto*** (.) Este tipo de relación entre clases se puede utilizar para agrupar varias clases dentro de una clase más general. Lo mismo puede decirse de las ***interfaces internas***.

Las ***clases internas static*** pueden ver y utilizar los miembros ***static*** de la ***clase contenedora***. No se necesitan objetos de la clase contenedora para crear objetos de la ***clase interna static***. Los métodos de la ***clase interna static*** no pueden acceder directamente a los objetos de la clase contenedora, caso de que los haya: deben disponer de una referencia a dichos objetos, como
cualquier otra clase.

La sentencia ***import*** puede utilizarse para importar una ***clase interna static***, en la misma forma que si se tratara de importar una clase de un ***package*** (con el punto (.)). Por ejemplo, si la interface ***Linkable*** es interna a la clase ***List***, para implementar dicha interface hay que escribir:

```
implements List.Linkable
```

y para importarla hay que usar,

```
import List.*; // o bien
import List.Linkable;
```

Otras características importantes son las siguientes:
1. Pueden definirse ***clases*** e ***interfaces internas*** dentro de ***interface*** y ***clases contenedoras***, con las cuatro combinaciones posibles.
2. Puede haber ***varios niveles***, esto es una ***clase interna static*** puede ser ***clase contenedora*** de otra ***clase interna static***, y así sucesivamente.
3. Las ***clases e interfaces internas static*** pertenecen al ***package*** de la ***clase contenedora***.
4. Pueden utilizarse los calificadores ***final***, ***public***, ***private*** y ***protected***. Ésta es una forma más de controlar el acceso a ciertas clases.

A continuación se presenta un ejemplo de ***clase interna static***:

```
// archivo ClasesIntStatic.java
class A{
      int i=1; // variable miembro de objeto
      static int is=-1; // variable miembro de clase
      public A(int i) {this.i=i;} // constructor
      // a los métodos de la clase contenedora hay que pasarles referencias
      // a los objetos de la clase interna static
      public void printA(Bs unBs) {
            System.out.println("i="+i+" unBs.j="+unBs.j);
      }
      // definición de una clase interna static
      static class Bs {
            int j=2;
```

```
            public Bs(int j) {this.j=j;} // constructor
            // los métodos de la clase interna static no pueden acceder a la i
            // pues es una variable de objeto. Sí pueden acceder a is
            public void printBs() {
                  System.out.println(" j=" + j + " is=" + is);
            }
      } // fin clase Bs
} // fin clase contenedora A

class ClasesIntStatic {
      public static void main(String [] arg) {
            A a1 = new A(11), a2 = new A(12);
            println("a1.i=" + a1.i + " a2.i=" + a2.i);
            // dos formas de crear objetos de la clase interna static
            A.Bs b1 = new A.Bs(-10); // necesario poner A.Bs
            A.Bs b2 = a1.new Bs(-11); // b2 es independiente de a1
            // referencia directa a los objetos b1 y b2
            println("b1.j=" + b1.j + " b2.j=" + b2.j);
            // los métodos de la clase interna acceden directamente a las variables
            // de la clase contenedora sólo si son static
            b1.printBs(); // escribe: j=-10 is=-1
            b2.printBs(); // escribe: j=-20 is=-1
            // a los métodos de la clase contenedora hay que pasarles referencias
            // a los objetos de la clase interna, pera que puedan identificarlos
            a1.printA(b1); // escribe: i=11 unBs.j=-10
            a1.printA(b2); // escribe: i=11 unBs.j=-11
      } // fin de main()

      public static void println(String str) {
            System.out.println(str);
      }
} // fin clase ClasesIntStatic
```

Clases internas miembro (no static)

Las ***clases internas miembro*** o simplemente ***clases internas***, son clases definidas al máximo nivel de la ***clase contenedora*** (directamente en el bloque de diminución de dicha clase), sin la palabra ***static***. Se suelen llamar ***clases internas miembro*** o simplemente ***clases internas***. No existen ***interfaces internas*** de este tipo.

