Apuntes Lenguaje Java

por Antonio Bel Puchol - abelp@arrakis.es

Estos apuntes del lenguaje Java son el guión estructurado de un curso de Introducción al Lenguaje Java. No son una referencia exahustiva del lenguaje sino que pretenden proporcionar una aproximación progresiva, partiendo de lo más elemental e introduciendo los conceptos ordenadamente, apoyándose en lo anterior.

Están estructurados en forma de lecciones breves, que abarcan todos los aspectos del lenguaje en un nivel básico-medio. No se presuponen conocimientos de otros lenguajes de programación, aunque obviamente estar familiarizados con conceptos relativos a la programación como variables, algoritmos, objetos, etc. facilita mucho la tarea. Sin embargo, se ha evitado intencionadamente la referencia o la comparación con otros lenguajes, especialmente con C/C++ que tan presente está en muchos textos de Java.

La primera parte de estos apuntes cubre todos los aspectos del lenguaje (Clases, Interfaces, herencia, control de flujo, tipos de datos, etc.). La segunda parte, actualmente en elaboración, trata materias de la API de Java, agrupándolos por temas (Entrada/Salida, Threads, applets, etc.).

Cualquier comentario será bien recibido en abelp@arrakis.es . Puedes probar algunos programas Java hechos por mi aquí.

Antonio Bel Puchol - abelp@arrakis.es

Ultima actualización - 17-Junio-2001

Puedes encontrar estos apuntes en:
http://www.arrakis.es/~abelp/ApuntesJava/indice.htm
o también en
http://www.geocities.com/belpuchol/ApuntesJava/indice.htm

Tabla de Contenido

5. Clases - Constructores (Actualizado 18-2-2001)

6. Clases - Miembros estáticos (Actualizado 4-3-2001)

7. Clases - Otros aspectos (Actualizado 25-3-2001)

8. Control de la Ejecución (Actualizado 25-3-2001)

9. Arrays (Actualizado 25-3-2001)

10. Strings (Actualizado 25-3-2001)

11. Packages (Actualizado 17-6-2001)

12. Compilación y ejecución de programas (Actualizado 17-6-2001)

13. Modificadores de acceso (Actualizado 17-6-2001)

14. Herencia - I

3. Entrada / Salida --- En elaboración ---

4. Applets --- En elaboración ---

5. Contenedores --- En elaboración ---

Indice Siguiente Anterior

Comentarios, documentación y convenciones de nombres

Comentarios

En Java existen comentarios de línea con // y bloques de comentario que comienzan con /* y terminan con */. Por ejemplo:

```
// Comentario de una linea
/* comienzo de comentario
   continua comentario
   fin de comentario */
```

Comentarios para documentación

El JDK proporciona una herramienta para generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. El nombre de la herramienta es javadoc. Para que javadoc pueda generar los textos HTML es necesario que se sigan unas normas de documentación en el fuente, que son las siguientes:

- Los comentarios de documentación deben empezar con `/**` y terminar con `*/`.
- Se pueden incorporar comentarios de documentación a nivel de clase, a nivel de variable (dato miembro) y a nivel de método.
- Se genera la documentación para miembros public y protected.
- Se pueden usar tags para documentar ciertos aspectos concretos como listas de parámetros o valores devueltos. Los tags se indican a continuación.

Tipo de tag	Formato	Descripción
Todos	@see	Permite crear una referencia a la documentación de otra clase o método.
Clases	@version	Comentario con datos indicativos del número de versión.
Clases	@author	Nombre del autor.

Clases	@since	Fecha desde la que está presente la clase.
Métodos	@param	Parámetros que recibe el método.
Métodos	@return	Significado del dato devuelto por el método
Métodos	@throws	Comentario sobre las excepciones que lanza.
Métodos	@deprecated	Indicación de que el método es obsoleto.

Toda la documentación del API de Java está creada usando esta técnica y la herramienta javadoc.

Una clase comentada

```
import java.util.*;

/** Un programa Java simple.
  * Envia un saludo y dice que día es hoy.
  * @author Antonio Bel
  * @version 1
  */
public class HolaATodos {

/** Unico punto de entrada.
  * @param args Array de Strings.
  * @return No devuelve ningun valor.
  * @throws No dispara ninguna excepcion.
  */
public static void main(String [ ] args) {
    System.out.println("Hola a todos");
    System.out.println(new Date());
}

}
```

Convenciones de nombres

SUN recomienda un estilo de codificación que es seguido en el API de Java y en estos apuntes que consiste en:

- Utilizar nombres descriptivos para las clases, evitando los nombres muy largos.
- Para los nombres de clases poner la primera letra en mayúsculas y las demás en minúsculas. Por ejemplo: Empleado
- Si el nombre tiene varias palabras ponerlas todas juntas (sin separar con - o _) y poner la primera letra de cada palabra en mayúsculas. Por ejemplo: InstrumentoMusical.
- Para los nombres de miembros (datos y métodos) seguir la misma norma, pero con la primera

letra de la primera palabra en minúsculas. Por ejemplo: registrarOyente.
- Para las constantes (datos con el modificador final) usar nombres en mayúsculas, separando las palabras con _

Indice Siguiente Anterior

Indice Siguiente Anterior

Clases embebidas (Inner classes)

Concepto

Una clase embebida es una clase que se define dentro de otra. Es una característica de Java que permite agrupar clases lógicamente relacionadas y controlar la 'visibilidad' de una clase. El uso de las clases embebidas no es obvio y contienen detallas algo más complejos que escapan del ámbito de esta introducción.

Se puede definir una clase embebida de la siguiente forma:

```
class Externa {
    . . .
    class Interna {
        . . .
    }
}
```

La clase Externa puede instanciar y usar la clase Interna como cualquier otra, sin limitación ni cambio en la sintaxis de acceso:

```
class Externa {
    . . .
    class Interna {
        . . .
    }
    void metodo() {
        Interna i = new Interna(. . .);
        . . .
    }
}
```

Una diferencia importante es que un objeto de la clase embebida está relacionado siempre con un objeto de la clase que la envuelve, de tal forma que las instancias de la clase embebida deben ser creadas por una instancia de la clase que la envuelve. Desde el exterior estas referencias pueden manejarse, pero calificandolas completamente, es decir nombrando la clase externa y luego la interna. Además una instancia de la clase embebida tiene acceso a todos los datos miembros de la clase que la

envuelve sin usar ningún calificador de acceso especial (como si le pertenecieran). Todo esto se ve en el ejemplo siguiente.

Ejemplo

Un ejemplo donde puede apreciarse fácilmente el uso de clases embebidas es el concepto de iterador. Un iterador es una clase diseñada para recorrer y devolver ordenadamente los elementos de algún tipo de contenedor de objetos. En el ejemplo se hace una implementación muy elemental que trabaja sobre un array.

```
class Almacen {
    private Object [] listaObjetos;
    private numElementos = 0;
    Almacen (int maxElementos) {
        listaObjetos = new Object[maxElementos];
    }
    public Object get(int i) {
        return listaObject[i];
    }
    public void add(Object obj) {
        listaObjetos[numElementos++] = obj;
    }
    public Iterador getIterador() {
        new Iterador();
    }

    class Iterador {
        int indice = 0;
        Object siguiente() {
            if (indice < numElementos) return
listaObjetos[indice++];
            else return null;
        }
    }
}
```

Y la forma de usarse, sería:

```
Almacen alm = new Almacen(10); // se crea un
nuevo almacen
. . .
alm.add(...);    // se añaden objetos
. . .
// para recorrerlo
Almacen.Iterador i = alm.getIterador(); //
```

```
obtengo un iterador para alm
Object o;
while ( (o = i.siguiente()) != null) {
    . . .
}
```

Indice Siguiente Anterior

1. Introducción

Java se creó como parte de un proyecto de investigación para el desarrollo de software avanzado para una amplia variedad de dispositivos de red y sistemas embebidos. La meta era diseñar una plataforma operativa sencilla, fiable, portable, distribuida y de tiempo real. Cuando se inició el proyecto, C++ era el lenguaje del momento. Pero a lo largo del tiempo, las dificultades encontradas con C++ crecieron hasta el punto en que se pensó que los problemas podrían resolverse mejor creando una plataforma de lenguaje completamente nueva. Se extrajeron decisiones de diseño y arquitectura de una amplia variedad de lenguajes como Eiffel, SmallTalk, Objetive C y Cedar/Mesa. El resultado es un lenguaje que se ha mostrado ideal para desarrollar aplicaciones de usuario final seguras, distribuidas y basadas en red en un amplio rango de entornos desde los dispositivos de red embebidos hasta los sistemas de sobremesa e Internet.

1.1. Objetivos de diseño de Java

Java fue diseñado para ser:

- **Sencillo, orientado a objetos y familiar**: Sencillo, para que no requiera grandes esfuerzos de entrenamiento para los desarrolladores. Orientado a objetos, porque la tecnología de objetos se considera madura y es el enfoque más adecuado para las necesidades de los sistemas distribuidos y/o cliente/servidor. Familiar, porque aunque se rechazó C++, se mantuvo Java lo más parecido posible a C++, eliminando sus complejidades innecesarias, para facilitar la migración al nuevo lenguaje.

- **Robusto y seguro**: Robusto, simplificando la gestión de memoria y eliminando las complejidades de la gestión explicita de punteros y aritmética de punteros del C. Seguro para que pueda operar en un entorno de red.

- **Independiente de la arquitertura y portable**: Java está diseñado para soportar aplicaciones que serán instaladas en un entorno de red heterogéneo, con hardware y

sistemas operativos diversos. Para hacer esto posible el compilador Java genera 'bytecodes', un formato de código independiente de la plataforma diseñado para transportar código eficientemente a través de múltiples plataformas de hardware y software. Es además portable en el sentido de que es rigurosamente el mismo lenguaje en todas las plataformas. El 'bytecode' es traducido a código máquina y ejecutado por la Java Virtual Machine, que es la implementación Java para cada plataforma hardware-software concreta.

- **Alto rendimiento**: A pesar de ser interpretado, Java tiene en cuenta el rendimiento, y particularmente en las últimas versiones dispone de diversas herramientas para su optimización. Cuando se necesitan capacidades de proceso intensivas, pueden usarse llamadas a código nativo.

- **Interpretado, multi-hilo y dinámico**: El intérprete Java puede ejecutar bytecodes en cualquier máquina que disponga de una Máquina Virtual Java (JVM). Además Java incorpora capacidades avanzadas de ejecución multi-hilo (ejecución simultánea de más de un flujo de programa) y proporciona mecanismos de carga dinámica de clases en tiempo de ejecución.

1.2. Características de Java

- Lenguaje de propósito general.
- Lenguaje Orientado a Objetos.
- Sintaxis inspirada en la de C/C++.
- Lenguaje multiplataforma: Los programas Java se ejecutan sin variación (sin recompilar) en cualquier plataforma soportada (Windows, UNIX, Mac...)
- Lenguaje interpretado: El intérprete a código máquina (dependiente de la plataforma) se llama Java Virtual Machine (JVM). El compilador produce un código intermedio independiente del sistema denominado *bytecode*.
- Lenguaje gratuito: Creado por SUN Microsystems, que distribuye gratuitamente el producto base, denominado JDK (Java Development Toolkit) o actualmente J2SE (Java 2 Standard Edition).
- API distribuida con el J2SE muy amplia. Código fuente de la API disponible.

1.3. Qué incluye el J2SE (Java 2 Standard Edition)

- Herramientas para generar programas Java. Compilador, depurador, herramienta para documentación, etc.
- La JVM, necesaria para ejecutar programas Java.
- La API de Java (jerarquía de clases).
- Código fuente de la API (Opcional).
- Documentación.

La versión actual (Enero 2001) es la 1.3.0.

1.4. Qué es el JRE (Java Runtime Environment)

JRE es el entorno mínimo para ejecutar programas Java 2. Incluye la JVM y la API. Está incluida en el J2SE aunque puede descargarse e instalarse separadamente. En aquellos sistemas donde se vayan a ejecutar programas Java, pero no compilarlos, el JRE es suficiente.

El JRE incluye el Java Plug-in, que es el 'añadido' que necesitan lo navegadores (Explorer o Netscape) para poder ejecutar programas Java 2. Es decir que instalando el JRE se tiene soporte completo Java 2, tanto para aplicaciones normales (denominadas 'standalone') como para Applets (programas Java que se ejecutan en una página Web, cuando esta es accedida desde un navegador).

1.5. Qué se necesita para empezar

El entorno mínimo necesario para escribir, compilar y ejecutar programas Java es el siguiente:

- J2SE (Java 2 Standard Edition) y la documentación (Se descargan por separado). Esto incluye el compilador Java, la JVM, el entorno de tiempo de ejecución y varias herramientas de ayuda. La documentación contiene la referencia completa de la API. Puedes descargar esto en http://java.sun.com/j2se
- Un editor de textos. Cualquiera sirve. Pero un editor especializado con ayudas específicas para Java (como el marcado de la sintaxis, indentación, paréntesis, etc.) hace más cómodo el desarrollo. Por ejemplo Jext (Java Text Editor) es un magnifico editor. Es gratuito y además está escrito en Java. Puedes descargarlo en http://sourceforge.net/projects/jext/
- De forma opcional puede usarse un Entorno de Desarrollo Integrado para Java (IDE). Una herramienta de este tipo resulta aconsejable como ayuda para desarrollar aplicaciones o componentes. Sin embargo, en las primeras etapas del aprendizaje de Java no resulta necesario (ni siquiera conveniente, en mi opinión). Un IDE excelente, gratuito y perfectamente adaptado a todas las características de Java es Netbeans (versión open-source del Forte for Java de Sun). Puedes descargarlo en http://www.netbeans.org

---

Ultima actualización - 26-Enero-2001

Antonio Bel Puchol - abelp@arrakis.es

Otras páginas del autor

abelp@arrakis.es

StarMap - Un viaje a través de las estrellas

StarMap es un programa que presenta un mapa a escala del espacio próximo alrededor de nuestro Sol, en un radio aproximado de 80 años luz. Contiene las posiciones, nombres y características más importantes de unas 3800 estrellas. Programa Java 2.

http://www.arrakis.es/~abelp/index.html

---

jEscoba - El juego de la escoba

jEscoba es un programa para jugar al tradicional juego de cartas "La Escoba". Puede jugarse desde un solo ordenador, o entre varios ordenadores conectados en una Red Local (LAN) o a través de Internet, enfrentando a dos, tres, cuatro o seis jugadores, ya sean personas o autómatas controlados por el propio programa. Programa Java 2.

http://www.arrakis.es/~abelp/escoba/indice.html

---

jRabino - El juego del rabino francés

jRabino es un programa para jugar al juego de cartas 'El Rabino', en una variante conocida como 'Rabino francés'. jRabino puede jugarse en un sólo ordenador, enfrentándote con uno, dos o tres 'jugadores automáticos' controlados por el sistema, o bien entre varios ordenadores conectados en una red local o a través de Internet. Programa Java 2.

http://www.arrakis.es/~abelp/rabino/rabino.htm

Apuntes del lenguaje Java

Breve curso de introducción al Lenguaje Java.

http://www.arrakis.es/~abelp/ApuntesJava/indice.htm

http://www.geocities.com/belpuchol/ApuntesJava/indice.htm

Colección de solitarios de cartas - jSol

Colección de solitarios de cartas, escrito en Java. Klondike, y otros más.

http://www.arrakis.es/~abelp/jSol/solitarios.html

http://www.geocities.com/belpuchol/jSol/solitarios.html

abelp@arrakis.es

Indice de Modificaciones

25-Marzo-2001

- Revisión capítulos 7, 8, 9 y 10.
- Apendice I. Sintaxis Java. Continuación (Sin terminar)

4-Marzo-2001

- Revisión capítulo 6.
- Añadido Apéndice I. Sintaxis de Java (Sin terminar).

18-Febrero-2001

- Añadida lección 23. Threads I.
- Revisión capítulos 4 y 5.

4-Febrero-2001

- Añadido capítulo 22. Introducción API.
- Cambio de formato y mejoras capítulos 2 y 3.

26-Enero-2001

- Cambios en portada. Capítulos numerados. Inclusión en Geocities.
- Introducción ampliada.

