Java desde Cero

Con ésta comienzo una serie de notas sobre Java, especialmente para aquellos que quieren comenzar a conocerlo y usarlo. Esto se originó en un interés que surgió en algunos de los suscriptores del mailing list de desarrolladores de web, y que pongo a disposición también del de webmasters.
Seguramente muchos de ustedes sabrán mucho más sobre Java que yo, y les agradeceré todo tipo de comentarios o correcciones.
La idea es dar una guía ordenada para el estudio de este lenguaje, muy poderoso y de gran coherencia, aunque todavía adolece de algunas limitaciones que seguramente se irán superando con el tiempo.

Qué es Java

Java es un lenguaje originalmente desarrollado por un grupo de ingenieros de Sun, utilizado por Netscape posteriormente como base para Javascript. Si bien su uso se destaca en el Web, sirve para crear todo tipo de aplicaciones (locales, intranet o internet).
Java es un lenguaje:

- de objetos
- independiente de la plataforma

Algunas características notables:

- robusto
- gestiona la memoria automáticamente
- no permite el uso de técnicas de programación inadecuadas
- multithreading
- cliente-servidor
- mecanismos de seguridad incorporados
- herramientas de documentación incorporadas

Lenguaje de Objetos

Por qué puse "de" objetos y no "orientado a" objetos? Para destacar que, al contrario de otros lenguajes como C++, no es un lenguaje modificado para poder trabajar con objetos sino que es un lenguaje creado para trabajar con objetos desde cero. De hecho, TODO lo que hay en Java son objetos.

Qué es un objeto?

Bueno, se puede decir que todo puede verse como un objeto. Pero seamos más claros. Un objeto, desde nuestro punto de vista, puede verse como una pieza de software que cumple con ciertas características:

- encapsulamiento
- herencia

Encapsulamiento significa que el objeto es auto-contenido, o sea que la misma definición del objeto incluye tanto los datos que éste usa (*atributos*) como los procedimientos (*métodos*) que actúan sobre los mismos.
Cuando se utiliza programación orientada a objetos, se definen *clases* (que definen objetos genéricos) y la forma en que los objetos interactúan entre ellos, a través de *mensajes*. Al crear un objeto de una clase dada, se dice que se crea una *instancia* de la clase, o un objeto propiamente dicho. Por ejemplo, una clase podría ser "autos", y un auto dado es una *instancia* de la clase.
La ventaja de esto es que como no hay programas que actúen modificando al objeto, éste se mantiene en cierto modo independiente del resto de la aplicación. Si es necesario modificar el objeto (por ejemplo, para darle más capacidades), esto se puede hacer sin tocar el resto de la aplicación… lo que ahorra mucho tiempo

de desarrollo y debugging! En Java, inclusive, *ni siquiera existen las variables globales*! (Aunque parezca difícil de aceptar, esto es una gran ventaja desde el punto de vista del desarrollo).
En cuanto a la herencia, simplemente significa que se pueden crear nuevas clases que hereden de otras preexistentes; esto simplifica la programación, porque las clases hijas incorporan automáticamente los métodos de las madres. Por ejemplo, nuestra clase "auto" podría heredar de otra más general, "vehículo", y simplemente redefinir los métodos para el caso particular de los automóviles… lo que significa que, con una buena biblioteca de clases, se puede reutilizar mucho código inclusive sin saber lo que tiene adentro.

Un ejemplo simple

Para ir teniendo una idea, vamos a poner un ejemplo de una clase Java:

```
public class Muestra extends Frame {
// atributos de la clase
Button si;
Button no;
// métodos de la clase:
public Muestra () {
Label comentario = new Label("Presione un botón", Label.CENTER);
si = new Button("Sí");
no = new Button("No");
add("North", comentario);
add("East", si);
add("West", no);
}
}
```

Esta clase no está muy completa así, pero da una idea… Es una clase heredera de la clase *Frame* (un tipo de ventana) que tiene un par de botones y un texto. Contiene dos atributos ("si" y "no"), que son dos objetos del tipo *Button*, y un único método llamado Muestra (igual que la clase, por lo que es lo que se llama un *constructor*).