Las ***clases internas*** no pueden tener ***variables miembro static***. Tienen una nueva sintaxis para las palabras ***this***, ***new*** y ***super***, que se verá un poco más adelante. La característica principal de estas clases internas e que cada objeto de la ***clase interna*** existe siempre dentro de un y sólo un objeto de la ***clase contenedora***. Un objeto de la ***clase contenedora*** puede estar relacionado con uno o más objetos de la ***clase interna***. Tener esto presente es muy importante para entender las características que se explican a continuación. Relación entre las ***clases interna*** y ***contenedora*** respecto al ***acceso a las variables miembro***:

1. Debido a la relación uno a uno, los métodos de la clase interna ven directamente las variables miembro del objeto de la clase contenedora, sin necesidad de cualificarlos.
2. Sin embargo, los métodos de la clase contenedora no ven directamente las variables miembro de los objetos de la clase interna: necesitan cualificarlos con una referencia a los correspondientes objetos. Esto es consecuencia de la relación uno a varios que existe entre los objetos de la clase contenedora y los de la clase interna.
3. ***Otras clases diferentes*** de las ***clases contenedora*** e ***interna*** pueden utilizar directamente los ***objetos de la clase interna***, sin cualificarlos con el ***objeto o el nombre de la clase contenedora***. De hecho, se puede seguir accediendo a los objetos de la clase interna aunque se pierda la referencia al objeto de la clase contenedora con el que están asociados.

Respecto a los ***permisos de acceso***:

1. Las ***clases internas*** pueden también ser ***private*** y ***protected*** (las clases normales sólo pueden ser ***public*** y ***package***). Esto permite nuevas posibilidades de Encapsulamiento.
2. Los métodos de las ***clases internas*** acceden directamente a todos los miembros, incluso ***private***, de la ***clase contenedora***.

3. También la ***clase contenedora*** puede acceder –si dispone de una referencia- a todas las variables miembro (incluso ***private***) de sus ***clases internas***.
4. Una ***clase interna*** puede acceder también a los miembros (incluso ***private***) de otras ***clases internas*** definidas en la misma ***clase contenedora***.

Otras características de las ***clases internas*** son las siguientes:
1. Una ***clase interna miembro*** puede contener otra ***clase interna miembro***, hasta el nivel que se desee (aunque no se considera buena técnica de programación utilizar muchos niveles).
2. En la ***clase interna***, la palabra ***this*** se refiere al objeto de la propia ***clase interna***. Para acceder al objeto de la ***clase contenedora*** se utiliza ***ClaseContenedora.this***.
3. Para crear un nuevo objeto de la ***clase interna*** se puede utilizar ***new***, precedido por la referencia al objeto de la ***clase contenedora*** que contendrá el nuevo objeto: ***unObjCC.new()***. El ***tipo del objeto*** es el nombre de la clase contenedora seguido del nombre de la clase interna, como por ejemplo:

```
ClaseCont.ClaseInt unObjClInt = unObjClaCont.new ClaseInt(...);
```

4. Supóngase como ejemplo adicional que B es una clase interna de A y que C es una clase interna de B. La creación de objetos de las tres clases se puede hacer del siguiente modo:

```
A a = new A(); // se crea un objeto de la clase A
A.B b = a.new B(); // b es un objeto de la clase interna B dentro de a
A.B.C c = b.new C(); // c es un objeto de la clase interna C dentro de b
```

5. Nunca se puede crear un objeto de la ***clase interna*** sin una referencia a un objeto de la ***clase contenedora***. Los ***constructores*** de la ***clase interna*** tienen como argumento oculto una referencia al objeto de la ***clase contenedora***.
6. El nuevo significado de la palabra ***super*** es un poco complicado: Si una clase deriva de una ***clase interna***, su constructor no puede llamar a ***super()*** directamente. Ello hace que el compilador no pueda crear un ***constructor por defecto***. Al constructor hay que pasarle una referencia a la ***clase contenedora*** de la ***clase interna super-clase***, y con esa referencia ***ref*** llamar a ***ref.super()***.

Las ***clases internas*** pueden ***derivar*** de otras clases diferentes de la ***clase contenedora***. En este caso, conviene tener en cuenta las siguientes reglas:
1. Las ***clases internas*** constituyen como una ***segunda jerarquía de clases*** en ***Java***: por una parte están en la ***clases contenedora*** y ven sus variables; por otra parte ***pueden derivar de otra clase*** que no tenga nada que ver con la ***clase contenedora***. Es muy importante evitar conflictos con los nombres. En caso de conflicto entre un nombre heredado y un nombre en la clase contenedora, el nombre heredado debe tener prioridad.
2. En caso de conflicto de nombres, ***Java*** obliga a utilizar la referencia ***this*** con un nuevo significado: para referirse a la ***variable o método miembro heredado*** se utiliza ***this.name***, mientras que se utiliza ***NombreClaseCont.this.name*** para el ***miembro de la clase contenedora***..
3. Si una ***clase contenedora*** deriva de una ***super-clase*** que tiene una ***clase interna***, la ***clase interna de la sub-clase*** puede a su vez derivar de la ***clase interna de la super-clase*** y redefinir todos los métodos que necesite. La casuística se puede complicar todo lo que se desee, pero siempre hay que recomendar hacer las cosas lo más sencillas que sea posible.