Octubre 2000

Versión inicial


TOC

---

2. Tipos, Valores y Variables

2.1. Tipos

Java es un lenguaje con control fuerte de Tipos (*Strongly Typed*). Esto significa que cada variable y cada expresión tiene un Tipo que es conocido en el momento de la compilación. El Tipo limita los valores que una variable puede contener, limita las operaciones soportadas sobre esos valores y determina el significado de la operaciones. El control fuerte de tipos ayuda a detectar errores en tiempo de compilación.

En Java existen dos categorias de Tipos:

- Tipos Primitivos
- Referencias

Las referencias se usan para manipular objetos. Se verán en una lección posterior.

2.2. Tipos primitivos

Los tipos primitivos son los que permiten manmipular valores numéricos (con distintos grados de precisión), caracteres y valores booleanos (verdadero / falso). Los Tipos Primitivos son:

- **`boolean`** : Puede contener los valores **`true`** o **`false`**.
- **`byte`** : Enteros. Tamaño 8-bits. Valores entre -128 y 127.
- **`short`** : Enteros. Tamaño 16-bits. Entre -32768 y 32767.
- **`int`** : Enteros. Tamaño 32-bits. Entre -2147483648 y 2147483647.
- **`long`** : Enteros. Tamaño 64-bits. Entre -9223372036854775808 y 9223372036854775807.
- **`float`** : Números en coma flotante. Tamaño 32-bits.

- **`double`** : Números en coma flotante. Tamaño 64-bits.
- **`char`** : Caracteres. Tamaño 16-bits. Unicode. Desde `'\u0000'` a `'\uffff'` inclusive. Esto es desde 0 a 65535

El tamaño de los tipos de datos no depende de la implementación de Java. Son siempre los mismos.

2.3. Variables

Una variable es un área en memoria que tiene un nombre y un Tipo asociado. El Tipo es o bien un Tipo primitivo o una Referencia.

Es obligatorio declarar las variables antes de usarlas. Para declararlas se indica su nombre y su Tipo, de la siguiente forma:

tipo_variable nombre **;**

Ejemplos:

```
int i;          // Declaracion de un entero
char letra;     // Declaracion de un caracter
boolean flag;   // Declaracion de un booleano
```

- El **;** es el separador de sentencias en Java.
- El símbolo `//` indica comentarios de línea.
- En Java las mayúsculas y minúsculas son significativas. No es lo mismo el nombre `letra` que `Letra`.
- Todas las palabras reservadas del lenguaje van en minúsculas.
- Todas las palabras que forman parte del lenguaje van en **negrita** a lo largo de todos los apuntes.

Se pueden asignar valores a las variables mediante la instrucción de asignación. Por ejemplo:

```
i = 5;          // a la variable i se le asigna el valor 5
letra = 'c';    // a la variable letra se le asigna el
valor 'c'
flag = false;  // a la variable flag se le asigna el valor
false
```

La declaración y la combinación se pueden combinar en una sola expresión:

```
int i = 5;
char letra = 'c';
boolean flag = false;
```

2.4. Literales

En los literales numéricos puede forzarse un tipo determinado con un sufijo:

- Entero largo: l ó L.
- Float: f ó F
- Double: d ó D.

Por ejemplo:

```
long l = 5L;
float numero = 5f;
```

En los literales numéricos para float y double puede usarse el exponente (10 elevado a...) de la forma: 1.5e3D (equivale a 1500)

Literales hexadecimales: Al valor en hexadecimal se antepone el símbolo 0x. Por ejemplo: 0xf (valor 15)

Literales booleanos: **`true`** (verdadero) y **`false`** (falso)

Literales caracter: Un caracter entre apóstrofes (') o bien una sequencia de escape (también entre '). Las secuencias de escape están formadas por el símbolo \ y una letra o un número. Algunas secuencias de escape útiles:

- `\n`: Salto de línea
- `\t`: Tabulador
- `\b`: Backspace.
- `\r`: Retorno de carro
- `\uxxxx`: donde xxxx es el código Unicode del carácter. Por ejemplo \u0027 es el apostrofe (')

2.5. Operaciones sobre Tipos primitivos

La siguiente tabla muestra un resumen de los operadores existentes para las distrintas clases de tipos primitivos. El grupo 'Enteros' incluye byte, short, int, long y char. El grupo 'Coma flotante' incluye float and double.

Tipos	**Grupo de operadores**	**Operadores**
Enteros	Operadores de comparación que devuelven un valor `boolean`	`< (menor) , <= (menor o igual) , > (mayor), >= (mayor o igual), == (igual), != (distinto)`

	Operadores numéricos, que devuelven un valor `int` o `long`	`+ (unario, positivo), - (unario, negativo), + (suma) , - (resta) , * (multiplicación), / (división), % (resto), ++ (incremento), -- (decremento), <<, >>, >>> (rotación) , ~ (complemento a nivel de bits), & (AND a nivel de bits), | (OR a nivel de bits) ^(XOR a nivel de bits)`
Coma flotante	Operadores de comparación que devuelven un valor `boolean`	`< (menor) , <= (menor o igual) , > (mayor), >= (mayor o igual), == (igual), != (distinto)`
	Operadores numéricos, que devuelven un valor `float` o `double`	`+ (unario, positivo), - (unario, negativo), + (suma) , - (resta) , * (multiplicación), / (división), % (resto), ++ (incremento), -- (decremento).`
Booleanos	Operadores booleanos	`== (igual), != (distinto), ! (complemento), & (AND), | (OR), ^ (XOR), && (AND condicional), || (OR condicional)`

Ultima actualización - 4-Febrero-2001

Antonio Bel Puchol - abelp@arrakis.es

3. Métodos

3.1 Declaración de métodos

En Java toda la lógica de programación (Algoritmos) está agrupada en funciones o métodos.

Un método es:

- Un bloque de código que tiene un nombre,
- recibe unos parámetros o argumentos (opcionalmente),
- contiene sentencias o instrucciones para realizar algo (opcionalmente) y
- devuelve un valor de algún Tipo conocido (opcionalmente).

La sintaxis global es:

Tipo_Valor_devuelto nombre_método **(** *lista_argumentos* **) {**
    bloque_de_codigo;
}

y la lista de argumentos se expresa declarando el tipo y nombre de los mismos (como en las declaraciones de variables). Si hay más de uno se separan por comas.

Por ejemplo:

```
int sumaEnteros ( int a, int b ) {
        int c = a + b;
        return c;
}
```

- El método se llama sumaEnteros.
- Recibe dos parámetros también enteros. Sus nombres son a y b.

- Devuelve un entero.

En el ejemplo la claúsula **`return`** se usa para finalizar el método devolviendo el valor de la variable c.

3.2. El termino `void`

El hecho de que un método devuelva o no un valor es opcional. En caso de que devuelva un valor se declara el tipo que devuelve. Pero si no necesita ningún valor, se declara como tipo del valor devuelto, la palabra reservada **`void`**. Por ejemplo:

```
void haceAlgo() {
     . . .
 }
```

Cuando no se devuelve ningún valor, la claúsula **`return`** no es necesaria. Observese que en el ejemplo el método haceAlgo tampoco recibe ningún parámetro. No obstante los paréntesis, son obligatorios.

3.3. Uso de métodos

Los métodos se invocan con su nombre, y pasando la lista de argumentos entre paréntesis. El conjunto se usa como si fuera una variable del Tipo devuelto por el método.

Por ejemplo:

```
int x;
x = sumaEnteros(2,3);
```

Nota: Esta sintaxis no está completa, pero sirve para nuestros propósitos en este momento. La sintaxis completa se verá cuando se hable de objetos.

Aunque el método no reciba ningún argumento, los paréntesis en la llamada son obligatorios. Por ejemplo para llamar a la función haceAlgo, simplemente se pondría:

```
haceAlgo();
```

Observese que como la función tampoco devuelve ningún valor no se asigna a ninguna variable. (No hay nada que asignar).

Ultima actualización - 4-Febrero-2001

Antonio Bel Puchol - abelp@arrakis.es

4. Clases - Introducción

4.1. Clases

Las clases son el mecanismo por el que se pueden crear nuevos Tipos en Java. Las clases son el punto central sobre el que giran la mayoría de los conceptos de la Orientación a Objetos.

Una clase es una agrupación de datos y de código que actua sobre esos datos, a la que se le da un nombre.

Una clase contiene:

- Datos (se denominan Datos Miembro). Estos pueden ser de tipos primitivos o referencias.
- Métodos (se denominan Métodos Miembro).

La sintaxis general para la declaración de una clase es:

```
modificadores class nombre_clase {
        declaraciones_de_miembros ;
}
```

Por ejemplo:

```
class Punto {
     int x;
     int y;
}
```

Los modificadores son palabras clave que afectan al comportamiento de la clase. Los iremos viendo progresivamente en los sucesivos capítulos.

4.2. Objetos, miembros y referencias

Un objeto es una instancia (ejemplar) de una clase. La clase es la definición general y el objeto es la materialización concreta (en la memoria del ordenador) de una clase.

El fenómeno de crear objetos de una clase se llama instanciación.

Los objetos se manipulan con referencias. Una referencia es una variable que apunta a un objeto. Las referencias se declaran igual que las variables de Tipos primitivos (tipo nombre). Los objetos se crean (se instancian) con el operador de instanciación **new**.

Ejemplo:

```
Punto p;
p = new Punto();
```

La primera línea del ejemplo declara una referencia (p) que es de Tipo Punto. La referencia no apunta a ningún sitio. En la segunda línea se crea un objeto de Tipo Punto y se hace que la referencia p apunte a él. Se puede hacer ambas operaciones en la misma expresión:

```
Punto p = new Punto();
```

A los miembros de un objeto se accede a través de su referencia. La sintaxis es:

nombre_referencia.miembro

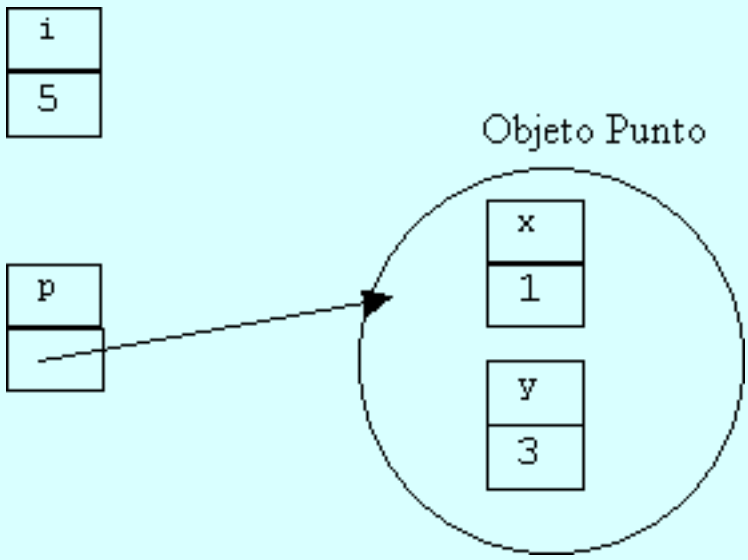
En el ejemplo, se puede poner:

```
p.x = 1;
p.y = 3;
```

Se puede visualizar gráficamente los datos primitivos, referencias y objetos de la siguiente forma:

- Datos primitivos: `int i = 5;`
- Referencias y objetos:

```
Punto p = new Punto();
p.x = 1;
p.y = 3;
```



Es importante señalar que en el ejemplo, p no es el objeto. Es una referencia que apunta al objeto.

Los métodos miembro se declaran dentro de la declaración de la clase, tal como se ha visto en el capítulo anterior. Por ejemplo:

```
class Circulo {
        Punto centro;   // dato miembro.
Referencia a un objeto punto
        int radio;      // dato miembro. Valor
primitivo
        float superficie() {                //
método miembro.
                return 3.14 * radio * radio;
        }                                   //
fin del método superficie
}                                           //
fin de la clase Circulo
```

El acceso a métodos miembros es igual que el que ya se ha visto para datos miembro. En el ejemplo:

```
Circulo c = new Circulo();
c.centro.x = 2;
c.centro.y = 3;
c.radio = 5;
float s = c.superficie();
```

Es interesante observar en el ejemplo:

- Los datos miembro pueden ser tanto primitivos como referencias. La clase Circulo contiene un dato miembro de tipo Punto (que es el centro del círculo).
- El acceso a los datos miembros del Punto centro se hace encadenando el operador . en la expresión `c.centro.x` que se podría leer como 'el miembro x del objeto (Punto) centro del

objeto (Circulo) c'.
- Aunque el método superficie no recibe ningún argumento los paréntesis son obligatorios (Distinguen los datos de los métodos).
- Existe un Objeto Punto para cada instancia de la clase Circulo (que se crea cuando se crea el objeto Circulo).

4.3. Conceptos básicos. Resumen

- Una Clase es una definición de un nuevo Tipo, al que se da un nombre.
- Una Clase contiene Datos Miembro y Métodos Miembro que configuran el estado y las operaciones que puede realizar.
- Un Objeto es la materialización (instanciación) de una clase. Puede haber tantos Objetos de una Clase como resulte necesario.
- Los Objetos se crean (se les asigna memoria) con el Operador **new**.
- Los Objetos se manipulan con Referencias.
- Una Referencia es una Variable que apunta a un Objeto.
- El acceso a los elementos de un Objeto (Datos o métodos) se hace con el operador . (punto) : *nombre_referencia.miembro*

Ultima actualización - 18-Febrero-2001

Antonio Bel Puchol - abelp@arrakis.es

5. Clases - Constructores

5.1. Noción de constructor

Cuando se crea un objeto (se instancia una clase) es posible definir un proceso de inicialización que prepare el objeto para ser usado. Esta inicialización se lleva a cabo invocando un método especial denominado constructor. Esta invocación es implícita y se realiza automáticamente cuando se utiliza el operador **new**. Los constructores tienen algunas características especiales:

- El nombre del constructor tiene que se igual al de la clase.
- Puede recibir cualquier número de argumentos de cualquier tipo, como cualquier otro método.
- No devuelve ningún valor (en su declaración no se declara ni siquiera **void**).

El constructor no es un miembro más de una clase. Sólo es invocado cuando se crea el objeto (con el operador **new**). No puede invocarse explicitamente en ningún otro momento.

Continuando con los ejemplos del capítulo anterior se podría escribir un constructor para la clase Punto, de la siguiente forma:

```
class Punto {
    int x , y ;
    Punto ( int a , int b ) {
        x = a ; y = b ;
    }
}
```

Con este constructor se crearía un objeto de la clase Punto de la siguiente forma:

```
Punto p = new Punto ( 1 , 2 );
```

5.2. Constructor no-args.

Si una clase no declara ningún constructor, Java incorpora un constructor por defecto (denominado constructor no-args) que no recibe ningún argumento y no hace nada.

Si se declara algún constructor, entonces ya no se puede usar el constructor no-args. Es necesario usar el constructor declarado en la clase.

En el ejemplo el constructor no-args sería:

```
class Punto {
    int x , y ;
    Punto ( ) { }
}
```

5.3. Sobrecarga de constructores.

Una clase puede definir varios constructores (un objeto puede inicializarse de varias formas). Para cada instanciación se usa el que coincide en número y tipo de argumentos. Si no hay ninguno coincidente se produce un error en tiempo de compilación.

Por ejemplo:

```
class Punto {
    int x , y ;
    Punto ( int a , int b ) {
        x = a ; y = b ;
    }
    Punto () {
        x = 0 ; y = 0;
    }
}
```

En el ejemplo se definen dos constructores. El citado en el ejemplo anterior y un segundo que no recibe argumentos e inicializa las variables miembro a 0. (Nota: veremos más adelante que este tipo de inicialización es innecesario, pero para nuestro ejemplo sirve).

Desde un constructor puede invocarse explicitamente a otro constructor utilizando la palabra reservada **`this`** . **`this`** es una referencia al propio objeto. Cuando **`this`** es seguido por paréntesis se entiende que se está invocando al constructor del objeto en cuestión. Puedes ver el uso más habitual de this aquí. El ejemplo anterior puede reescribirse de la siguiente forma:

```
class Punto {
    int x , y ;
    Punto ( int a , int b ) {
```

```
            x = a ; y = b ;
        }
        Punto () {
            this (0,0);
        }
    }
```

Cuando se declaran varios constructores para una misma clase estos deben distinguirse en la lista de argumentos, bien en el número, bien en el tipo.

Esta característica de definir métodos con el mismo nombre se denomina sobrecarga y es aplicable a cualquier método miembro de una clase como veremos más adelante.