Independiente de la plataforma

Esto es casi del todo cierto…
En realidad, Java podría hacerse correr hasta sobre una Commodore 64! La realidad es que para utilizarlo en todo su potencial, requiere un sistema operativo multithreading (como Unix, Windows95, OS/2…).
Cómo es esto? Porque en realidad Java es un lenguaje interpretado… al menos en principio.
Al compilar un programa Java, lo que se genera es un seudocódigo definido por Sun, para una máquina genérica. Luego, al correr sobre una máquina dada, el software de ejecución Java simplemente interpreta las instrucciones, emulando a dicha máquina genérica. Por supuesto esto no es muy eficiente, por lo que tanto Netscape como Hotjava o Explorer, al ejecutar el código por primera vez, lo van compilando (mediante un *JIT*: *J*ust *I*n *T*ime compiler), de modo que al crear por ejemplo la segunda instancia de un objeto el código ya esté compilado específicamente para la máquina huésped.
Además, Sun e Intel se han puesto de acuerdo para desarrollar procesadores que trabajen directamente en Java, con lo que planean hacer máquinas muy baratas que puedan conectarse a la red y ejecutar aplicaciones Java cliente-servidor a muy bajo costo.
El lenguaje de dicha máquina genérica es público, y si uno quisiera hacer un intérprete Java para una Commodore sólo tendría que implementarlo y pedirle a Sun la aprobación (para que verifique que cumple con los requisitos de Java en cuanto a cómo interpreta cada instrucción, la seguridad, etc.)

Algunas características…

Entre las características que nombramos nos referimos a la robustez. Justamente por la forma en que está diseñado, Java no permite el manejo directo del hardware ni de la memoria (inclusive no permite modificar

valores de punteros, por ejemplo); de modo que se puede decir que es virtualmente imposible colgar un programa Java. El intérprete siempre tiene el control.
Inclusive el compilador es suficientemente inteligente como para no permitir un montón de cosas que podrían traer problemas, como usar variables sin inicializarlas, modificar valores de punteros directamente, acceder a métodos o variables en forma incorrecta, utilizar herencia múltiple, etc.
Además, Java implementa mecanismos de seguridad que limitan el acceso a recursos de las máquinas donde se ejecuta, especialmente en el caso de los Applets (que son aplicaciones que se cargan desde un servidor y se ejecutan en el cliente).
También está diseñado específicamente para trabajar sobre una red, de modo que incorpora objetos que permiten acceder a archivos en forma remota (via URL por ejemplo).
Además, con el JDK (Java Development Kit) vienen incorporadas muchas herramientas, entre ellas un generador automático de documentación que, con un poco de atención al poner los comentarios en las clases, crea inclusive toda la documentación de las mismas en formato HTML!

El Java Development Kit

Todo lo que puedan pedir para desarrollar aplicaciones en Java está en:

- http://java.sun.com/aboutJava/index.html

En particular, deberían bajarse el JDK y el API Documentation de:

- http://java.sun.com/java.sun.com/products/JDK/1.0.2/index.html

(También les puede interesar en particular el Tool Documentation y alguno de los otros paquetes de la página)
Nota: en este site también hay un tutorial de Java, aunque es un poco difícil de seguir para el principiante.
El JDK (versión 1.0.2) está disponible para SPARC/Solaris, x86/Solaris, MS-Windows 95/NT, y MacOS.
También está disponible el fuente para el que quiera adaptarlo para otro sistema operativo, y he leído por ahí que hay una versión dando vueltas para Linux y HP-UX.
Básicamente, el JDK consiste de:

- el compilador Java, javac
- el intérprete Java, java
- un visualizador de applets, appletviewer
- el debugger Java, jdb (que para trabajar necesita conectarse al server de Sun)
- el generador de documentación, javadoc

También se puede bajar del mismo site un browser que soporta Java (y de hecho está escrito *totalmente* en Java), el Hotjava.
Para instalarlo simplemente hay que descompactar el archivo (sugiero que creen un directorio java para eso), pero tengan en cuenta NO DESCOMPRIMIR el archivo classes.zip!
Importante para los usuarios de Windows95: todas estas aplicaciones deben ejecutarse desde una ventana DOS. En particular, utilizan nombres largos y distinguen mayúsculas de minúsculas, así que tengan en cuenta esto que es fuente de muchos errores.
Una cosa muy importante: para que todo ande bien aceitado, agreguen:

- el directorio de los programas en el path (ej: c:\java\bin)
- las variables de entorno:
    - CLASSPATH=.;C:\java\lib\classes.zip
    - HOMEDRIVE=C:
    - HOMEPATH=\
    - HOME=C:\

con los valores adecuados a su entorno.
Noten que en CLASSPATH agregué el directorio actual (.), para poder compilar y ejecutar desde cualquier directorio.

Empecemos de una vez!