El uso de las ***clases internas miembro*** tiene las siguientes restricciones:
1. Las ***clases internas*** no pueden tener el mismo nombre que la ***clase contenedora*** o ***package***.
2. Tampoco pueden tener miembros ***static***: variables, métodos o clases. A continuación se presenta un ejemplo completo de utilización de ***clases internas miembro***:

```
// archivo ClasesInternas.java
// clase contenedora
class A {
      int i=1; // variable miembro
      public A(int i) {this.i=i;} // constructor
      // los métoodos de la clase contenedora necesitan una
      // referencia a los objetos de la clase interna
      public void printA(B unB) {
            System.out.println("i="+i+" unB.j="+unB.j); // sí acepta unB.j
      }
```

```
      // la clase interna puede tener cualquier visibilidad. Con private da error
      // porque main() no puede acceder a la clase interna
      protected class B {
            int j=2;
            public B(int j) {this.j=j;} // constructor
            public void printB() {
                  System.out.println("i=" + i + " j=" + j); // sí sabe qué es j
            }
      } // fin clase B
} // fin clase contenedora A

class ClasesInternas {
      public static void main(String [] arg) {
            A a1 = new A(11); A a2 = new A(12);
            println("a1.i=" + a1.i + " a2.i=" + a2.i);
            // forma de crear objetos de la clase interna
            // asociados a un objeto de la clase contenedora
            A.B b1 = a1.new B(-10), b2 = a1.new B(-20);
            // referencia directa a los objetos b1 y b2 (sin cualificar).
            println("b1.j=" + b1.j + " b2.j=" + b2.j);
            // los métodos de la clase interna pueden acceder directamente a
            // las variables miembro del objeto de la clase contenedora
            b1.printB(); // escribe: i=11 j=-10
            b2.printB(); // escribe: i=11 j=-20
            // los métodos de la clase contenedora deben recibir referencias
            // a los objetos de la clase interna, pera que puedan identificarlos
            a1.printA(b1); a1.printA(b2);
            A a3 = new A(13);
            A.B b3 = a3.new B(-30);
            println("b3.j=" + b3.j);
            a3 = null; // se destruye la referencia al objeto de la clase contenedora
            b3.printB(); // escribe: i=13 j=-30
            a3 = new A(14); // se crea un nuevo objeto asociado a la referencia a3
            // b3 sigue asociado an anterior objeto de la clase contenedora
            b3.printB(); // escribe: i=13 j=-30
      } // fin de main()
      public static void println(String str) {System.out.println(str);}
} // fin clase ClasesInternas
```

Clases internas locales

Las ***clases internas locales*** o simplemente ***clases locales*** no se declaran dentro de otra clase al máximo nivel, sino dentro de un ***bloque de código***, normalmente en un ***método***, aunque también se pueden crear en un ***inicializador static*** o de objeto. Las principales características de las ***clases locales*** so las siguientes:

1. Como las ***variables locales***, las ***clases locales*** sólo son visibles y utilizables en el bloque de código en el que están definidas. Los objetos de la ***clase local*** deben ser creados en el mismo bloque en que dicha clase ha sido definida. De esta forma se puede acercar la definición al uso de la clase.
2. Las ***clases internas locales*** tienen acceso a todas las variables miembro y métodos de la ***clase contenedora***. Pueden ver también los ***miembros heredados***, tanto por la ***clase interna local*** como por la ***clase contenedora***.
3. Las ***clases locales*** pueden utilizar las variables locales y argumentos de métodos ***visibles en ese bloque de código***, pero sólo si son ***final5*** (en realidad la ***clase local*** trabaja con sus copias de las ***variables locales*** y por eso se exige que sean ***final*** y no puedan cambiar).
4. Un objeto de una ***clase interna local*** sólo puede existir en relación con un objeto de la ***clase contenedora***, que debe existir previamente.
5. La palabra ***this*** se puede utilizar en la misma forma que en las ***clases internas*** miembro, pero no las palabras ***new*** y ***super***.

Restricciones en el uso de las ***clases internas locales***:

1. No pueden tener el mismo nombre que ninguna de sus clases contenedoras.
2. No pueden definir variables, métodos y clases ***static***.

3. No pueden ser declaradas ***public***, ***protected***, ***private*** o ***package***, pues su visibilidad es siempre la de las variables locales, es decir, la del bloque en que han sido definidas.