Ultima actualización - 18-Febrero-2001

Antonio Bel Puchol - abelp@arrakis.es

Indice Siguiente Anterior

Clases - Miembros estáticos

Datos estáticos

Un dato estático es una variable miembro que no se asocia a un objeto (instancia) de una clase, sino que se asocia a la clase misma; no hay una copia del dato para cada objeto sino una sola copia que es compartida por todos los objetos de la clase.

Por ejemplo:

```
class Punto {
    int x , y ;
    static int numPuntos = 0;

    Punto ( int a , int b ) {
        x = a ; y = b;
        numPuntos ++ ;
    }
}
```

En el ejemplo `numPuntos` es un contador que se incrementa cada vez que se crea una instancia de la clase Punto.

Observese la forma en que se declara `numPuntos`, colocando el modificador **`static`** delante del tipo. La sintaxis general para declarar una variable es:

[*modificadores*] *tipo_variable nombre****;***

`static` es un modificador. En los siguientes capítulos se irán viendo otros modificadores. Los [ ] en la expresión anterior quieren decir que los modificadores son opcionales.

El acceso a las variables estáticas desde fuera de la clase donde se definen se raliza a través del nombre de la clase y no del nombre del objeto como sucede con las variables miembro normales (no estáticas). En el ejemplo anterior puede escribirse:

```
int x = Punto.numPuntos;
```

No obstante también es posible acceder a las variables estáticas a través de una referencia a un objeto de la clase. Por ejemplo:

```
Punto p = new Punto();
int x = p.numPuntos;
```

Las variables estáticas de una clase existen, se inicializan y pueden usarse antes de que se cree ningún objeto de la clase.

Métodos estáticos

Para los métodos, la idea es la misma que para los datos: los métodos estáticos se asocian a una clase, no a una instancia.

Por ejemplo:

```
class Punto {
    int x , y ;
    static int numPuntos = 0;

    Punto ( int a , int b ) {
        x = a ; y = b;
        numPuntos ++ ;
    }

    static int cuantosPuntos() {
        return numPuntos;
    }
}
```

La sintaxis general para la definición de los métodos es, por tanto, la siguiente:

[modificadores] *Tipo_Valor_devuelto nombre_método* **(**
lista_argumentos **) {**
    bloque_de_codigo;
}

El aceso a los métodos estáticos se hace igual que a los datos estáticos, es decir, usando el nombre de la clase, en lugar de usar una referencia:

```
int totalPuntos = Punto.cuantosPuntos();
```

Dado que los métodos estáticos tienen sentido a nivel de clase y no a nivel de objeto (instancia) los métodos estáticos no pueden acceder a datos miembros que no sean estáticos.

El método `main`

Un programa Java se inicia proporcionando al intérprete Java un nombre de clase. La JVM carga en memoria la clase indicada e inicia su ejecución por un método estático que debe estar codificado en esa clase. El nombre de este método es `main` y debe declararse de la siguiente forma:

```
static void main ( String [] args)
```

- Es un método estático. Se aplica por tanto a la clase y no a una instancia en particular, lo que es conveniente puesto que en el momento de iniciar la ejecución todavía no se ha creado ninguna instancia de ninguna clase.
- Recibe un argumento de tipo `String [].` String es una clase que representa una cadena de caracteres (se verá más adelante),
- Los corchetes `[]` indican que se trata de un array que se verán en un capítulo posterior.

No es obligatorio que todas las clases declaren un método `main` . Sólo aquellos métodos que vayan a ser invocados directamente desde la línea de comandos de la JVM necesitan tenerlo. En la práctica la mayor parte de las clases no lo tienen.

Inicializadores estáticos

En ocasiones es necesario escribir código para inicializar los datos estáticos, quizá creando algún otro objeto de alguna clase o realizando algún tipo de control. El fragmento de código que realiza esta tarea se llama inicializador estático. Es un bloque de sentencias que no tiene nombre, ni recibe argumentos, ni devuelve valor. Simplemente se declara con el modificador **`static`** .

La forma de declarar el inicializador estático es:

`static` { *bloque_codigo* }

Por ejemplo:

```
class Punto {
    int x , y ;
    static int numPuntos;

    static {
        numPuntos = 0;
    }

    Punto ( int a , int b ) {
        x = a ; y = b;
        numPuntos ++ ;
```

```
        }

        static int cuantosPuntos() {
            return numPuntos;
        }
    }
```

Nota: El ejemplo, una vez más, muestra sólo la sintaxis y forma de codificación. Es innecesario inicializar la variable tal como se verá más adelante. Además podría inicializarse directamente con:
`static int numPuntos = 0;`

Indice Siguiente Anterior

7. Clases - Otros aspectos

7.1. Inicialización de variables

Desde el punto de vista del lugar donde se declaran existen dos tipos de variables:

- Variables miembro: Se declaran en una clase, fuera de cualquier método.
- Variables locales: Se declaran y usan en un bloque de código dentro de un método.

Las variables miembro son inicializadas automáticamente, de la siguiente forma:

- Las numéricas a 0.
- Las booleanas a **`false`**.
- Las char al caracter nulo (hexadecimal 0).
- Las referencias a **`null`**.

Nota: **`null`**es un literal que indica referencia nula.

Las variables miembro pueden inicializarse con valores distintos de los anteriores en su declaración.

Las variables locales no se inicializan automáticamente. Se debe asignarles un valor antes de ser usadas. Si el compilador detecta una variable local que se usa antes de que se le asigne un valor produce un error. Por ejemplo:

```
int p;
int q = p;    // error
```

El compilador también produce un error si se intenta usar una variable local que podría no haberse inicializado, dependiendo del flujo de ejecución del programa. Por ejemplo:

```
int p;
if (. . . ) {
     p = 5 ;
}
int q = p;    // error
```

El compilador produce un error del tipo 'La variable podría no haber sido inicializada', independientemente de la condición del if.

7.2. Ambito de las variables

El ámbito de una variable es el área del programa donde la variable existe y puede ser utilizada. Fuera de ese ámbito la variable, o bien no existe o no puede ser usada (que viene a ser lo mismo).

El ámbito de una variable miembro (que pertenece a un objeto) es el de la usabilidad de un objeto. Un objeto es utilizable desde el momento en que se crea y mientras existe una referencia que apunte a él. Cuando la última referencia que lo apunta sale de su ámbito el objeto queda 'perdido' y el espacio de memoria ocupado por el objeto puede ser recuperado por la JVM cuando lo considere oportuno. Esta recuperación de espacio en memoria se denomina 'recogida de basura' y es descrita un poco más adelante.

El ámbito de las variables locales es el bloque de código donde se declaran. Fuera de ese bloque la variable es desconocida.

Ejemplo:

```
{
    int x;     // empieza el ámbito de x. (x es
conocida y utilizable)
    {
        int q;    // empieza el ámbito de q. x
sigue siendo conocida.
        . . .
    }            // finaliza el ámbito de q
(termina el bloque de código)
     . . .        // q ya no es utilizable
}               // finaliza el ámbito de x
```

7.3. Recogida de basura

Cuando ya no se necesita un objeto simplemente puede dejar de referenciarse. No existe una operación explícita para 'destruir' un objeto o liberar el área de memoria usada por él.

La liberación de memoria la realiza el recolector de basura (*garbage collector*) que es una función de la JVM. El recolector revisa toda el área de memoria del programa y determina que objetos pueden ser borrados porque ya no tienen referencias activas que los apunten. El recolector de basura actua cuando la JVM lo determina (tiene un mecanismo de actuación no trivial).

En ocasiones es necesario realizar alguna acción asociada a la acción de liberar la memoria asignada al objeto (como por ejemplo liberar otros recursos del sistema, como descriptores de ficheros). Esto puede hacerse codificando un método `finalize` que debe declararse como:

```
protected void finalize() throws Throwable { }
```

Nota: las clausulas **`protected`** y **`throws`** se explican en capítulos posteriores.

El método `finalize` es invocando por la JVM antes de liberar la memoria por el recolector de basura, o antes de terminar la JVM. No existe un momento concreto en que las áreas de memoria son liberadas, sino que lo determina en cada momento la JVM en función de sus necesidades de espacio.

7.4. Sobrecarga de métodos

Una misma clase puede tener varios métodos con el mismo nombre siempre que se diferencien en el tipo o número de los argurmentos. Cuando esto sucede se dice que el método está sobrecargado. Por ejemplo, una misma clase podría tener los métodos:

```
int metodoSobrecargado() { . . .}
int metodoSobrecargado(int x) { . . .}
```

Sin embargo no se puede sobrecargar cambiando sólo el tipo del valor devuelto. Por ejemplo:

```
int metodoSobrecargado() { . . .}
void metodoSobrecargado() { . . .}  // error en
compilación
```

con esta definición, en la expresión `y.metodoSobrecargado()` la JVM no sabría que método invocar.

Se puede sobrecargar cualquier método miembro de una clase, así como el constructor.

7.5. La referencia `this`

En ocasiones es conveniente disponer de una referencia que apunte al propio objeto que se está

manipulando. Esto se consigue con la palabra reservada **this**. **this** es una referencia implicita que tienen todos los objetos y que apunta a si mismo. Por ejemplo:

```
class Circulo {
    Punto centro;
    int radio;
    . . .
    Circulo elMayor(Circulo c) {
        if (radio > c.radio) return this;
        else return c;
    }
}
```

El método elMayor devuelve una referencia al círculo que tiene mayor radio, comparando los radios del Circulo c que se recibe como argumento y el propio. En caso de que el propio resulte mayor el método debe devolver una referencia a si mismo. Esto se consigue con la expresión **return this.**

7.6. La referencia null

Para asignar a una referencia el valor nulo se utiliza la constante null. El ejemplo del caso anterior se podría completar con:

```
class Circulo {
    Punto centro;
    int radio;
    . . .
    Circulo elMayor(Circulo c) {
        if (radio > c.radio) return this;
        else if (c.radio > radio) return c;
              else return null;
    }
}
```

7.7. Ocultamiento de variables

Puede ocurrir que una variable local y una variable miembro reciban el mismo nombre (en muchos casos por error). Cuando se produce esto la variable miembro queda oculta por la variable local, durante el bloque de código en que la variable local existe y es accesible. Cuando se sale fuera del ámbito de la variable local, entonces la variable miembro queda accesible. Observese esto en el ejemplo siguiente:

```
. . .
String x = "Variable miembro";
. . .
```

```
void variableOculta() {
    System.out.println(x);
    {
        String x = "Variable local";
        System.out.println(x);
    }
    System.out.println(x);
}
```

Nota: El uso de Strings se verá en un capítulo posterior, aunque su uso aquí resulta bastante intuitivo. La llamada `System.out.println` envia a la consola (la salida estándar habitual) las variables que se pasan como argumentos.

La llamada al método `variableOculta()` producirá la siguiente salida:

```
Variable miembro
Variable local
Variable miembro
```

Se puede acceder a la variable miembro oculta usando la referencia this. En el ejemplo anterior la expresión:

```
System.out.println(this.x);
```

siempre producirá la salida '`Variable miembro`', puesto que `this.x` se refiere siempre a la variable miembro.

Ultima actualización - 25-Marzo-2001

Antonio Bel Puchol - abelp@arrakis.es

8. Control de la ejecución

8.1. Resumen de operadores

La siguiente tabla muestra un resumen de operadores clasificados por grupos:

Grupo de operador	Operador	Significado
Aritméticos	+ - * / %	Suma Resta Multiplicación División Resto
Relacionales	> >= < <= == !=	Mayor Mayor o igual Menor Menor o igual Igual Distinto
Logicos	&& || !	AND OR NOT

A nivel de bits	`&`
	`|`
	`^`
	`<<`
	`>>`
	`>>>`
		AND
		OR
		NOT
		Desplazamiento a la izquierda
		Desplazamiento a la derecha rellenando con 1
		Desplazamiento a la derecha rellenando con 0

Otros	`+`
	`++`
	`--`
	`=`
	`+=`
	`-=`
	`*=`
	`/=`
	`?:`
		Concatenación de cadenas
		Autoincremento (actua como prefijo o sufijo)
		Autodecremento (actua como preficjo o sufijo)
		Asignación
		Suma y asignación
		Resta y asignación
		Multiplicación y asignación
		División y asignación
		Condicional

8.2. Ejecución condicional

El formato general es:

```
if (expresion_booleana)
    sentencia
[else
    sentencia]
```

sentencia (a todo lo largo de este capítulo) puede ser una sola sentencia o un bloque de sentencias separadas por `;` y enmarcadas por llaves `{` y `}`. Es decir

```
if (expresion_booleana) {
    sentencia;
    sentencia;
    . . .
}
else {
    sentencia;
    sentencia;
    . . .
}
```

expresion_booleana es una expresión que se evalua como **`true`** o **`false`** (es decir, de tipo

booleano). Si el resultado es true la ejecución bifurca a la sentencia que sigue al **`if`** . En caso contrario se bifurca a la sentencia que sigue al **`else`**.

Los corchetes en el formato anterior indican que la clausula **`else`** es opcional.

8.3. Iteraciones con `while`

Sintaxis formato 1:

```
while (expresion_booleana)
    sentencia
```

Sintaxis formato 2:

```
do
    sentencia
while (expresion_booleana)
```

La sentencia o bloque se sentencias (se aplica la misma idea que para el `if-else`) se ejecuta mientras que la *`expresion_booleana`* se evalue como **`true`**

La diferencia entre ambos formatos es que en el primero la expresión se evalua al principio del bloque de sentencias y en el segundo se evalua al final.

8.4. Iteraciones con `for`

El formato es:

```
for ( inicializacion ; expresion_booleana ; step
)
    sentencia
```

`inicializacion` es una sentencia que se ejecuta la primera vez que se entra en el bucle **`for`** . Normalmente es una asignación. Es opcional.

`expresion_booleana` es una expresión que se evalua antes de la ejecución de la sentencia, o bloque de sentencias, para cada iteración. La sentencia o bloque de sentencias se ejecutan mientras que la *`expresion_booleana`* se evalue como cierta. Es opcional.

`step` es una sentencia que se ejecuta cada vez que se llega al final de la sentencia o bloque de sentencias. Es opcional.

Una utilización clásica de un bucle de tipo for se muestra a continuación para evaluar un contador un

número fijo de veces:

```
for ( int i = 1 ; i <= 10 ; i++ )
    sentencia
```

La sentencia (o bloque de sentencias) se evaluará 10 veces. En cada ejecución (pasada) el valor de la variable i irá variando desde 1 hasta 10 (inclusive). Cuando salga del bloque de sentencias i estará fuera de su ámbito (porque se define en el bloque for).

Si se omiten las tres clausulas del bucle se obtiene un bucle infinito:

```
for ( ; ; )
    sentencia
```

Obsérvese que se pueden omitir las clausulas pero no los separadores (;).

8.5. Evaluación múltiple

El formato es:

```
switch ( expresion_entera ) {
    case valor_entero:
        sentencia;
        break;
    case valor_entero:
        sentencia;
        break;
        . . .
    default:
        sentencia;
}
```

Cuidado: en el **`switch`** la expresión que se evalua no es una expresión booleana como en el `if-else`, sino una expresión entera.

Se ejecuta el bloque **`case`** cuyo valor coincida con el resultado de la expresión entera de la clausula **`switch`** . Se ejecuta hasta que se encuentra una sentencia **`break`** o se llega al final del **`switch`** .

Si ningún valor de **`case`** coincide con el resultado de la expresión entera se ejecuta el bloque **`default`**(si está presente).

`default` y **`break`** son opcionales.

8.6. Devolución de control

El formato es:

```
return valor
```

Se utiliza en los métodos para terminar la ejecución y devolver un valor a quien lo llamó.

valor debe ser del tipo declarado en el método.

valor es opcional. No debe existir cuando el método se declara de tipo **void**. En este caso, la claúsula **return** al final del método es opcional, pero puede usarse para devolver el control al llamador en cualquier momento.

8.7. Expresiones

La mayor parte del trabajo en un programa se hace mediante la evaluación de expresiones, bien por sus efectos tales como asignaciones a variables, bien por sus valores, que pueden ser usados como argumentos u operandos en expresiones mayores, o afectar a la secuencia de ejecución de instrucciones.