Bueno, suponiendo que hayan instalado todo, y antes de comenzar a programar en Java, una pequeña aclaración :

En realidad se puede decir que hay tres Javas por ahí:

- Javascript: es una versión de Java directamente interpretada, que se incluye como parte de una página HTML, lo que lo hace muy fácil y cómodo para aplicaciones muy pequeñas, pero que en realidad tiene muchas limitaciones:
  - no soporta clases ni herencia
  - no se precompila
  - no es obligatorio declarar las variables
  - verifica las referencias en tiempo de ejecución
  - no tiene protección del código, ya que se baja en ascii
  - no todos los browsers lo soportan completamente; Explorer, por ejemplo, no soporta las últimas adiciones de Netscape, como las imágenes animadas.
- Java standalone: programas Java que se ejecutan directamente mediante el intérprete java.
- Applets: programas Java que corren bajo el entorno de un browser (o del appletviewer)

En sí los dos últimos son el mismo lenguaje, pero cambia un poco la forma en que se implementa el objeto principal (la aplicación). Vamos a ver cómo crear las aplicaciones para que, sin cambios, se puedan ejecutar casi igual en forma standalone o como applet (en realidad hay cosas que los applets no pueden hacer, como acceder a archivos sin autorización).

Javascript

No vamos a detenernos mucho en Javascript, por las limitaciones antedichas; si les interesa podemos dedicarnos un poco a este lenguaje en el futuro. Por ahora, sólo un ejemplo sencillo:
Calculadora en Javascript:

```
<HTML>
<HEAD>
<SCRIPT LANGUAJE="Javascript">
 function calcula(form) {
  if (confirm("¿Está seguro?"))
    form.resultado.value = eval(form.expr.value)
  else
    alert("Vuelva a intentarlo...")
 }
</SCRIPT>
</HEAD>
<BODY>
<FORM>
 Introduzca una expresión:
 <INPUT TYPE="text" NAME="expr" SIZE=15>
 <INPUT TYPE="button" NAME="Boton" VALUE="Calcular" ONCLICK="calcula(this.form)">
 <BR>
 Resultado:
 <INPUT TYPE="text" NAME="resultado" SIZE=15>
 <BR>
</FORM>
</BODY>
</HTML>
```

Básicamente, el código se encuadra entre los tags <SCRIPT>…</SCRIPT>, y los parámetros se pasan al mismo mediante un form (<FORM>…</FORM>). El lenguaje utilizado es muy parecido al C++, y básicamente el código se ejecuta mediante una acción de un botón (…ONCLICK="calcula(this.form)").
Al presionar el botón, se llama a la función **calcula** con el parámetro **this.form**, que se refiere al form al que pertenece el botón.

La función asigna al **valor** del campo **resultado** del **form** que se le pasa como parámetro (*form.resultado.value*) el resultado de **eval**uar el **valor** de la expresión del campo **expr** de dicho **form** (*eval(form.expr.value)*).
Hay MUCHOS ejemploes de Javascript en:

- http://www.c2.net/~andreww/javascript/

incluyendo decenas de calculadoras, juegos y otras yerbas!
Allí también encontrarán la documentación y un tutorial sobre Javascript.

Las clases en Java

Bueno, antes que nada conviene saber que en Java hay un montón de clases ya definidas y utilizables.
Éstas vienen en las bibliotecas estándar:

- java.lang - clases esenciales, números, strings, objetos, compilador, runtime, seguridad y threads (es el único paquete que se incluye automáticamente en todo programa Java)
- java.io - clases que manejan entradas y salidas
- java.util - clases útiles, como estructuras genéricas, manejo de fecha, hora y strings, número aleatorios, etc.
- java.net - clases para soportar redes: URL, TCP, UDP, IP, etc.
- java.awt - clases para manejo de interface gráfica, ventanas, etc.
- java.awt.image - clases para manejo de imágenes
- java.awt.peer - clases que conectan la interface gráfica a implementaciones dependientes de la plataforma (motif, windows)
- java.applet - clases para la creación de applets y recursos para reproducción de audio.

Para que se den una idea, los números enteros, por ejemplo, son "instancias" de una clase no redefinible, *Integer*, que desciende de la clase *Number* e implementa los siguientes atributos y métodos:

```
public final class java.lang.Integer extends java.lang.Number {
    // Atributos
  public final static int MAX_VALUE;
  public final static int MIN_VALUE;
    // Métodos Constructores
  public Integer(int value);
  public Integer(String s);
    // Más Métodos
  public double doubleValue();
  public boolean equals(Object obj);
  public float floatValue();
  public static Integer getInteger(String nm);
  public static Integer getInteger(String nm, int val);
  public static Integer getInteger(String nm, Integer val);
  public int hashCode();
  public int intValue();
  public long longValue();
  public static int parseInt(String s);
  public static int parseInt(String s, int radix);
  public static String toBinaryString(int i);
  public static String toHexString(int i);
  public static String toOctalString(int i);
  public String toString();
  public static String toString(int i);
```

```
   public static String  toString(int  i, int  radix);
   public static Integer valueOf(String  s);
   public static Integer valueOf(String  s, int  radix);
}
```

Mucho, no?
Esto también nos da algunas ideas:

- la estructura de una clase
- caramba, hay métodos repetidos!