Las ***clases internas locales*** se utilizan para definir clases ***Adapter*** en el AWT. A continuación se presenta un ejemplo de definición de clases internas locales:

```
// archivo ClasesIntLocales.java
// Este archivo demuestra cómo se crean clases locales
class A {
      int i=-1; // variable miembro
      // constructor
      public A(int i) {this.i=i;}
      // definición de un método de la clase A
      public void getAi(final long k) { // argumento final
            final double f=3.14; // variable local final
            // definición de una clase interna local
            class BL {
                  int j=2;
                  public BL(int j) {this.j=j;} // constructor
                  public void printBL() {
                        System.out.println(" j="+j+" i="+i+" f="+f+" k="+k);
                  }
            } // fin clase BL
            // se crea un objeto de BL
            BL bl = new BL(2*i);
            // se imprimen los datos de ese objeto
            bl.printBL();
      } // fin getAi
} // fin clase contenedora A

class ClasesIntLocales {
      public static void main(String [] arg) {
            // se crea dos objetos de la clase contenedora
            A a1 = new A(-10);
            A a2 = new A(-11);
            // se llama al método getAi()
            a1.getAi(1000); // se crea y accede a un objeto de la clase local
            a2.getAi(2000);
      } // fin de main()
      public static void println(String str) {System.out.println(str);}
} // fin clase ClasesIntLocales
```

Clases anónimas

Las ***clases anónimas*** son muy similares a las ***clases internas locales***, pero ***sin nombre***. En las ***clases internas locales*** primero se define la clase y luego se crean uno o más objetos. En las ***clases anónimas*** se unen estos dos pasos: Como la clase no tiene nombre sólo se puede crear un único objeto, ya que las ***clases anónimas*** no pueden definir ***constructores***. Las clases anónimas se utilizan con mucha frecuencia en el AWT para definir clases y objetos que gestionen los eventos de los distintos componentes de la interface de usuario. No hay ***interfaces anónimas***.

Formas de definir una ***clase anónima***:

1. Las ***clases anónimas*** requieren una extensión de la palabra clave ***new***. Se definen en una expresión de ***Java***, incluida en una asignación o en la llamada a un método. Se incluye la palabra ***new*** seguida de la ***definición de la clase anónima***, entre llaves {…}.
2. Otra forma de definirlas es mediante la palabra ***new*** seguida del nombre de la clase de la que hereda (sin ***extends***) y la definición de la ***clase anónima*** entre llaves {…}. El nombre de la ***super-clase*** puede ir seguido de argumentos para su ***constructor*** (entre paréntesis, que con mucha frecuencia estarán vacíos pues se utilizará un constructor por defecto).
3. Una tercera forma de definirlas es con la palabra ***new*** seguida del nombre de la ***interface*** que implementa (sin ***implements***) y la definición de la ***clase anónima*** entre llaves {…}. En este caso la ***clase anónima*** deriva de ***Object***. El nombre de la ***interface*** va seguido por paréntesis vacíos, pues el constructor de ***Object*** no tiene

argumentos. Para las ***clases anónimas compiladas*** el compilador produce archivos con un nombre del tipo ***ClaseContenedora$1.class***, asignando un número correlativo a cada una de las ***clases anónimas***.

Conviene ser muy cuidadoso respecto a los aspectos tipográficos de la definición de ***clases anónimas***, pues al no tener nombre dichas clases suelen resultar difíciles de leer e interpretar. Se aconseja utilizar las siguientes normas ***tipográficas***:
1. Se aconseja que la palabra ***new*** esté en la misma línea que el resto de la expresión.
2. Las ***llaves*** se abren en la misma línea que ***new***, después del cierre del paréntesis de los argumentos del constructor.
3. El ***cuerpo de la clase anónima*** se debe sangrar o indentar respecto a las líneas anteriores de código para que resulte claramente distinguible.
4. El ***cierre de las llaves*** va seguido por el ***resto de la expresión*** en la que se ha definido la ***clase anónima***. Esto puede servir como indicación tipográfica del cierre. Puede ser algo así como **};** o **});**

A continuación se presenta un ejemplo de definición de clase anónima en relación con el AWT:

```
unObjeto.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            ...
      }
});
```

donde en negrita se señala la ***clase anónima***, que deriva de ***Object*** e implementa la interface ***ActionListener***. Las ***clases anónimas*** se utilizan en lugar de ***clases locales*** para clases con muy poco código, de las que sólo hace falta un objeto. No pueden tener ***constructores***, pero sí ***inicializadores static*** o de ***objeto***. Además de las restricciones citadas, tienen ***restricciones*** similares a las ***clases locales***.

11. PERMISOS DE ACCESO EN JAVA

Una de las características de la *Programación Orientada a Objetos* es la ***Encapsulamiento***, 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.