Cuando se evalua una expresión en un programa el resultado puede denotar una de tres cosas:

- Una variable. (Si por ejemplo es una asignación)
- Un valor. (Por ejemplo una expresión aritmética, booleana, una llamada a un método, etc.)
- Nada. (Por ejemplo una llamada a un método declarado void)

Si la expresión denota una variable o un valor, entonces la expresión tiene siempre un tipo conocido en el momento de la compilación. Las reglas para determinar el tipo de la expresión varían dependiendo de la forma de las expresiones pero resultan bastante naturales. Por ejemplo, en una expresión aritmética con operandos de diversas precisiones el resultado es de un tipo tal que no se produzca pérdida de información, realizandose internamente las conversiones necesarias. El análisis pormenorizado de las conversiones de tipos, evaluaciones de expresiones, etc, queda fuera del ámbito de estos apuntes. En general puede decirse que es bastante similar a otros lenguajes, en particular C, teniendo en cuenta la característica primordial de Java de tratarse de un lenguaje con control fuerte de tipos.

Ultima actualización - 25-Marzo-2001

Antonio Bel Puchol - abelp@arrakis.es

9. Arrays

9.1. Declaración y acceso

Un array es una colección ordenada de elementos del mismo tipo, que son accesibles a través de un índice.

Un array puede contener datos primitivos o referencias a objetos.

Los arrays se declaran:

[*modificadores*] *tipo_variable [ ] nombre**;***

Por ejemplo:

```
int [ ] a;
Punto [ ] p;
```

La declaración dice que a es un array de enteros y p un array de objetos de tipo Punto. Más exactamente a es una referencia a una colección de enteros, aunque todavía no se sabe cuantos elementos tiene el array. p es una referencia a una colección de referencias que apuntarán objetos Punto.

Un array se crea como si se tratara de un objeto (de hecho las variables de tipo array son referencias):

```
a = new int [5];
p = new Punto[3];
```

En el momento de la creación del array se dimensiona el mismo y se reserva la memoria necesaria.

También puede crearse de forma explícita asignando valores a todos los elementos del array en el momento de la declaración, de la siguiente forma:

```
int [ ] a = { 5 , 3 , 2 };
```

El acceso a los elementos del array se realiza indicando entre corchetes el elemento del array que se desea, teniendo en cuenta que siempre el primer elemento del array es el índice 0. Por ejemplo `a[1].` En este ejemplo los índices del array de tres elementos son 0, 1 y 2. Si se intenta usar un índice que está fuera del rango válido para ese array se produce un error (en realidad una excepción. Las excepciones se tratan en un capítulo posterior) de 'Indice fuera de rango'. En el ejemplo anterior se produce esta excepción si el índice es menor que 0 o mayor que 2.

Se puede conocer el número de elementos de un array usando la variable `length`. En el ejemplo `a.length` contiene el valor 3.

Un array, como cualquier otra referencia puede formar parte de la lista de parámetros o constituir el valor de retorno de un método. En ambos casos se indica que se trata de un array con los corchetes que siguen al tipo. Por ejemplo:

```
String [ ] metodoConArrays ( Punto [ ] ) { . . }
```

El método metodoConArrays recibe como parámetro un array de Puntos y devuelve un array de Strings. El método podría invocarse de la siguiente forma:

```
Punto p [ ] = new Punto [3];
. . .
String [ ] resultado = metodoConArrays(p);
```

9.2. Arrays multidimensionales

Es posible declarar arrays de más de una dimensión. Los conceptos son los mismos que para los arrays monodimensionales.

Por ejemplo:

```
int [ ][ ] a = { { 1 , 2 } , { 3 , 4 } , { 5 , 6
} };
int x = a[1][0];  // contiene 3
int y = a[2][1];  // contiene 6
```

Se pueden recorrer los elementos de un array multidimensional, de la siguiente forma:

```
int [ ][ ] a = new int [3][2];
for ( int i = 0 ; i < a.length ; i++ )
    for ( int j = 0 ; j < a[i].length ; j++)
        a[i][j] = i * j;
```

Obsérvese en el ejemplo la forma de acceder al tamaño de cada dimensión del array.

Ultima actualización - 25-Marzo-2001

Antonio Bel Puchol - abelp@arrakis.es

10. Strings

10.1. La clase String

En Java no existe un tipo de datos primitivo que sirva para la manipulación de cadenas de caracteres. En su lugar se utiliza una clase definida en la API que es la clase String. Esto significa que en Java las cadenas de caracteres son, a todos los efectos, objetos que se manipulan como tales, aunque existen ciertas operaciones, como la creación de Strings, para los que el lenguaje tiene soporte directo, con lo que se simplifican algunas operaciones.

La clase String forma parte del package java.lang y se describe completamente en la documentación del API del JDK.

10.2. Creación de Strings

Un String puede crearse como se crea cualquier otro objeto de cualquier clase; mediante el operador new:

```
String s = new String("Esto es una cadena de
caracteres");
```

Observese que los literales de cadena de caracteres se indican entre comillas dobles ("), a diferencia de los caracteres, que utilizan comillas simples (').

Sin embargo también es posible crear un String directamente, sin usar el operador new, haciendo una asignación simple (como si se tratara de un dato primitivo):

```
String s = "Esto es una cadena de caracteres";
```

Ambas expresiones conducen al mismo objeto.

Los Strings no se modifican una vez que se les ha asignado valor. Si se produce una reasignación se crea un nuevo objeto String con el nuevo contenido.

Además la clase String proporciona constructores para crear Strings a partir de arrays de caracteres y arrays de bytes. Consultar la documentación del API del JDK para más detalles.

10.3. Concatenación de Strings

Java define el operador + (suma) con un significado especial cuando las operandos son de tipo String. En este caso el operador suma significa concatenación. El resultado de la concatenación es un nuevo String compuesto por las dos cadenas, una tras otra. Por ejemplo:

```
String x = "Concatenar" + "Cadenas";
```

da como resultado el String `"ConcatenarCadenas".`

También es posible concatenar a un String datos primitivos, tanto numéricos como booleanos y char. Por ejemplo, se puede usar:

```
int i = 5;
String x = "El valor de i es " + i;
```

Cuando se usa el operador + y una de las variables de la expresión es un String, Java transforma la otra variable (si es de tipo primitivo) en un String y las concatena. Si la otra variable es una referencia a un objeto entonces invoca el método toString() que existe en todas las clases (es un método de la clase Object).

10.4. Otros métodos de la clase String

La clase String dispone de una amplia gama de métodos para la manipulación de las cadenas de caracteres. Para una referencia completa consultar la documentación del API del JDK. El siguiente cuadro muestra un resumen con algunos de los métodos más significativos:

Método	**Descripción**
`char charAt(int index)`	Devuelve el carácter en la posición indicada por index. El rango de index va de 0 a length() - 1.

`boolean equals(Object obj)`	Compara el String con el objeto especificado. El resultado es true si y solo si el argumento es no nulo y es un objeto String que contiene la misma secuencia de caracteres.
`boolean equalsIgnoreCase(String s)`	Compara el String con otro, ignorando consideraciones de mayúsculas y minúsculas. Los dos Strings se consideran iguales si tienen la misma longitud y, los caracteres correspondientes en ambos Strings son iguales sin tener en cuenta mayúsculas y minúsculas.
`int indexOf(char c)`	Devuelve el indice donde se produce la primera aparición de c. Devuelve -1 si c no está en el string.
`int indexOf(String s)`	Igual que el anterior pero buscando la subcadena representada por s.
`int length()`	Devuelve la longitud del String (número de caracteres)
`String substring(int begin, int end)`	Devuelve un substring desde el índice begin hasta el end
`static String valueOf(int i)`	Devuelve un string que es la representación del entero i. Observese que este método es estático. Hay métodos equivalentes donde el argumento es un float, double, etc.

`char[] toCharArray()`
`String toLowerCase()`
`String toUpperCase()`
	Transforman el string en un array de caracteres, o a mayúsculas o a minúsculas.

10.5. La clase StringBuffer

Dado que la clase String sólo manipula cadenas de caracteres constantes resulta poco conveniente cuando se precisa manipular intensivamente cadenas (reemplazadno caracteres, añadiendo o suprimiendo, etc.). Cuando esto es necesario puede usarse la clase StringBuffer definida también en el package java.lang. del API. Esta clase implanta un buffer dinámico y tiene métodos que permiten su manipulación comodamente. Ver la documentación del API.

Ultima actualización - 25-Marzo-2001

Antonio Bel Puchol - abelp@arrakis.es

---

11. Packages

11.1 Claúsula package

Un package es una agrupación de clases afines. Equivale al concepto de librería existente en otros lenguajes o sistemas. Una clase puede definirse como perteneciente a un package y puede usar otras clases definidas en ese o en otros packages.

Los packages delimitan el espacio de nombres (space name). El nombre de una clase debe ser único dentro del package donde se define. Dos clases con el mismo nombre en dos packages distintos pueden coexistir e incluso pueden ser usadas en el mismo programa.

Una clase se declara perteneciente a un package con la clausula `package`, cuya sintaxis es:

```
package nombre_package;
```

La clausula **`package`** debe ser la primera sentencia del archivo fuente. Cualquier clase declarada en ese archivo pertenece al package indicado.

Por ejemplo, un archivo que contenga las sentencias:

```
package miPackage;
. . .
class miClase {
. . .
```

declara que la clase miClase pertenece al package miPackage.

La claúsula package es opcional. Si no se utiliza, las clases declaradas en el archivo fuente no pertenecen a ningún package concreto, sino que pertenecen a un package por defecto sin nombre.

La agrupación de clases en packages es conveniente desde el punto de vista organizativo, para mantener bajo una ubicación común clases relacionadas que cooperan desde algún punto de vista. También resulta importante por la implicación que los packages tienen en los modificadores de acceso, que se explican en un capítulo posterior.

11.2 Claúsula `import`

Cuando se referencia cualquier clase dentro de otra se asume, si no se indica otra cosa, que ésta otra está declarada en el mismo package. Por ejemplo:

```
package Geometria;
. . .
class Circulo {
    Punto centro;
    . . .
}
```

En esta declaración definimos la clase Circulo perteneciente al package Geometria. Esta clase usa la clase Punto. El compilador y la JVM asumen que Punto pertenece también al package Geometria, y tal como está hecha la definición, para que la clase Punto sea accesible (conocida) por el compilador, es necesario que esté definida en el mismo package.

Si esto no es así, es necesario hacer accesible el espacio de nombres donde está definida la clase Punto a nuestra nueva clase. Esto se hace con la clausula import. Supongamos que la clase Punto estuviera definida de esta forma:

```
package GeometriaBase;
class Punto {
    int x , y;
}
```

Entonces, para usar la clase Punto en nuestra clase Circulo deberiamos poner:

```
package GeometriaAmpliada;

import GeometriaBase.*;

class Circulo {
    Punto centro;
    . . .
}
```

Con la claúsula **`import GeometriaBase.*;`** se hacen accesibles todos los nombres (todas las

clases) declaradas en el package GeometriaBase. Si sólo se quisiera tener accesible la clase Punto se podría declarar: **`import`** `GeometriaBase.Punto;`

También es posible hacer accesibles los nombres de un package sin usar la clausula **`import`** calificando completamente los nombres de aquellas clases pertenecientes a otros packages. Por ejemplo:

```
package GeometriaAmpliada;

class Circulo {
    GeometriaBase.Punto centro;
    . . .
}
```

Sin embargo si no se usa **`import`** es necesario especificar el nombre del package cada vez que se usa el nombre Punto.

La claúsula **`import`** simplemente indica al compilador donde debe buscar clases adicionales, cuando no pueda encontrarlas en el package actual y delimita los espacios de nombres y modificadores de acceso. Sin embargo, no tiene la implicación de 'importar' o copiar código fuente u objeto alguno. En una clase puede haber tantas sentencias **`import`** como sean necesarias. Las cláusulas **`import`** se colocan después de la cláusula **`package`** (si es que existe) y antes de las definiciones de las clases.

11.3. Nombres de los packages

Los packages se pueden nombrar usando nombres compuestos separados por puntos, de forma similar a como se componen las direcciones URL de Internet. Por ejemplo se puede tener un package de nombre `misPackages.Geometria.Base.` Cuando se utiliza esta estructura se habla de packages y subpackages. En el ejemplo `misPackages` es el Package base, `Geometria` es un subpackage de `misPackages` y `Base` es un subpackage de `Geometria`.

De esta forma se pueden tener los packages ordenados según una jerarquía equivalente a un sistema de archivos jerárquico.

El API de java está estructurado de esta forma, con un primer calificador (java o javax) que indica la base, un segundo calificador (awt, util, swing, etc.) que indica el grupo funcional de clases y opcionalmente subpackages en un tercer nivel, dependiendo de la amplitud del grupo. Cuando se crean packages de usuario no es recomendable usar nombres de packages que empiecen por java o javax.

11.4. Ubicación de packages en el sistema de archivos

Además del significado lógico descrito hasta ahora, los packages también tienen un significado físico que sirve para almacenar los módulos ejecutables (ficheros con extensión .class) en el sistema de archivos del ordenador.

Supongamos que definimos una clase de nombre `miClase` que pertenece a un package de nombre `misPackages.Geometria.Base.` Cuando la JVM vaya a cargar en memoria `miClase` buscará el módulo ejecutable (de nombre miClase.class) en un directorio en la ruta de acceso misPackages/Geometria/Base. Está ruta deberá existir y estar accesible a la JVM para que encuentre las clases. En el capítulo siguiente se dan detalles sobre compilación y ejecución de programas usando el compilador y la máquina virtural distribuida por SUN Microsystems (JDK).

Si una clase no pertenece a ningún package (no existe clausula **`package`**) se asume que pertenece a un package por defecto sin nombre, y la JVM buscará el archivo .class en el directorio actual.

Para que una clase pueda ser usada fuera del package donde se definió debe ser declarada con el modificador de acceso public, de la siguiente forma:

```
package GeometriaBase;

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

Nota: Los modificadores de acceso se explicarán detalladamente en un capítulo posterior.

Si una clase no se declara **`public`** sólo puede ser usada por clases que pertenezcan al mismo package.

Ultima actualización - 17-Junio-2001

Antonio Bel Puchol - abelp@arrakis.es

12. Compilación y ejecución de programas

En este apartado se asume que se ha instalado el JDK (J2SE) distribuido por SUN Microsystems y que tanto el compilador (javac) como la JVM (java) están accesibles. Asumiremos que los comandos se emitirán desde una ventana DOS en un sistema Windows, siendo la sintaxis en un entorno UNIX muy parecida. En este capítulo se verán todos los pasos necesarios para crear, compilar y ejecutar un programa Java.

12.1. Creación y Compilación de un programa Java

PASO 1: Con un editor de texto simple (incluso notepad sirve, aunque resulta poco aconsejabe) creamos un archivo con el contenido siguiente:

```
package Programas.Ejemplo1;

class HolaMundo {
    public static void main ( String [] args) {
        System.out.println("Hola a todos");
    }
}
```

Guardamos el fichero fuente con nombre `HolaMundo.java` en la carpeta: `C:\ApuntesJava\Programas\Ejemplo1.`

PASO 2: Abrimos una ventana DOS y en ella:

```
C:> cd C:\ApuntesJava
C:\ApuntesJava>javac
```

```
Programas\Ejemplo1\HolaMundo.java
```

Si no hay ningún error en el programa se producirá la compilación y el compilador almacenará en el directorio `C:\ApuntesJava\Programas\Ejemplo1` un fichero de nombre `HolaMundo.class,` con el código ejecutable correspondiente a la clase HolaMundo.

Recuerda que en Java las mayúsculas y minúsculas son significativas. No es lo mismo la clase ejemplo1 que la clase Ejemplo1. Esto suele ser fuente de errores, sobre todo al principio. Sin embargo, ten en cuenta que en algunos sistemas operativos como Windows, o más concretamente en una ventana DOS, esta distinción no existe. Puedes poner `cd C:\ApuntesJava o cd C:\APUNTESJAVA` indistintamente: el resultado será el mismo (no así en cualquier UNIX, que sí distingue unas y otras). Asegurate por tanto, de que las palabras están correctamente escritas.

Cuando pones `javac Programas\Ejemplo1\HolaMundo.java` estás indicando al compilador que busque un archivo de nombre HolaMundo.java en la ruta Programas\Ejemplo1, a partir del directorio actual; es decir, estás especificando la ruta de un archivo.

En el ejemplo se utiliza la clase del API de Java `System`. Sin embargo el programa no tiene ningún **`import`**. No obstante el compilador no detecta ningún error y genera el código ejecutable directamente. Esto se debe a que la clase `System` está definida en el package `java.lang`, que es el único del que no es necesario hacer el **`import`**, que es hecho implicitamente por el compilador. Para cualquier clase del API, definida fuera de este package es necesario hacer el impor correspondiente.

12.2. Ejecución de un programa Java

PASO 3: Ejecutar el programa: Desde la ventana DOS.