De la estructura enseguida hablaremos; en cuanto a los métodos repetidos (como *parseInt* por ejemplo), al llamarse al método el compilador decide cuál de las implementaciones del mismo usar basándose en la cantidad y tipo de parámetros que le pasamos. Por ejemplo, parseInt("134") y parseInt("134",16), al compilarse, generarán llamados a dos métodos distintos.

Estructura de una clase

Una clase consiste en:

```
algunas_palabras class nombre_de_la_clase [algo_más] {
            [lista_de_atributos]
            [lista_de_métodos]
}
```

Lo que está entre [y] es opcional…
Ya veremos qué poner en "algunas_palabras" y "algo_más", por ahora sigamos un poco más.
La lista de atributos (nuestras viejas variables locales) sigue el mismo formato de C: se define primero el tipo y luego el nombre del atributo, y finalmente el ";".

```
public final static int MAX_VALUE
;
```

También tenemos "algunas_palabras" adelante, pero en seguida las analizaremos.
En cuanto a los métodos, también siguen la sintaxis del C; un ejemplo:

```
public int incContador() {          // declaración y apertura de {
   cnt++;                           // instrucciones, separadas por ";"
   return(cnt);
}                                                   // cierre de }
```

Finalmente, se aceptan comentarios entre /* y */, como en C, o bien usando // al principio del comentario (el comentario termina al final de la línea).
Veamos un ejemplo:

```
// Implementación de un contador sencillo
// GRABAR EN UN ARCHIVO     "Contador.java"        (OJO CON LAS MAYUSCULAS!)
// COMPILAR CON:            "javac Contador.java"  (NO OLVIDAR EL .java!)
// ESTA CLASE NO ES UNA APLICACION, pero nos va a servir enseguida

public class Contador {                         // Se define la clase Contador

 // Atributos
 int cnt;                                                // Un entero para guardar el valor actual

 // Constructor                                          // Un método constructor…
 public Contador() {                            // …lleva el mismo nombre que la clase
  cnt = 0;                                      // Simplemente, inicializa (1)
 }

 // Métodos
```

```
 public int incCuenta() {                    // Un método para incrementar el contador
  cnt++;                                     // incrementa cnt
  return cnt;                                // y de paso devuelve el nuevo valor
 }
 public int getCuenta() {                    // Este sólo devuelve el valor actual
  return cnt;                                // del contador
 }
}
```

Cuando, desde una aplicación u otro objeto, se crea una **instancia** de la clase **Contador**, mediante la instrucción:

```
new Contador()
```

el compilador busca un método con el mismo nombre de la clase y que se corresponda con la llamada en cuanto al tipo y número de parámetros. Dicho método se llama Constructor, y una clase puede tener más de un constructor (no así un objeto o instancia, ya que una vez que fue creado no puede recrearse sobre sí mismo).

En tiempo de ejecución, al encontrar dicha instrucción, el intérprete reserva espacio para el objeto/instancia, crea su estructura y llama al constructor.

O sea que el efecto de `new Contador()` es, precisamente, reservar espacio para el contador e inicializarlo en cero.

En cuanto a los otros métodos, se pueden llamar desde otros objetos (lo que incluye a las aplicaciones) del mismo modo que se llama una función desde C.

Por ejemplo, usemos nuestro contador en un programa bien sencillo que nos muestre cómo evoluciona:

```
// Usemos nuestro contador en una mini-aplicación
// GRABAR EN UN ARCHIVO    "Ejemplo1.java"      (OJO CON LAS MAYUSCULAS!)
// COMPILAR CON:           "javac Ejemplo.java" (NO OLVIDAR EL .java!)
// EJECUTAR CON:           "java Ejemplo1"                (SIN el .java)

import java.io.*;                              // Uso la biblioteca de entradas/salidas

public class Ejemplo1 {                        // IMPORTANTE: Nombre de la clase
                                               // igual al nombre del archivo!
 // entero para asignarle el valor del contador e imprimirlo
 // aunque en realidad no me hace falta.
 static int n;
 // y una variable tipo Contador para instanciar el objeto…
 static Contador laCuenta;

 // ESTE METODO, MAIN, ES EL QUE HACE QUE ESTO SE COMPORTE
 // COMO APLICACION. Es donde arranca el programa cuando ejecuto "java Ejemplo1"
 // NOTA: main debe ser public & static.
 public static void main ( String args[] ) {
  System.out.println ("Cuenta… ");             // Imprimo el título
  laCuenta = new Contador();                   // Creo una instancia del Contador
  System.out.println (laCuenta.getCuenta());   // 0 - Imprimo el valor actual (cero!)
  n = laCuenta.incCuenta();                    // 1 - Asignación e incremento
  System.out.println (n);                      // Ahora imprimo n
  laCuenta.incCuenta();                        // 2 - Lo incremento (no uso el valor…
  System.out.println (laCuenta.getCuenta());   // …de retorno) y lo imprimo
  System.out.println (laCuenta.incCuenta());   // 3 - Ahora todo en un paso!
 }
}
```