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 archivos. En este sentido, un ***package*** es accesible si sus directorios y archivos 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.

Accesibilidad de clases o interfaces

En principio, cualquier ***clase*** o ***interface*** de un ***package*** es accesible para todas las demás clases el ***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).

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 Tabla muestra un resumen de los permisos de acceso en ***Java***.

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

12. 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.

Conversión de tipos primitivos

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 (más amplio) 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***.

13. POLIMORFISMO

Ya se vio en el ejemplo presentado en el Apartado 1.3.8 y en los comentarios incluidos en qué consistía el ***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*** (en tiempo de compilación) o ***tardía*** (en tiempo de ejecución). Con funciones normales o sobrecargadas se utiliza vinculación temprana (es posible y es lo más eficiente). Con funciones redefinidas en ***Java*** se utiliza siempre ***vinculación tardía***, excepto si el método es ***final***. El ***polimorfismo*** es la ***opción por defecto*** en ***Java***.

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*** permite a los programadores separar las cosas que cambian de las que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas. El ***polimorfismo*** puede hacerse con referencias de ***super-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.

Conversión de objetos

El ***polimorfismo*** visto previamente está basado en utilizar referencias de un tipo más "amplio" que los objetos a los que apuntan. Las ventajas del polimorfismo son evidentes, pero hay una importante limitación: el tipo de la referencia (clase abstracta, clase base o interface) limita los métodos que se pueden utilizar y las variables miembro a las que se pueden acceder. Por ejemplo, un objeto puede tener una referencia cuyo tipo sea una ***interface***, aunque sólo en el caso en que su ***clase*** o ***una de sus super-clases*** implemente dicha ***interface***. Un objeto cuya referencia es un tipo ***interface*** sólo puede utilizar los métodos definidos en dicha ***interface***. Dicho de otro modo, ese objeto no puede utilizar las variables y los métodos propios de su clase. De esta forma las ***referencias de tipo interface*** definen, limitan y unifican la forma de utilizarse de objetos pertenecientes a clases muy distintas (que implementan dicha interface).

Si se desea utilizar todos los métodos y acceder a todas las variables que la clase de un objeto permite, hay que utilizar un ***cast explícito***, que convierta su referencia más general en la del tipo específico del objeto. De aquí una parte importante del interés del ***cast*** entre objetos (más bien entre referencias, habría que decir).

Para la conversión entre objetos de distintas clases, ***Java*** exige que dichas clases estén relacionadas por ***herencia*** (una deberá ser ***sub-clase*** de la otra). Se realiza una ***conversión implícita*** o ***automática*** de una ***sub-clase*** a una ***super-clase*** siempre que se necesite, ya que el objeto de la ***sub-clase*** siempre tiene toda la información necesaria para ser utilizado en lugar de un objeto de la ***super-clase***. No importa que la ***super-clase*** no sea capaz de contener toda la información de la ***subclase***.

La conversión en sentido contrario -utilizar un objeto de una ***super-clase*** donde se espera encontrar uno de la ***sub-clase***- debe hacerse de modo ***explícito*** y puede producir errores por falta de información o de métodos. Si falta información, se obtiene una ***ClassCastException***. *No se puede acceder a las variables exclusivas de la sub-clase a través de una referencia de la super-clase. Sólo se pueden utilizar los métodos definidos en la super-clase, aunque la definición utilizada para dichos métodos sea la de la sub-clase.*

Por ejemplo, supóngase que se crea un objeto de una ***sub-clase B*** y se referencia con un nombre de una ***super-clase A***,

```
A a = new B();
```

en este caso el objeto creado dispone de más información de la que la referencia ***a*** le permite acceder (podría ser, por ejemplo, una nueva variable miembro ***j*** declarada en ***B***). Para acceder a esta información adicional hay que hacer un ***cast*** explícito en la forma ***(B)a***. Para imprimir esa variable ***j*** habría que escribir (los paréntesis son necesarios):

```
System.out.println( ((B)a).j );
```

Un ***cast*** de un objeto a la ***super-clase*** puede permitir utilizar variables -no métodos- de la ***super-clase***, aunque estén redefinidos en la ***sub-clase***. Considérese el siguiente ejemplo: La clase ***C*** deriva de ***B*** y ***B*** deriva de ***A***. Las tres definen una variable ***x***. En este caso, si desde el código de la sub-clase ***C*** se utiliza:

```
x // se accede a la x de C
this.x // se accede a la x de C
super.x // se accede a la x de B. Sólo se puede subir un nivel
((B)this).x // se accede a la x de B
((A)this).x // se accede a la x de A
```