```
C:\ApuntesJava>java Programas.Ejemplo1.HolaMundo
```

Se cargará la JVM, cargará la clase HolaMundo y llamará a su método main que producirá en la ventana DOS la salida:

```
Hola a todos
```

Los archivos .class son invocables directamente desde la línea de comandos (con la sintaxis java nombreDeClase) si tienen un método main definido tal como se vio en un capítulo anterior.

Se puede indicar a la JVM que busque las clases en rutas alternativas al directorio actual. Esto se hace con el parámetro `-classpath` (abreviadamente -cp) en la línea de comandos. Por ejemplo si el directorio actual es otro, podemos invocar el programa de ejemplo de la forma:

```
C:\Windows>java -cp C:\ApuntesJava
Programas.Ejemplo1.HolaMundo
```

Con el parámetro -cp se puede especificar diversas rutas alternativas para la búsqueda de clases separadas por ;

Cuando pones `java Programas.Ejemplo1.HolaMundo` estás indicando a la JVM que cargue y ejecute la clase HolaMundo del Package Programas, subpackage Ejemplo1. Para cumplir está orden, expresada en términos Java de clases y packages la JVM buscará el archivo HolaMundo.class en la ruta Programas\Ejemplo1 que es algo expresado en términos del sistema de archivos, y por tanto del Sistema Operativo.

12.3. Archivos fuente (`.java`) y ejecutables (`.class`)

El esquema habitual es tener un archivo fuente por clase y asignar al archivo fuente el mismo nombre que la clase con la extensión .java (el nombre .java para la extensión es obligatorio). Esto generará al compilar un archivo .class con el mismo nombre que el fuente (y que la clase). Fuentes y módulos residirán en el mismo directorio.

Lo habitual es tener uno o varios packages que compartan un esquema jerárquico de directorios en función de nuestras necesidades (packages por aplicaciones, temas, etc.)

Es posible definir más de una clase en un archivo fuente, pero sólo una de ellas podrá ser declarada `public` (es decir podrá ser utilizada fuera del package donde se define). Todas las demás clases declaradas en el fuente serán internas al package. Si hay una clase `public` entonces, obligatoriamente, el nombre del fuente tiene que coincider con el de la clase declarada como `public` . Los modificadores de acceso (public, es uno de ellos) se verán en el capítulo siguiente.

Ultima actualización - 17-Junio-2001

Antonio Bel Puchol - abelp@arrakis.es

13. Modificadores de acceso

13.1. Modificadores

Los modificadores son elementos del lenguaje que se colocan delante de la definición de variables locales, datos miembro, métodos o clases y que alteran o condicionan el significado del elemento. En capítulos anteriores se ha descrito alguno, como es el modificador **`static`** que se usa para definir datos miembros o métodos como pertenecientes a una clase, en lugar de pertenecer a una instancia. En capítulos posteriores se tratarán otros modificadores como **`final`**, **`abstract`** o **`synchronized`**. En este capítulo se presentan los modificadores de acceso, que son aquellos que permiten limitar o generalizar el acceso a los componentes de una clase o a la clase en si misma.

13.2. Modificadores de acceso

Los modificadores de acceso permiten al diseñador de una clase determinar quien accede a los datos y métodos miembros de una clase.

Los modificadores de acceso preceden a la declaración de un elemento de la clase (ya sea dato o método), de la siguiente forma:

[*modificadores*] *tipo_variable nombre;*

[*modificadores*] *tipo_devuelto nombre_Metodo (lista_Argumentos);*

Existen los siguientes modificadores de acceso:

- **`public`** - Todo el mundo puede acceder al elemento. Si es un dato miembro, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo. Si es un método todo el mundo puede invocarlo.
- **`private`** - Sólo se puede acceder al elemento desde métodos de la clase, o sólo puede

invocarse el método desde otro método de la clase.
- **protected** - Se explicará en el capítulo dedicado a la herencia.
- sin modificador - Se puede acceder al elemento desde cualquier clase del package donde se define la clase.

Pueden utilizarse estos modificadores para cualquier tipo de miembros de la clase, incluidos los constructores (con lo que se puede limitar quien puede crear instancias de la clase).

En el ejemplo los datos miembros de la clase Punto se declaran como private, y se incluyen métodos que devuelven las coordenadas del punto. De esta forma el diseñador de la clase controla el contenido de los datos que representan la clase e independiza la implementación de la interface.

```
class Punto {
    private int x , y ;
    static private int numPuntos = 0;

    Punto ( int a , int b ) {
        x = a ; y = b;
        numPuntos ++ ;
    }

    int getX() {
        return x;
    }

    int getY() {
        return y;
    }

    static int cuantosPuntos() {
        return numPuntos;
    }

}
```

Si alguien, desde una clase externa a Punto, intenta:

```
. . .
Punto p = new Punto(0,0);
p.x = 5;
. . .
```

obtendrá un error del compilador.

13.3. Modificadores de acceso para clases

Las clases en si mismas pueden declararse:

- **`public`** - Todo el mundo puede usar la clase. Se pueden crear instancias de esa clase, siempre y cuando alguno de sus constructores sea accesible.
- sin modificador - La clase puede ser usada e instanciada por clases dentro del package donde se define.

Las clases no pueden declararse ni **`protected`** , ni **`private`** .

13.4. ¿Son importantes los modificadores de acceso?

Los modificadores de acceso permiten al diseñador de clases delimitar la frontera entre lo que es accesible para los usuarios de la clase, lo que es estrictamente privado y 'no importa' a nadie más que al diseñador de la clase e incluso lo que podría llegar a importar a otros diseñadores de clases que quisieran alterar, completar o especializar el comportamiento de la clase.

Con el uso de estos modificadores se consigue uno de los principios básicos de la Programación Orientada a Objetos, que es la encapsulación: Las clases tienen un comportamiento definido para quienes las usan conformado por los elementos que tienen un acceso público, y una implementación oculta formada por los elementos privados, de la que no tienen que preocuparse los usuarios de la clase.

Los otros dos modificadores, protected y el acceso por defecto (package) complementan a los otros dos. El primero es muy importante cuando se utilizan relaciones de herencia entre las clases y el segundo establece relaciones de 'confianza' entre clases afines dentro del mismo package. Así, la pertenencia de las clases a un mismo package es algo más que una clasificación de clases por cuestiones de orden.

Cuando se diseñan clases, es importante pararse a pensar en términos de quien debe tener acceso a que. Qué cosas son parte de la implantación y deberían ocultarse (y en que grado) y que cosas forman parte de la interface y deberían ser públicas.

Ultima actualización - 17-Junio-2001

Antonio Bel Puchol - abelp@arrakis.es

Indice Siguiente Anterior

Herencia

Composición

En anteriores ejemplos se ha visto que una clase tiene datos miembro que son instancias de otras clases. Por ejemplo:

```
class Circulo {
        Punto centro;
        int radio;
        float superficie() {
                return 3.14 * radio * radio;
        }
}
```

Esta técnica en la que una clase se compone o contiene instancias de otras clases se denomina composición. Es una técnica muy habitual cuando se diseñan clases. En el ejemplo diríamos que un Circulo tiene un Punto (centro) y un radio.

Herencia

Pero además de esta técnica de composición es posible pensar en casos en los que una clase es una extensión de otra. Es decir una clase es como otra y además tiene algún tipo de característica propia que la distingue. Por ejemplo podríamos pensar en la clase Empleado y definirla como:

```
class Empleado {
    String nombre;
    int numEmpleado , sueldo;

    static private int contador = 0;

    Empleado(String nombre, int sueldo) {
        this.nombre = nombre;
        this.sueldo = sueldo;
        numEmpleado = ++contador;
    }
```

```
    public void aumentarSueldo(int porcentaje) {
        sueldo += (int)(sueldo * aumento / 100);
    }

    public String toString() {
        return "Num. empleado " + numEmpleado + "
Nombre: " + nombre +
                " Sueldo: " + sueldo;
    }
}
```

En el ejemplo el Empleado se caracteriza por un nombre (String) y por un número de empleado y sueldo (enteros). La clase define un constructor que asigna los valores de nombre y sueldo y calcula el número de empleado a partir de un contador (variable estática que siempre irá aumentando), y dos métodos, uno para calcular el nuevo sueldo cuando se produce un aumento de sueldo (método aumentarSueldo) y un segundo que devuelve una representación de los datos del empleado en un String.(método toString).

Con esta representación podemos pensar en otra clase que reuna todas las características de Empleado y añada alguna propia. Por ejemplo, la clase Ejecutivo. A los objetos de esta clase se les podría aplicar todos los datos y métodos de la clase Empleado y añadir algunos, como por ejemplo el hecho de que un Ejecutivo tiene un presupuesto.

Así diriamos que la clase Ejecutivo extiende o hereda la clase Empleado. Esto en Java se hace con la clausula **`extends`** que se incorpora en la definición de la clase, de la siguiente forma:

```
class Ejecutivo extends Empleado {
    int presupuesto;
    void asignarPresupuesto(int p) {
        presupuesto = p;
    }
}
```

Con esta definición un Ejecutivo es un Empleado que además tiene algún rasgo distintivo propio. El cuerpo de la clase Ejecutivo incorpora sólo los miembros que son específicos de esta clase, pero implícitamente tiene todo lo que tiene la clase Empleado.

A Empleado se le llama clase base o superclase y a Ejecutivo clase derivada o subclase.

Los objetos de las clases derivadas se crean igual que los de la clase base y pueden acceder tanto sus datos y métodos como a los de la clase base. Por ejemplo:

```
Ejecutivo jefe = new Ejecutivo( "Armando Mucho",
1000);
jefe.asignarPresupuesto(1500);
```

```
jefe.aumentarSueldo(5);
```

Nota: La discusión acerca de los constructores la veremos un poco más adelante.

Atención!: Un Ejecutivo ES un Empleado, pero lo contrario no es cierto. Si escribimos:

```
Empleado curri = new Empleado ( "Esteban Comex
Plota" , 100) ;
curri.asignarPresupuesto(5000);  // error
```

se producirá un error de compilación pues en la clase Empleado no existe ningún método llamado asignarPresupuesto.

Redefinición de métodos. El uso de super.

Además se podría pensar en redefinir algunos métodos de la clase base pero haciendo que métodos con el mismo nombre y características se comporten de forma distinta. Por ejemplo podríamos pensar en rediseñar el método toString de la clase Empleado añadiendo las características propias de la clase Ejecutivo. Así se podría poner:

```
class Ejecutivo extends Empleado {
    int presupuesto;

    void asignarPresupuesto(int p) {
        presupuesto = p;
    }

    public String toString() {
        String s = super.toString();
        s = s + " Presupuesto: " + presupuesto;
        return s;
    }
}
```

De esta forma cuando se invoque `jefe.toString()` se usará el método toString de la clase Ejecutivo en lugar del existente en la clase Empleado.

Observese en el ejemplo el uso de **`super`**, que representa referencia interna implícita a la clase base (superclase). Mediante **`super`**`.toString()` se invoca el método toString de la clase Empleado

Inicialización de clases derivadas

Cuando se crea un objeto de una clase derivada se crea implicitamente un objeto de la clase base que se inicializa con su constructor correspondiente. Si en la creación del objeto se usa el constructor no-

args, entonces se produce una llamada implicita al constructor no-args para la clase base. Pero si se usan otros constructores es necesario invocarlos explicitamente.

En nuestro ejemplo dado que la clase método define un constructor, necesitaremos también un constructor para la clase Ejecutivo, que podemos completar así:

```
class Ejecutivo extends Empleado {
    int presupuesto;

    Ejecutivo (String n, int s) {
        super(n,s);
    }

    void asignarPresupuesto(int p) {
        presupuesto = p;
    }

    public String toString() {
        String s = super.toString();
        s = s + " Presupuesto: " + presupuesto;
        return s;
    }
}
```

Observese que el constructor de Ejecutivo invoca directamente al constructor de Empleado mediante `super(`*`argumentos`*`)`. En caso de resultar necesaria la invocación al constructor de la superclase debe ser la primera sentencia del constructor de la subclase.

Indice Siguiente Anterior

Indice Siguiente Anterior

Herencia - II

El modificador de acceso `protected`

El modificador de acceso `protected` es una combinación de los accesos que proporcionan los modificadores `public` y `private`. `protected` proporciona acceso público para las clases derivadas y acceso privado (prohibido) para el resto de clases.

Por ejemplo, si en la clase Empleado definimos:

```
class Empleado {
    protected int sueldo;
    . . .
}
```

entonces desde la clase Ejecutivo se puede acceder al dato miembro sueldo, mientras que si se declara private no.

Up-casting y Down-casting

Siguiendo con el ejemplo de los apartados anteriores, dado que un Ejecutivo ES un Empleado se puede escribir la sentencia:

```
Empleado emp = new Ejecutivo("Máximo Dueño" ,
2000);
```

Aquí se crea un objeto de la clase Ejecutivo que se asigna a una referencia de tipo Empleado. Esto es posible y no da error ni al compilar ni al ejecutar porque Ejecutivo es una clase derivada de Empleado. A esta operación en que un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases se denomina 'upcasting'.

Cuando se realiza este tipo de operaciones, hay que tener cuidado porque para la referencia emp no existen los miembros de la clase Ejecutivo, aunque la referencia apunte a un objeto de este tipo. Así, las expresiones:

```
emp.aumentarSueldo(3);  // 1. ok. aumentarSueldo
```

```
es de Empleado
emp.asignarPresupuesto(1500);  // 2. error de
compilación
```

En la primera expresión no hay error porque el método aumentarSueldo está definido en la clase Empleado. En la segunda expresión se produce un error de compilación porque el método asignarPresupuesto no existe para la clase Empleado.

Por último, la situación para el método toString es algo más compleja. Si se invoca el método:

```
emp.toString();  // se invoca el metodo toString
de Ejecutivo
```

el método que resultará llamado es el de la clase Ejecutivo. `toString` existe tanto para Empleado como para Ejecutivo, por lo que el compilador Java no determina en el momento de la compilación que método va a usarse. Sintácticamente la expresión es correcta. El compilador retrasa la decisión de invocar a un método o a otro al momento de la ejecución. Esta técnica se conoce con el nombre de dinamic binding o late binding. En el momento de la ejecución la JVM comprueba el contenido de la referencia `emp`. Si apunta a un objeto de la clase Empleado invocará al método `toString` de esta clase. Si apunta a un objeto Ejecutivo invocará por el contrario al método toString de Ejecutivo.

Operador cast

Si se desea acceder a los métodos de la clase derivada teniendo una referencia de una clase base, como en el ejemplo del apartado anterior hay que convertir explicitamente la referencia de un tipo a otro. Esto se hace con el operador de cast de la siguiente forma:

```
Empleado emp = new Ejecutivo("Máximo Dueño" ,
2000);
Ejecutivo ej = (Ejecutivo)emp; // se convierte la
referencia de tipo
ej.asignarPresupuesto(1500);
```

La expresión de la segunda línea convierte la referencia de tipo Empleado asignándola a una referencia de tipo Ejecutivo. Para el compilador es correcto porque Ejecutivo es una clase derivada de Empleado. En tiempo de ejecución la JVM convertirá la referencia si efectivamente `emp` apunta a un objeto de la clase Ejecutivo. Si se intenta:

```
Empleado emp = new Empleado("Javier Todudas" ,
2000);
Ejecutivo ej = (Ejecutivo)emp;
```

no dará problemas al compilar, pero al ejecutar se producirá un error porque la referencia `emp` apunta a un objeto de clase Empleado y no a uno de clas Ejecutivo.

La clase `Object`

En Java existe una clase base que es la raíz de la jerarquía y de la cual heredan todas aunque no se diga explicitamente mediante la clausula `extends`. Esta clase base se llama `Object` y contiene algunos métodos básicos. La mayor parte de ellos no hacen nada pero pueden ser redefinidos por las clases derivadas para implementar comportamientos específicos. Los métodos declarados por la clase `Object` son los siguientes:

```
public class Object {
    public final Class getClass() { . . . }
    public String toString() { . . . }
    public boolean equals(Object obj) { . . . }
    public int hashCode() { . . . }
    protected Object clone() throws
CloneNotSupportedException { . . . }
    public final void wait() throws
IllegalMonitorStateException,

InterruptedException { . . . }
    public final void wait(long millis) throws
IllegalMonitorStateException,

InterruptedException {. . .}
    public final void wait(long millis, int
nanos) throws

IllegalMonitorStateException,

InterruptedException { . . . }
    public final void notify() throws
IllegalMonitorStateException { . . . }
    public final void notifyAll() throws

IllegalMonitorStateException { . . . }
    protected void finalize() throws Throwable {
. . . }
}
```

Las cláusulas `final` y `throws` se verán más adelante. Como puede verse `toString` es un método de Object, que puede ser redefinido en las clases derivadas. Los métodos `wait`, `notify` y `notifyAll` tienen que ver con la gestión de threads de la JVM. El método finalize ya se ha comentado al hablar del recolector de basura.

Para una descripción exahustiva de los métodos de Object se puede consultar la documentación de la

API del JDK.

La cláusula `final`

En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa no pueda ser extendida. Para esto está la cláusula `final`, que tiene significados levemente distintos según se aplique a un dato miembro, a un método o a una clase.

Para una clase, `final` significa que la clase no puede extenderse. Es, por tanto el punto final de la cadena de clases derivadas. Por ejemplo si se quisiera impedir la extensión de la clase Ejecutivo, se pondría:

```
final class Ejecutivo {
. . .
}
```

Para un método, `final` significa que no puede redefinirse en una clase derivada. Por ejemplo si declaramos:

```
class Empleado {
    . . .
    public final void aumentarSueldo(int
porcentaje) {
        . . .
    }
    . . .
}
```

entonces la clase Ejecutivo, clase derivada de Empleado no podría reescribir el método aumentarSueldo, y por tanto cambiar su comportamiento.

Para un dato miembro, `final` significa también que no puede ser redefinido en una clase derivada, como para los métodos, pero además significa que su valor no puede ser cambiado en ningún sitio; es decir el modificador `final` sirve también para definir valores constantes. Por ejemplo:

```
class Circulo {
    . . .
    public final static float PI = 3.141592;
    . . .
}
```

En el ejemplo se define el valor de PI como de tipo `float`, estático (es igual para todas las instancias), constante (modificador `final`) y de acceso público.

Herencia simple

Java incorpora un mecanismo de herencia simple. Es decir, una clase sólo puede tener una superclase directa de la cual hereda todos los datos y métodos. Puede existir una cadena de clases derivadas en que la clase A herede de B y B herede de C, pero no es posible escribir algo como:

```
class A extends B , C ....  // error
```

Este mecanismo de herencia múltiple no existe en Java.

Java implanta otro mecanismo que resulta parecido al de herencia múltiple que es el de las interfaces que se verá más adelante.

Indice Siguiente Anterior

Indice Siguiente Anterior

Gestión de Excepciones

Excepciones. Categorías.

Las excepciones son el mecanismo por el cual pueden controlarse en un programa Java las condiciones de error que se producen. Estas condiciones de error pueden ser errores en la lógica del programa como un índice de un array fuera de su rango, una división por cero o errores disparados por los propios objetos que denuncian algún tipo de estado no previsto, o condición que no pueden manejar.

La idea general es que cuando un objeto encuentra una condición que no sabe manejar crea y dispara una excepción que deberá ser capturada por el que le llamó o por alguien más arriba en la pila de llamadas. Las excepciones son objetos que contienen información del error que se ha producido y que heredan de la clase Throwable o de la clase Exception. Si nadie captura la excepción interviene un manejador por defecto que normalmente imprime información que ayuda a encontrar quién produjo la excepción.

Existen dos categorías de excepciones:

- Excepciones verificadas: El compilador obliga a verificarlas. Son todas las que son lanzadas explicitamente por objetos de usuario.
- Excepciones no verificadas: El compilador no obliga a su verificación. Son excepciones como divisiones por cero, excepciones de puntero nulo, o índices fuera de rango.

Generación de excepciones

Supongamos que tenemos una clase Empresa que tiene un array de objetos Empleado (clase vista en capítulos anteriores). En esta clase podríamos tener métodos para contratar un Empleado (añadir un nuevo objeto al array), despedirlo (quilarlo del array) u obtener el nombre a partir del número de empleado. La clase podría ser algo así como lo siguiente:

```
public class Empresa {
    String nombre;
    Empleado [] listaEmpleados;
    int totalEmpleados = 0;
    . . .
```

```
        Empresa(String n, int maxEmp) {
            nombre = n;
            listaEmpleados = new Empleado [maxEmp];
        }
        . . .
        void nuevoEmpleado(String nombre, int sueldo)
    {
           if (totalEmpleados < listaEmpleados.length
    ) {
                listaEmpleados[totalEmpleados++] =
    new Empleado(nombre,sueldo);
           }
        }
    }
```

Observese en el método nuevoEmpleado que se comprueba que hay sitio en el array para almacenar la referencia al nuevo empleado. Si lo hay se crea el objeto. Pero si no lo hay el método no hace nada más. No da ninguna indicación de si la operación ha tenido éxito o no. Se podría hacer una modificación para que, por ejemplo el método devolviera un valor booleano true si la operación se ha completado con éxito y false si ha habido algún problema.

Otra posibilidad es generar una excepción verificada (Una excepción no verificada se produciría si no se comprobara si el nuevo empleado va a caber o no en el array). Vamos a ver como se haría esto.

Las excepciones son clases, que heredan de la clase genérica Exception. Es necesario por tanto asignar un nombre a nuestra excepción. Se suelen asignar nombres que den alguna idea del tipo de error que controlan. En nuestro ejemplo le vamos a llamar CapacidadEmpresaExcedida.

Para que un método lance una excepción:

- Debe declarar el tipo de excepción que lanza con la cláusula throws, en su declaración.
- Debe lanzar la excepción, en el punto del código adecuado con la sentencia throw.

En nuestro ejemplo:

```
    void nuevoEmpleado(String nombre, int sueldo) throws
CapacidadEmpresaExcedida {
       if (totalEmpleados < listaEmpleados.length) {
            listaEmpleados[totalEmpleados++] = new
Empleado(nombre,sueldo);
       }
       else throw new CapacidadEmpresaExcedida(nombre);
    }
```

Además, necesitamos escribir la clase `CapacidadEmpresaExcedida.` Sería algo así:

```
public class CapacidadEmpresaExcedida extends Exception {
    CapacidadEmpresaExcedida(String nombre) {
        super("No es posible añadir el empleado " +
nombre);
    }
    . . .
}
```

La sentencia throw crea un objeto de la clase `CapacidadEmpresaExcedida` . El constructor tiene un argumento (el nombre del empleado). El constructor simplemente llama al constructor de la superclase pasándole como argumento un texto explicativo del error (y el nombre del empleado que no se ha podido añadir).

La clase de la excepción puede declarar otros métodos o guardar datos de depuración que se consideren oportunos. El único requisito es que extienda la clase Exception. Consultar la documentación del API para ver una descripción completa de la clase Exception.

De esta forma se pueden construir métodos que generen excepciones.

Captura de excepciones

Con la primera versión del método nuevoEmpleado (sin excepción) se invocaría este método de la siguiente forma:

```
Empresa em = new Empresa("La Mundial");
em.nuevoEmpleado("Adán Primero",500);
```

Si se utilizara este formato en el segundo caso (con excepción) el compilador produciría un error indicando que no se ha capturado la excepción verificada lanzada por el método nuevoEmpleado. Para capturar la excepción es utiliza la construcción try / catch, de la siguiente forma:

```
Empresa em = new Empresa("La Mundial");
try {
    em.nuevoEmpleado("Adán Primero",500);
} catch (CapacidadEmpresaExcedida exc) {
    System.out.println(exc.toString());
    System.exit(1);
}
```

- Se encierra el código que puede lanzar la excepción en un bloque try / catch.
- A continuación del catch se indica que tipo de excepción se va a capturar.
- Después del catch se escribe el código que se ejecutará si se lanza la excepción.
- Si no se lanza la excepción el bloque catch no se ejecuta.

El formato general del bloque try / catch es:

```
try {
    . . .
} catch (Clase_Excepcion nombre) { . . .}
  catch (Clase_Excepcion nombre) { . . .}
   . . .
```

Observese que se puede capturar más de un tipo de excepción declarando más de una sentencia catch. También se puede capturar una excepción genérica (clase Exception) que engloba a todas las demás.

En ocasiones el código que llama a un método que dispara una excepción tampoco puede (o sabe) manejar esa excepción. Si no sabe que hacer con ella puede de nuevo lanzarla hacia arriba en la pila de llamada para que la gestione quien le llamo (que a su vez puede capturarla o reenviarla). Cuando un método no tiene intención de capturar la excepción debe declararla metdiante la cláusula throws, tal como hemos visto en el método que genera la excepción.

Supongamos que, en nuestro ejemplo es el método main de una clase el que invoca el método nuevoEmpleado. Si no quiere capturar la excepción debe hacer lo siguiente:

```
public static void main(String [] args) throws
CapacidadEmpresaExcedida {
    Empresa em = new Empresa("La Mundial");
    em.nuevoEmpleado("Adán Primero",500);
}
```

Cláusula `finally`

La cláusula `finally` forma parte del bloque **`try`** / **`catch`** y sirve para especificar un bloque de código que se ejecutará tanto si se lanza la excepción como si no. Puede servir para limpieza del estado interno de los objetos afectados o para liberar recursos externos (descriptores de fichero, por ejemplo). La sintaxis global del bloque **`try`** / **`catch`** / **`finally`** es:

```
try {
    . . .
} catch (Clase_Excepcion nombre) { . . .}
  catch (Clase_Excepcion nombre) { . . .}
   . . .
  finally { . . .}
```

Indice Siguiente Anterior

Indice Siguiente Anterior

Clases envoltorio (Wrapper)

Definición y uso de clases envoltorio

En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, etc.) como objetos. Por ejemplo, los contenedores definidos por el API en el package java.util (Arrays dinámicos, listas enlazadas, colecciones, conjuntos, etc.) utilizan como unidad de almacenamiento la clase Object. Dado que Object es la raíz de toda la jerarquía de objetos en Java, estos contenedores pueden almacenar cualquier tipo de objetos. Pero los datos primitivos no son objetos, con lo que quedan en principio excluidos de estas posibilidades.

Para resolver esta situación el API de Java incorpora las clases envoltorio (wrapper class), que no son más que dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos. Por ejemplo podríamos definir una clase envoltorio para los enteros, de forma bastante sencilla, con:

```
public class Entero {
    private int valor;

    Entero(int valor) {
        this.valor = valor;
    }

    int intValue() {
        return valor;
    }
}
```

La API de Java hace innecesario esta tarea al proporcionar un conjunto completo de clases envoltorio para todos los tipos primitivos. Adicionalmente a la funcionalidad básica que se muestra en el ejemplo las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de / hacia datos primitivos, conversiones a String, etc.)

Las clases envoltorio existentes son:

- `Byte` para `byte`.
- `Short` para `short`.
- `Integer` para `int`.

- `Long` para `long`.
- `Boolean` para `boolean`
- `Float` para `float`.
- `Double` para `double` y
- `Character` para `char`.

Observese que las clases envoltorio tienen siempre la primera letra en mayúsculas.

Las clases envoltura se usan como cualquier otra:

```
Integer i = new Integer(5);
int x = i.intValue();
```

Hay que tener en cuenta que las operaciones aritméticas habituales (suma, resta, multiplicación ...) están definidas solo para los datos primitivos por lo que las clases envoltura no sirven para este fin.

Las variables primitivas tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes envolturas siempre que se pueda.

Resumen de métodos de Integer

Las clases envoltorio proporcionan también métodos de utilidad para la manipulación de datos primitivos. La siguiente tabla muestra un resumen de los métodos disponibles para la clase Integer

Método	**Descripción**
`Integer(int valor)` `Integer(String valor)`	Constructores a partir de int y String
`int intValue() / byte byteValue() / float floatValue() . . .`	Devuelve el valor en distintos formatos, int, long, float, etc.
`boolean equals(Object obj)`	Devuelve true si el objeto con el que se compara es un Integer y su valor es el mismo.
`static Integer getInteger(String s)`	Devuelve un Integer a partir de una cadena de caracteres. Estático
`static int parseInt(String s)`	Devuelve un int a partir de un String. Estático.

`static String toBinaryString(int i)`
`static String toOctalString(int i)`
`static String toHexString(int i)`
`static String toString(int i)`
	Convierte un entero a su representación en String en binario, octal, hexadecimal, etc. Estáticos

`String toString()`	
`static` `Integer valueOf(String s)`	Devuelve un Integer a partir de un String. Estático.

El API de Java contiene una descripción completa de todas las clases envoltorio en el package java.lang.

Indice Siguiente Anterior

Indice Siguiente Anterior

Clases abstractas

Concepto

Hay ocasiones, cuando se desarrolla una jerarquía de clases en que algún comportamiento está presente en todas ellas pero se materializa de forma distinta para cada una. Por ejemplo, pensemos en una estructura de clases para manipular figuras geométricas. Podríamos pensar en tener una clase genérica, que podría llamarse FiguraGeometrica y una serie de clases que extienden a la anterior que podrían ser Circulo, Poligono, etc. Podría haber un método dibujar dado que sobre todas las figuras puede llevarse a cabo esta acción, pero las operaciones concretas para llevarla a cabo dependen del tipo de figura en concreto (de su clase). Por otra parte la acción dibujar no tiene sentido para la clase genérica FiguraGeometrica, porque esta clase representa una abstracción del conjunto de figuras posibles.

Para resolver esta problemática Java proporciona las clases y métodos abstractos. Un método abstracto es un método declarado en una clase para el cual esa clase no proporciona la implementación (el código). Una clase abstracta es una clase que tiene al menos un método abstracto. Una clase que extiende a una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

Declaración e implementación de métodos abstractos

Siguiendo con el ejemplo del apartado anterior, se puede escribir:

```
abstract class FiguraGeometrica {
    . . .
    abstract void dibujar();
    . . .
}

class Circulo extends FiguraGeometrica {
    . . .
    void dibujar() {
        // codigo para dibujar Circulo
```

```
            . . .
        }
    }
```

La clase abstracta se declara simplemente con el modificador **`abstract`** en su declaración. Los métodos abstractos se declaran también con el mismo modificador, declarando el método pero sin implementarlo (sin el bloque de código encerrado entre {}). La clase derivada se declara e implementa de forma normal, como cualquier otra. Sin embargo si no declara e implementa los métodos abstractos de la clase base (en el ejemplo el método `dibujar`) el compilador genera un error indicando que no se han implementado todos los métodos abstractos y que, o bien, se implementan, o bien se declara la clase abstracta.

Referencias y objetos abstractos

Se pueden crear referencias a clases abstractas como cualquier otra. No hay ningún problema en poner:

```
FiguraGeometrica figura;
```

Sin embargo una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de una clase abstracta. El compilador producirá un error si se intenta:

```
FiguraGeometrica figura = new FiguraGeometrica();
```

Esto es coherente dado que una clase abstracta no tiene completa su implementación y encaja bien con la idea de que algo abstracto no puede materializarse.

Sin embargo utilizando el up-casting visto en el capítulo dedicado a la Herencia si se puede escribir:

```
FiguraGeometrica figura = new Circulo(. . .);
figura.dibujar();
```

La invocación al método `dibujar`se resolverá en tiempo de ejecución y la JVM llamará al método de la clase adecuada. En nuestro ejemplo se llamará al método `dibujar`de la clase `Circulo`.

Indice Siguiente Anterior

Indice Siguiente Anterior

Interfaces

Concepto de Interface

El concepto de Interface lleva un paso más adelante la idea de las clases abstractas. En Java una interface es una clase abstracta pura, es dcir una clase donde todos los métodos son abstractos (no se implementa ninguno). Permite al diseñador de clases establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero no bloques de código). Una interface puede también contener datos miembro, pero estos son siempre static y final. Una interface sirve para establecer un 'protocolo' entre clases.

Para crear una interface, se utiliza la palabra clave interface en lugar de class. La interface puede definirse public o sin modificador de acceso, y tiene el mismo significado que para las clases. Todos los métodos que declara una interface son siempre public.

Para indicar que una clase implementa los métodos de una interface se utiliza la palabra clave implements. El compilador se encargará de verificar que la clase efectivamente declare e implemente todos los métodos de la interface. Una clase puede implementar más de una interface.

Declaración y uso

Una interface se declara:

```
interface nombre_interface {
    tipo_retorno nombre_metodo ( lista_argumentos
) ;
    . . .
}
```

Por ejemplo:

```
interface InstrumentoMusical {
    void tocar();
    void afinar();
    String tipoInstrumento();
}
```

Y una clase que implementa la interface:

```
class InstrumentoViento extends Object implements
InstrumentoMusical {
    void tocar() { . . . };
    void afinar() { . . .};
    String tipoInstrumento() {}
}

class Guitarra extends InstrumentoViento {
    String tipoInstrumento() {
        return "Guitarra";
    }
}
```

La clase `InstrumentoViento` implementa la interface, declarando los métodos y escribiendo el código correspondiente. Una clase derivada puede también redefinir si es necesario alguno de los métodos de la interface.

Referencias a Interfaces

Es posible crear referencias a interfaces, pero las interfaces no pueden ser instanciadas. Una referencia a una interface puede ser asignada a cualquier objeto que implemente la interface. Por ejemplo:

```
InstrumentoMusical instrumento = new Guitarra();
instrumento.play();
System.out.prinln(instrumento.tipoInstrumento());

InstrumentoMusical i2 = new InstrumentoMusical();
//error.No se puede instanciar
```

Extensión de interfaces

Las interfaces pueden extender otras interfaces y, a diferencia de las clases, una interface puede extender más de una interface. La sintaxis es:

```
interface nombre_interface  extends
nombre_interface  , . . . {
    tipo_retorno nombre_metodo ( lista_argumentos
) ;
    . . .
}
```

Agrupaciones de constantes

Dado que, por definición, todos los datos miembros que se definen en una interface son static y final, y dado que las interfaces no pueden instanciarse resultan una buena herramienta para implantar grupos de constantes. Por ejemplo:

```
public interface Meses {
    int ENERO = 1 , FEBRERO = 2 . . . ;
    String [] NOMBRES_MESES = { " " , "Enero" ,
"Febrero" , . . . };
}
```

Esto puede usarse simplemente:

```
System.out.println(Meses.NOMBRES_MESES[ENERO]);
```

Un ejemplo casi real

El ejemplo mostrado a continuación es una simplificación de como funciona realmente la gestión de eventos en el sistema gráfico de usuario soportado por el API de Java (AWT o swing). Se han cambiado los nombres y se ha simplificado para mostrar un caso real en que el uso de interfaces resuelve un problema concreto.

Supongamos que tenemos una clase que representa un botón de acción en un entorno gráfico de usuario (el típico botón de confirmación de una acción o de cancelación). Esta clase pertenecerá a una amplia jerarquía de clases y tendrá mecanismos complejos de definición y uso que no son objeto del ejemplo. Sin embargo podríamos pensar que la clase Boton tiene miembros como los siguientes.

```
class Boton extends . . . {
    protected int x , y, ancho, alto; // posicion
del boton
    protected String texto;  // texto del boton
    Boton(. . .) {
        . . .
    }
    void dibujar() { . . .}
    public void asignarTexto(String t) { . . .}
    public String obtenerTexto() { . . .)
    . . .
}
```

Lo que aquí nos interesa es ver lo que sucede cuando el usuario, utilizando el ratón pulsa sobre el botón. Supongamos que la clase Boton tiene un método, de nombre por ejemplo click(), que es invocado por el gestor de ventanas cuando ha detectado que el usuario ha pulsado el botón del ratón sobre él. El botón deberá realizar alguna acción como dibujarse en posición 'pulsado' (si tiene efectos de tres dimensiones) y además, probablemente, querrá informar a alguien de que se ha producido la acción del usuario. Es en este mecanismo de 'notificación' donde entra el concepto de interface. Para ello definimos una interface Oyente de la siguiente forma:

```
interface Oyente {
    void botonPulsado(Boton b);
}
```

La interface define un único método `botonPulsado`. La idea es que este método sea invocado por la clase Boton cuando el usuario pulse el botón. Para que esto sea posible en algún momento hay que notificar al Boton quien es el Oyente que debe ser notificado. La clase Boton quedaría:

```
class Boton extends . . . {
    . . .
    private Oyente oyente;
    void registrarOyente(Oyente o) {
        oyente = o;
    }
    void click() {
        . . .
        oyente.botonPulsado(this);
    }
}
```

El método `registrarOyente` sirve para que alguien pueda 'apuntarse' como receptor de las acciones del usuario. Obsérvese que existe una referencia de tipo Oyente. A Boton no le importa que clase va a recibir su notificación. Simplemente le importa que implante la interface Oyente para poder invocar el método `botonPulsado`. En el método click se invoca este método. En el ejemplo se le pasa como parámetro una referencia al propio objeto Boton. En la realidad lo que se pasa es un objeto 'Evento' con información detallada de lo que ha ocurrido.

Con todo esto la clase que utiliza este mecanismo podría tener el siguiente aspecto:

```
class miAplicacion extends . . . implements
Oyente {
    public static main(String [] args) {
        new miAplicacion(. . .);
        . . .
    }
     . . .
    miAplicacion(. . .) {
```

```
            . . .
            Boton b = new Boton(. . .);
            b.registrarOyente(this);
        }