En el capítulo III vamos a analizar este programa en detalle. Por ahora veamos la diferencia con un applet que haga lo mismo:

```
            // Applet de acción similar a la aplicación Ejemplo1
            // GRABAR EN ARCHIVO:        "Ejemplo2.java"
            // COMPILAR CON:             "javac Ejemplo2.java"
            // PARA EJECUTAR:  Crear una página HTML como se indica luego
import java.applet.*;
import java.awt.*;

public class Ejemplo2 extends Applet {
 static int n;
 static Contador laCuenta;

 // Constructor…
 public Ejemplo2 () {
   laCuenta = new Contador();
 }

 // El método paint se ejecuta cada vez que hay que redibujar
 // NOTAR EL EFECTO DE ESTO CUANDO SE CAMBIA DE TAMAÑO LA
 // VENTANA DEL NAVEGADOR!
 public void paint (Graphics g) {
   g.drawString ("Cuenta...", 20, 20);
   g.drawString (String.valueOf(laCuenta.getCuenta()), 20, 35 );
   n = laCuenta.incCuenta();
   g.drawString (String.valueOf(n), 20, 50 );
   laCuenta.incCuenta();
   g.drawString (String.valueOf(laCuenta.getCuenta()), 20, 65 );
   g.drawString (String.valueOf(laCuenta.incCuenta()), 20, 80 );
 }
}
```

Ahora es necesario crear una página HTML para poder visualizarlo. Para esto, crear y luego cargar el archivo ejemplo2.htm con un browser que soporte Java (o bien ejecutar en la ventana DOS: "appletviewer ejemplo2.htm"):

```
<HTML>
<HEAD>
<TITLE>Ejemplo 2 - Applet Contador</TITLE>
</HEAD>
<BODY>
<applet code="Ejemplo2.class" width=170 height=150>
</applet>
</BODY>
</HTML>
```

Para terminar este capítulo, observemos las diferencias entre la aplicación standalone y el applet:

- La aplicación usa un método main, desde donde arranca
- El applet, en cambio, se arranca desde un constructor (método con el mismo nombre que la clase)

Además:

- En la aplicación utilizamos System.out.println para imprimir en la salida estándar
- En el applet necesitamos "dibujar" el texto sobre un fondo gráfico, por lo que usamos el método g.drawString dentro del método paint (que es llamado cada vez que es necesario redibujar el applet)

Con poco trabajo se pueden combinar ambos casos en un solo objeto, de modo que **la misma** clase sirva para utilizarla de las dos maneras:

```
// Archivo:  Ejemplo3.java
// Compilar con: javac Ejemplo3.java
import java.applet.*;
import java.awt.*;
import java.io.*;

public class Ejemplo3 extends Applet {
  static int n;
  static Contador laCuenta;

  public Ejemplo3 () {
    laCuenta = new Contador();
  }

  public static void main(String args[]) {
    laCuenta = new Contador();
    paint();
  }

  public static void paint () {
    System.out.println ("Cuenta...");
    System.out.println (laCuenta.getCuenta());
    n = laCuenta.incCuenta();
    System.out.println (n);
    laCuenta.incCuenta();
    System.out.println (laCuenta.getCuenta());
    System.out.println (laCuenta.incCuenta());
  }
  public void paint (Graphics g) {
    g.drawString ("Cuenta...", 20, 20);
    g.drawString (String.valueOf(laCuenta.getCuenta()), 20, 35 );
    n = laCuenta.incCuenta();
    g.drawString (String.valueOf(n), 20, 50 );
    laCuenta.incCuenta();
    g.drawString (String.valueOf(laCuenta.getCuenta()), 20, 65 );
    g.drawString (String.valueOf(laCuenta.incCuenta()), 20, 80 );
  }
}
```

Esta clase puede ejecutarse tanto con "java Ejemplo3" en una ventana DOS, como cargarse desde una página HTML con:

```
<applet code="Ejemplo3.class" width=170 height=150>
</applet>
```

Notar que conviene probar el applet con el appletviewer ("appletviewer ejemplo3.htm"), ya que éste indica en la ventana DOS si hay algún error durante la ejecución. Los browsers dejan pasar muchos errores, simplemente suprimiendo la salida a pantalla del código erróneo.

Notar que en todo este desarrollo de las clases Ejemplo1, Ejemplo2 y Ejemplo3, en ningún momento volvimos a tocar la clase Contador!

Estructura de clases

Vamos a comenzar analizando la clase Contador, para ir viendo las partes que forman una clase una por una y en detalle. Este capítulo va a ser un poco aburrido por lo exhaustivo (aunque algunos puntos más complicados como las excepciones y los threads los dejaremos para después), pero me parece bueno tener un resumen completo de la sintaxis desde ahora.
Luego iremos armando pequeñas aplicaciones para probar cada cosa.
Recordemos la definición de la clase Contador:

```
// Implementación de un contador sencillo
public class Contador {
// Atributos
int cnt;
// Constructor
public Contador() {
cnt = 0;
}
// Métodos
public int incCuenta() {
cnt++;
return cnt;
}
public int getCuenta() {
return cnt;
}
}
```

Declaración de la clase

La clase se declara mediante la línea public class Contador. En el caso más general, la declaración de una clase puede contener los siguientes elementos:

[**public**] [**final** | **abstract**] **class** Clase [**extends** ClaseMadre] [**implements** Interfase1 [, Interfase2]…]
o bien, para interfaces:
[**public**] **interface** Interfase [**extends** InterfaseMadre1 [, InterfaseMadre2]…]
Como se ve, lo único obligatorio es **class** y el nombre de la clase. Las interfases son un caso de clase particular que veremos más adelante.

Public, final o abstract

Definir una clase como pública (**public**) significa que puede ser usada por cualquier clase en cualquier paquete. Si no lo es, solamente puede ser utilizada por clases del mismo paquete (más sobre paquetes luego; básicamente, se trata de un grupo de clases e interfaces relacionadas, como los paquetes de biblioteca incluídos con Java).
Una clase final (**final**) es aquella que no puede tener clases que la hereden. Esto se utiliza básicamente por razones de seguridad (para que una clase no pueda ser reemplazada por otra que la herede), o por diseño de la aplicación.
Una clase abstracta (**abstract**) es una clase que puede tener herederas, pero no puede ser instanciada. Es, literalmente, abstracta (como la clase *Number* definida en java.lang). ¿Para qué sirve? Para modelar conceptos. Por ejemplo, la clase *Number* es una clase abstracta que representa cualquier tipo de números (y sus métodos no están implementados: son abstractos); las clases descendientes de ésta, como *Integer* o *Float*, sí implementan los métodos de la madre *Number*, y se pueden instanciar.
Por lo dicho, una clase no puede ser **final** y **abstract** a la vez (ya que la clase abstract requiere descendientes…)
¿Un poco complejo? Se va a entender mejor cuando veamos casos particulares, como las interfases (que por definición son abstractas ya que no implementan sus métodos).

Extends

La instrucción **extends** indica de qué clase desciende la nuestra. Si se omite, Java asume que desciende de la superclase **Object**.
Cuando una clase desciende de otra, esto significa que hereda sus atributos y sus métodos (es decir que, a menos que los redefinamos, sus métodos son los mismos que los de la clase madre y pueden utilizarse en forma transparente, a menos que sean *privados* en la clase madre o, para subclases de otros paquetes, protegidos o propios del paquete). Veremos la calificación de métodos muy pronto, a no desesperar!

Implements

Una interfase (**interface**) es una clase que declara sus métodos pero no los implementa; cuando una clase implementa (**implements**) una o más interfases, debe contener la implementación de todos los métodos (con las mismas listas de parámetros) de dichas interfases.
Esto sirve para dar un ascendiente común a varias clases, obligándolas a implementar los mismos métodos y, por lo tanto, a comportarse de forma similar en cuanto a su interfase con otras clases y subclases.

Interface

Una interfase (**interface**), como se dijo, es una clase que no implementa sus métodos sino que deja a cargo la implementación a otras clases. Las interfases pueden, asimismo, descender de otras interfases pero no de otras clases.
Todos sus métodos son por definición abstractos y sus atributos son finales (aunque esto no se indica en el cuerpo de la interfase).
Son útiles para generar relaciones entre clases que de otro modo no están relacionadas (haciendo que implementen los mismos métodos), o para distribuir paquetes de clases indicando la estructura de la interfase pero no las clases individuales (objetos anónimos).
Si bien diferentes clases pueden implementar las mismas interfases, y a la vez descender de otras clases, esto no es en realidad herencia múltiple ya que una clase no puede heredar atributos ni métodos de una interface; y las clases que implementan una interfase pueden no estar ni siquiera relacionadas entre sí.

El cuerpo de la clase

El cuerpo de la clase, encerrado entre { y }, es la lista de atributos (variables) y métodos (funciones) que constituyen la clase.
No es obligatorio, pero en general se listan primero los atributos y luego los métodos.

Declaración de atributos

En Java no hay variables globales; todas las variables se declaran dentro del cuerpo de la clase o dentro de un método. Las variables declaradas dentro de un método son **locales** al método; las variables declaradas en el cuerpo de la clase se dice que son *miembros* de la clase y son accesibles por todos los métodos de la clase.
Por otra parte, además de los atributos de la propia clase se puede acceder a todos los atributos de la clase de la que desciende; por ejemplo, cualquier clase que descienda de la clase **Polygon** hereda los atributos *npoints*, *xpoints* e *ypoints*.
Finalmente, los atributos miembros de la clase pueden ser *atributos de clase* o *atributos de instancia*; se dice que son atributos *de clase* si se usa la palabra clave **static**: en ese caso la variable es única para todas las instancias (objetos) de la clase (ocupa un único lugar en memoria). Si no se usa static, el sistema crea un lugar nuevo para esa variable con cada instancia (o sea que es independiente para cada objeto).
La declaración sigue siempre el mismo esquema:
[**private**|**protected**|**public**] [**static**] [**final**] [**transient**] [**volatile**] Tipo NombreVariable [= Valor]**;**

Private, protected o public

Java tiene 4 tipos de acceso diferente a las variables o métodos de una clase: privado, protegido, público o por paquete (si no se especifica nada).
De acuerdo a la forma en que se especifica un atributo, objetos de otras clases tienen distintas posibilidades de accederlos:

Acceso desde:	**private**	**protected**	**public**	(package)
la propia clase	S	S	S	S
subclase en el mismo paquete	N	S	S	S
otras clases en el mismo paquete	N	S	S	S
subclases en otros paquetes	N	X	S	N
otras clases en otros paquetes	N	N	S	N

S: puede acceder
N: no puede acceder
X: puede acceder al atributo en objetos que pertenezcan a la subclase, pero no en los que pertenecen a la clase madre. Es un caso especial ; más adelante veremos ejemplos de todo esto.

Static y final

Como ya se vio, **static** sirve para definir un atributo como *de clase*, o sea único para todos los objetos de la clase.
En cuanto a **final**, como en las clases, determina que un atributo no pueda ser sobreescrito o redefinido. O sea: no se trata de una variable, sino de una *constante*.

Transient y volatile

Son casos bastante particulares y que no habían sido implementados en Java 1.0.
Transient denomina atributos que no se graban cuando se archiva un objeto, o sea que no forman parte del estado permanente del mismo.

Volatile se utiliza con variables modificadas asincrónicamente por objetos en diferentes *threads* (literalmente "hilos", tareas que se ejecutan en paralelo); básicamente esto implica que distintas tareas pueden intentar modificar la variable simultáneamente, y *volatile* asegura que se vuelva a leer la variable (por si fue modificada) cada vez que se la va a usar (esto es, en lugar de usar registros de almacenamiento como buffer).

Los tipos de Java

Los tipos de variables disponibles son básicamente 3:

- tipos básicos (no son objetos)
- arreglos (arrays)
- clases e interfases

Con lo que vemos que cada vez que creamos una clase o interfase estamos definiendo un nuevo tipo.
Los **tipos básicos** son:

Tipo	**Tamaño/Formato**	**Descripción**
byte	8-bit complemento a 2	Entero de un byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
float	32-bit IEEE 754	Punto flotante, precisión simple
double	64-bit IEEE 754	Punto flotante, precisión doble
char	16-bit caracter Unicode	Un caracter
boolean	true, false	Valor booleano (verdadero o falso)

Los **arrays** son arreglos de cualquier tipo (básico o no). Por ejemplo, existe una clase *Integer*; un arreglo de objetos de dicha clase se notaría:
Integer vector[ ];
Los arreglos siempre son dinámicos, por lo que **no es válido** poner algo como:
Integer cadena[5];
Aunque sí es válido inicializar un arreglo, como en:
int días[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
char letras[ ] = { 'E', 'F', 'M', 'A', 'M', 'J', 'J', 'A', 'S', 'O', 'N', 'D' };
String nombres[ ] = new String[12];
Nota al margen: no confundir un String (*cadena* de caracteres) con un *arreglo* de caracteres! Son cosas bien distintas!
Ya hablaremos más adelante de las clases String y StringBuffer.
En Java, para todas las variables de tipo básico se accede al valor asignado a la misma directamente (no se conoce la dirección de memoria que ocupa). Para las demás (arrays, clases o interfases), se accede a través de un puntero a la variable. El valor del puntero no es accesible ni se puede modificar (como en C); Java no necesita esto y además eso atentaría contra la robustez del lenguaje.
De hecho, en Java no existen los tipos **pointer**, **struct** o **union**. Un objeto es más que una estructura, y las uniones no se hacen necesarias con un método de programación adecuado (y además se evita la posibilidad de acceder a los datos incorrectamente).
Algo más respecto a los arreglos: ya que Java gestiona el manejo de memoria para los mismos, y lanza excepciones si se intenta violar el espacio asignado a una variable, se evitan problemas típicos de C como acceder a lugares de memoria prohibidos o fuera del lugar definido para la variable (como cuando se usa un subíndice más grande que lo previsto para un arreglo…).

Y los métodos…

Los métodos, como las clases, tienen una declaración y un cuerpo.
La declaración es del tipo:
[**private**|**protected**|**public**] [**static**] [**abstract**] [**final**] [**native**] [**synchronized**] TipoDevuelto NombreMétodo **(** [tipo1 nombre1[, tipo2 nombre2]…] **)** [**throws** excepción1 [,excepción2]…]
A no preocuparse: poco a poco aclararemos todo con ejemplos.

Básicamente, los métodos son como las funciones de C: implementan, a través de funciones, operaciones y estructuras de control, el cálculo de algún parámetro que es el que devuelven al objeto que los llama. Sólo pueden devolver **un** valor (del tipo *TipoDevuelto*), aunque pueden no devolver ninguno (en ese caso *TipoDevuelto* es **void**). Como ya veremos, el valor de retorno se especifica con la instrucción **return**, dentro del método.
Los métodos pueden utilizar valores que les pasa el objeto que los llama (*parámetros*), indicados con *tipo1 nombre1, tipo2 nombre2…* en el esquema de la declaración.
Estos parámetros pueden ser de cualquiera de los tipos ya vistos. Si son tipos básicos, el método recibe el *valor* del parámetro; si son arrays, clases o interfases, recibe un puntero a los datos (*referencia*). Veamos un pequeño ejemplo:

```
public int AumentarCuenta(int cantidad) {
cnt = cnt + cantidad;
return cnt;
}
```

Este método, si lo agregamos a la clase **Contador**, le suma *cantidad* al acumulador **cnt**. En detalle:

- el método recibe un valor entero (*cantidad*)
- lo suma a la variable de instancia *cnt*
- devuelve la suma (*return cnt*)

¿Cómo hago si quiero devolver más de un valor? Por ejemplo, supongamos que queremos hacer un método dentro de una clase que devuelva la posición del mouse.
Lo siguiente no sirve:

```
                    void GetMousePos(int x, int y) {
                            x = ….;                 // esto no sirve!
                            y = ….;                 // esto tampoco!
}
```

porque el método no puede modificar los parámetros *x* e *y* (que han sido pasados por valor, o sea que el método recibe el valor numérico pero no sabe adónde están las variables en memoria).
La solución es utilizar, en lugar de tipos básicos, una clase:

```
                    class MousePos { public int x, y; }
```

y luego utilizar esa clase en nuestro método:

```
                    void GetMousePos( MousePos m ) {
                            m.x = ……;
                            m.y = ……;
}
```

El resto de la declaración

Public, **private** y **protected** actúan exactamente igual para los métodos que para los atributos, así que veamos el resto.
Los métodos estáticos (**static**), son, como los atributos, métodos *de clase*; si el método no es **static** es un método *de instancia*. El significado es el mismo que para los atributos: un método *static* es compartido por todas las instancias de la clase.
Ya hemos hablado de las clases abstractas; los métodos abstractos (**abstract**) son aquellos de los que se da la declaración pero no la implementación (o sea que consiste sólo del encabezamiento). Cualquier clase que contenga al menos un método abstracto (o cuya clase madre contenga al menos un método abstracto que no esté implementado en la hija) es una clase abstracta.
Es **final** un método que no puede ser redefinido por ningún descendiente de la clase.