        . . .
        void botonPulsado(Boton x) {
            // procesar click
            . . .
        }
    }
```

Obsérvese en el método `registrarOyente` que se pasa la referencia **`this`**que en el lado de la clase Boton es recogido como una referencia a la interface Oyente. Esto es posible porque la clase `miAplicacion` implementa la interface `Oyente` . En términos clásicos de herencia `miAplicacion` ES un `Oyente` .

Indice Siguiente Anterior

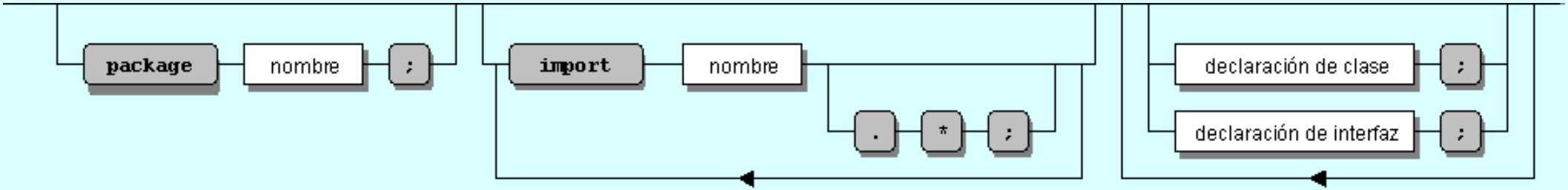
Apéndice I. Sintaxis del lenguaje Java

Presentamos aquí una sintaxis del lenguaje Java utilizando grafos sintácticos en lugar de las notaciones habituales. Esta forma de representación es mucho más visual y ayuda mejor a la comprensión del lenguaje.

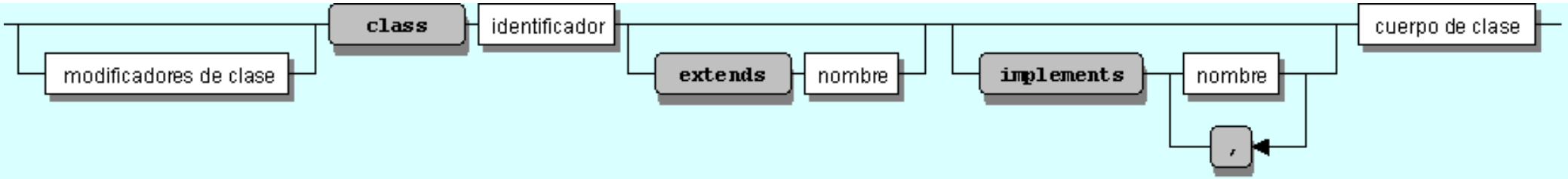
Los grafos sintácticos se leen siempre de izquierda a derecha. Solamente es posible retroceder de derecha a izquierda cuando se indica explicitamente con una flecha (normalmente para expresar una condición iterativa). Un rótulo con el fondo gris indica un símbolo o una palabra reservada del lenguaje (un símbolo terminal), que debe aparecer exactamente así. Un rótulo con el fondo blanco indica un símbolo no terminal que se desarrolla en otro grafo sintáctico, accesible pulsando sobre él.

Aviso: Esta página está en elaboración.
La sintaxis todavía no está completa.

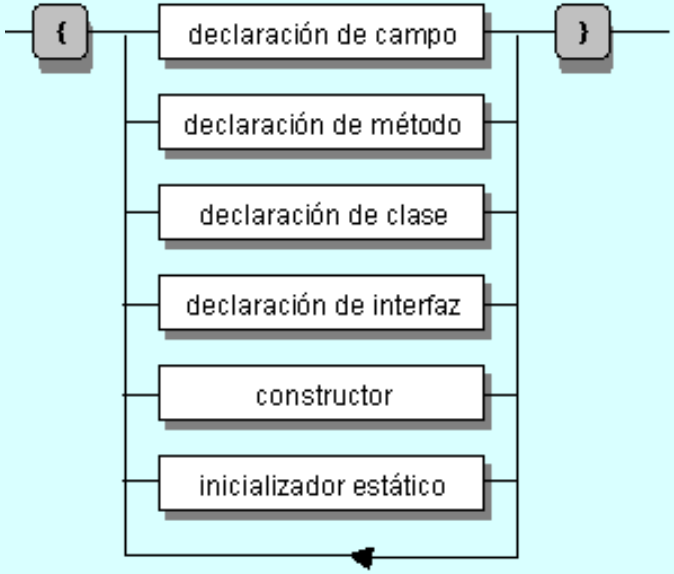
Unidad de compilación



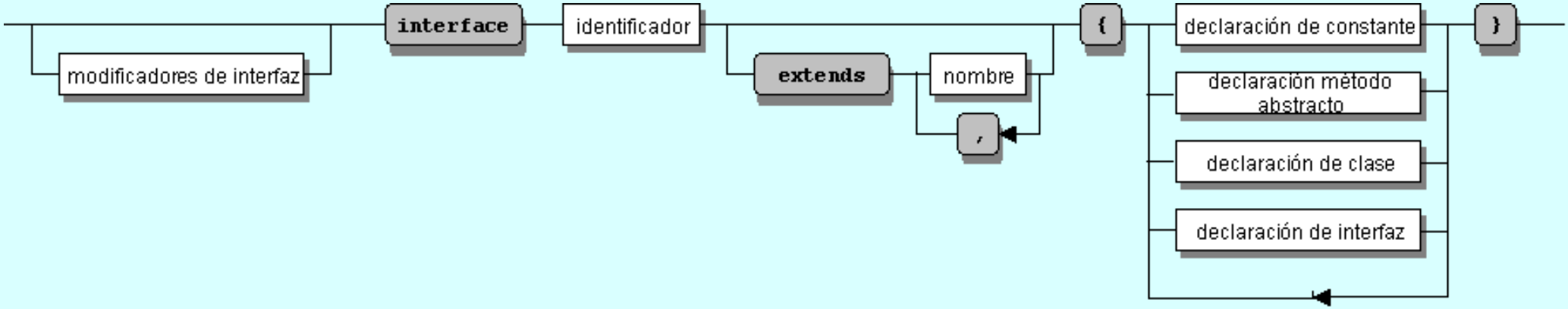
Declaración de clase



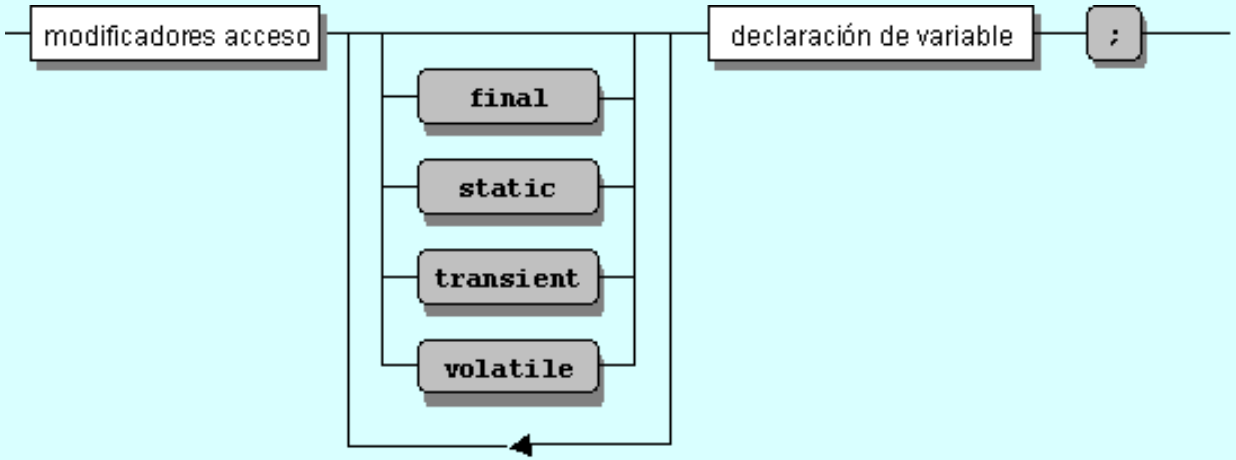
Cuerpo de clase



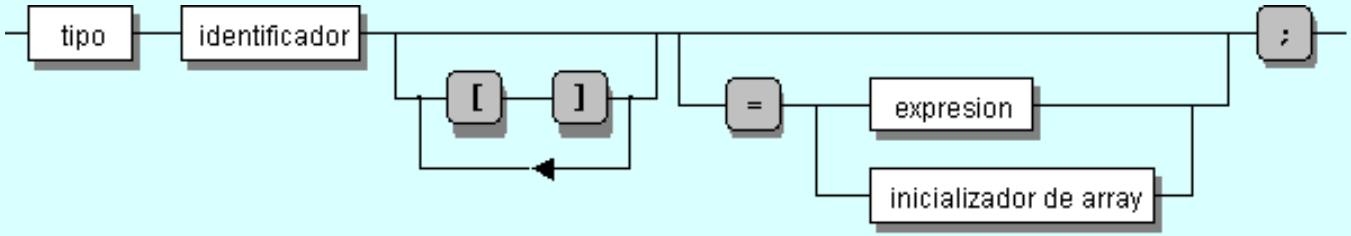
Declaración de interfaz



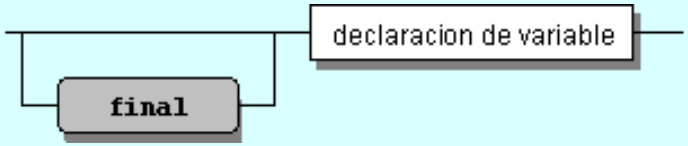
Declaración de campo



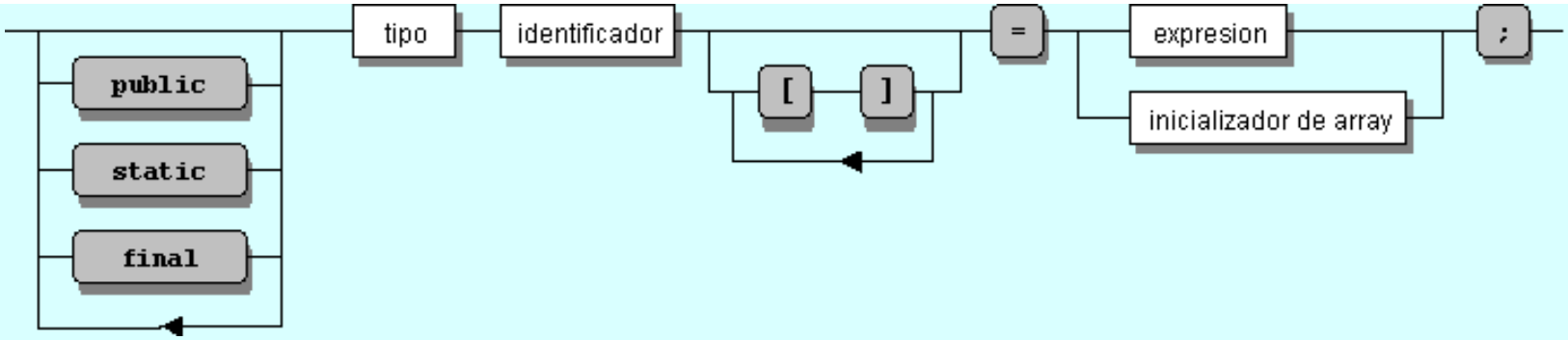
Declaración de variable



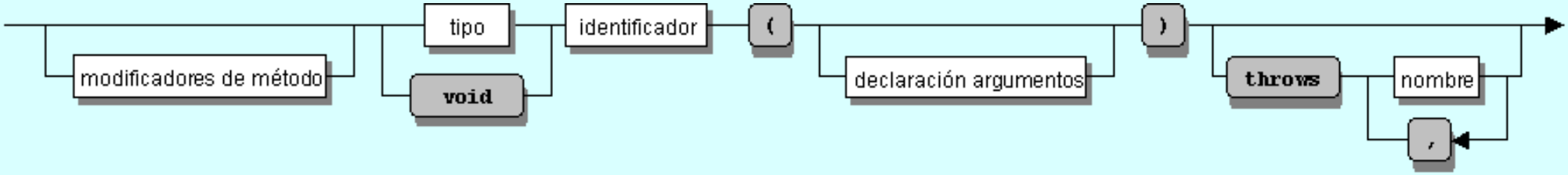
Declaración de variable local

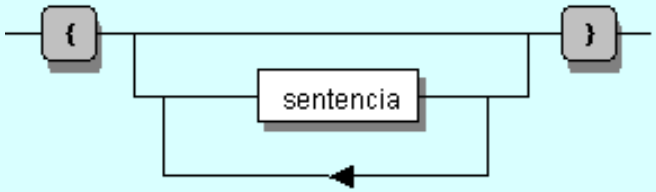


Declaración de constante

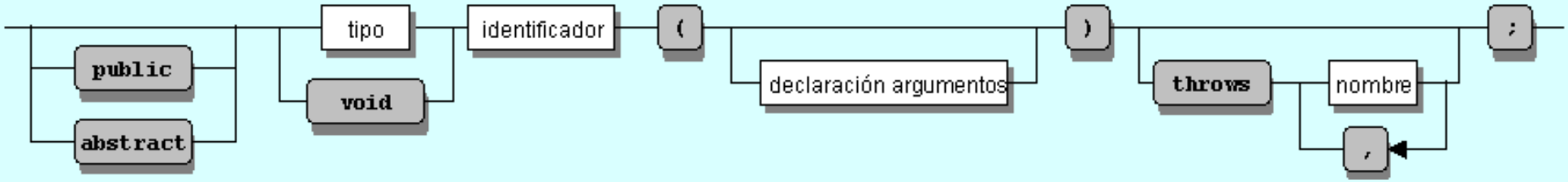


Declaración de método

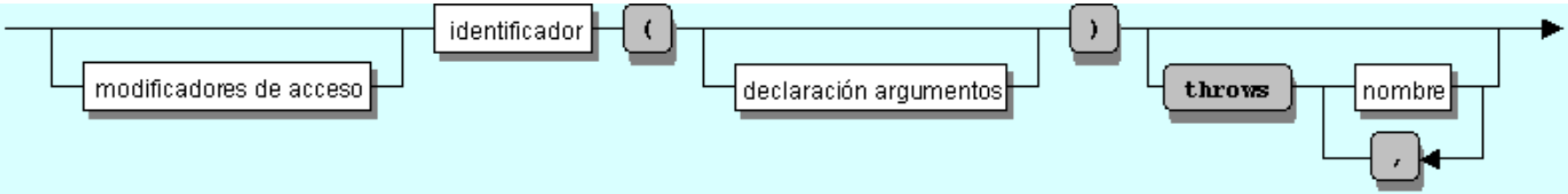


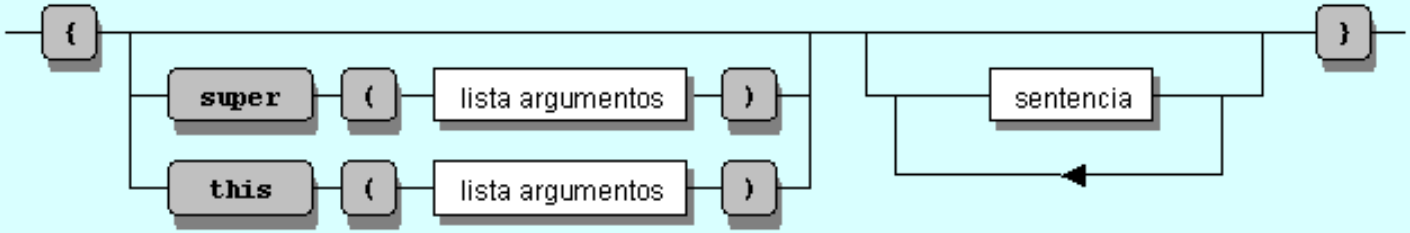


Declaración de método abstracto

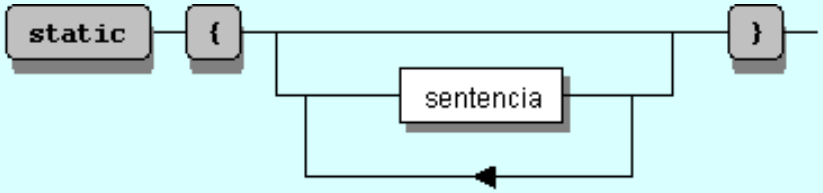


Constructor

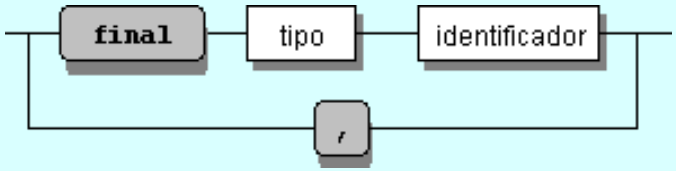




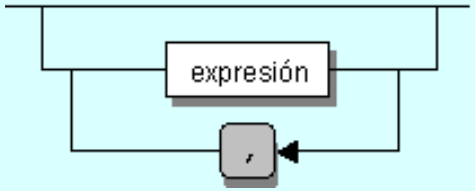
Inicializador estático



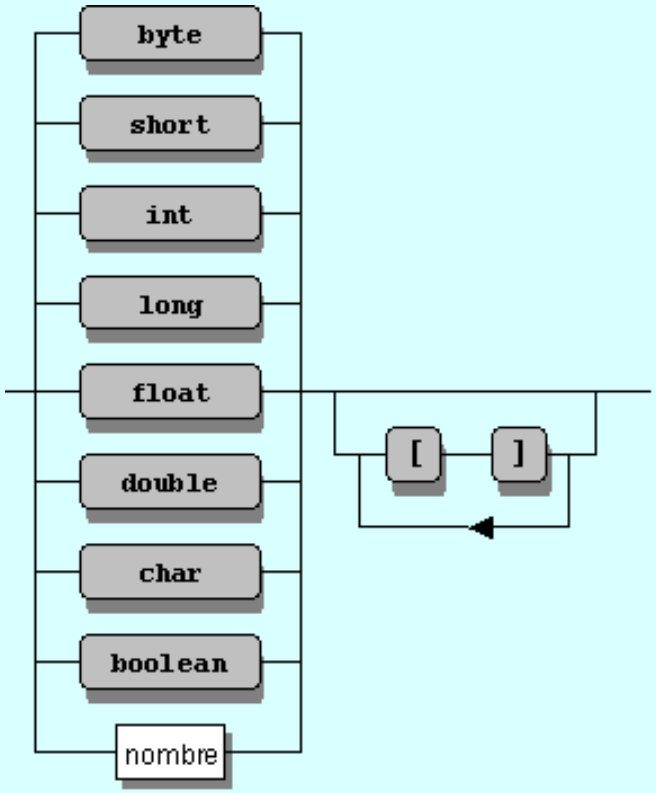
Declaración de argumentos



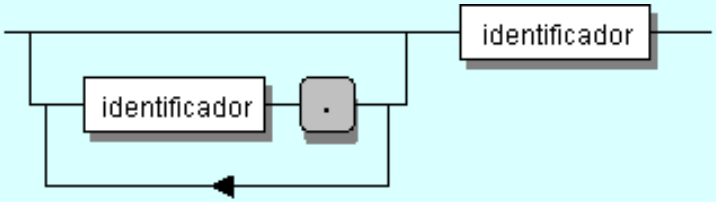
Lista de argumentos



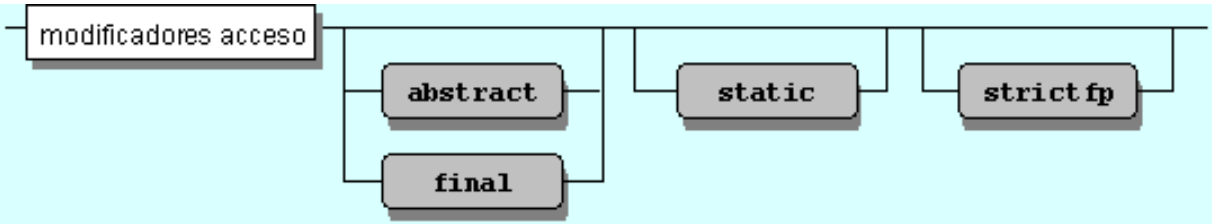
Tipo



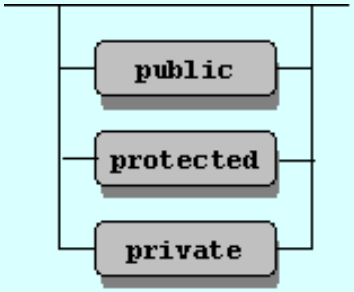
Nombre



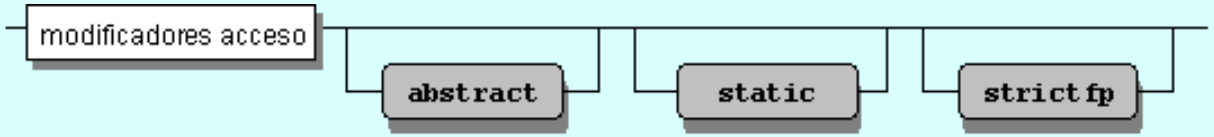
Modificadores de clase



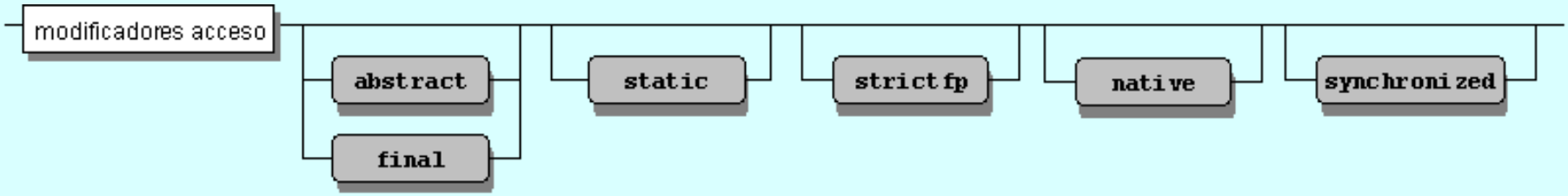
Modificadores de acceso



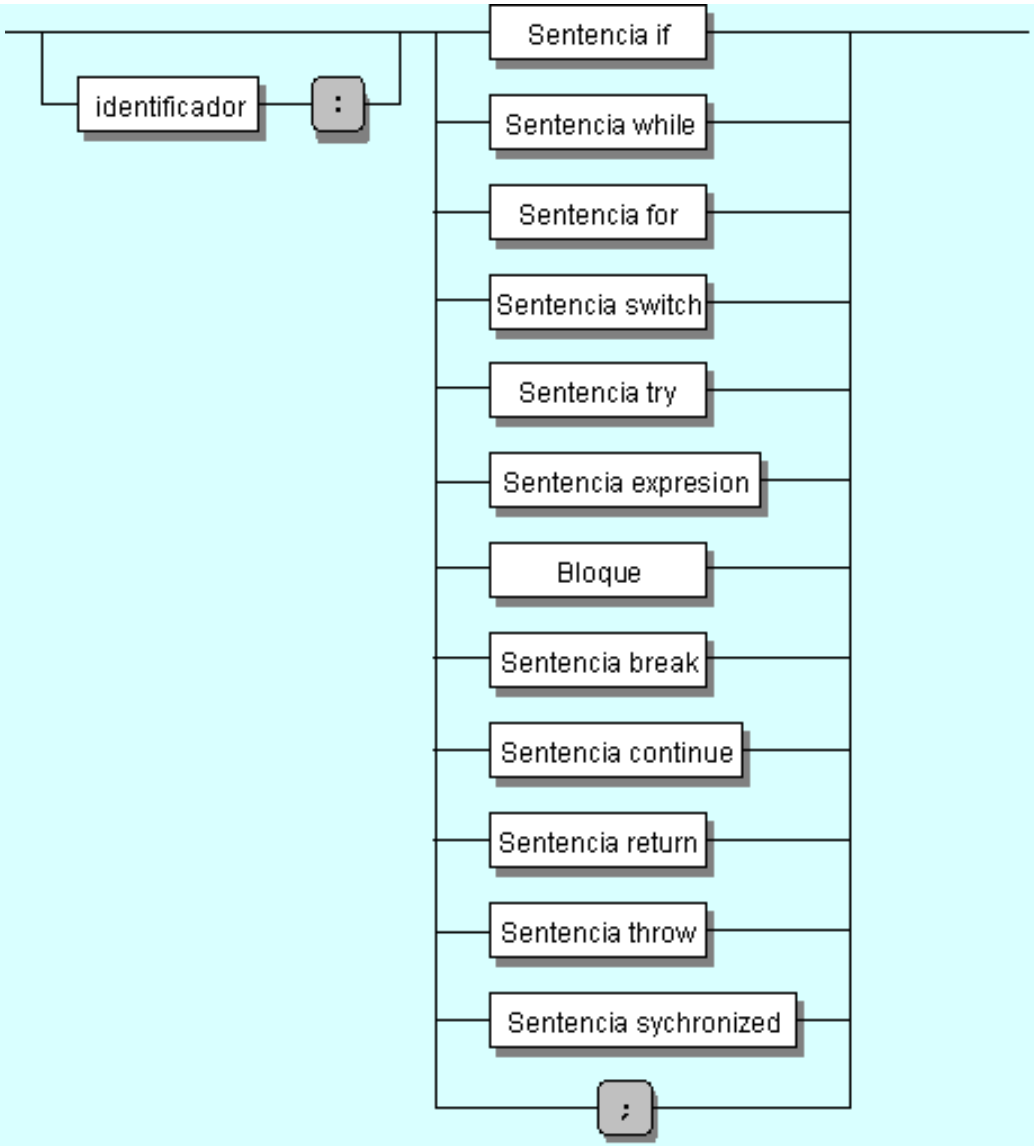
Modificadores de interfaz



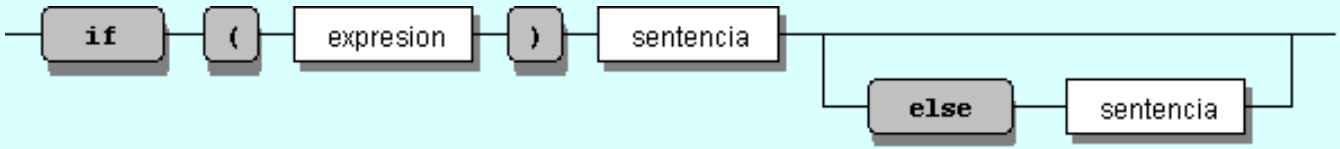
Modificadores de método



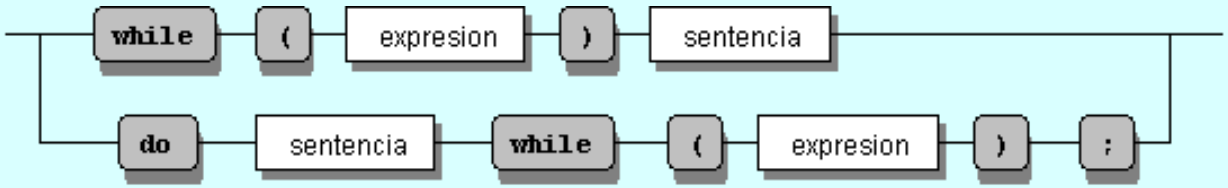
Sentencia



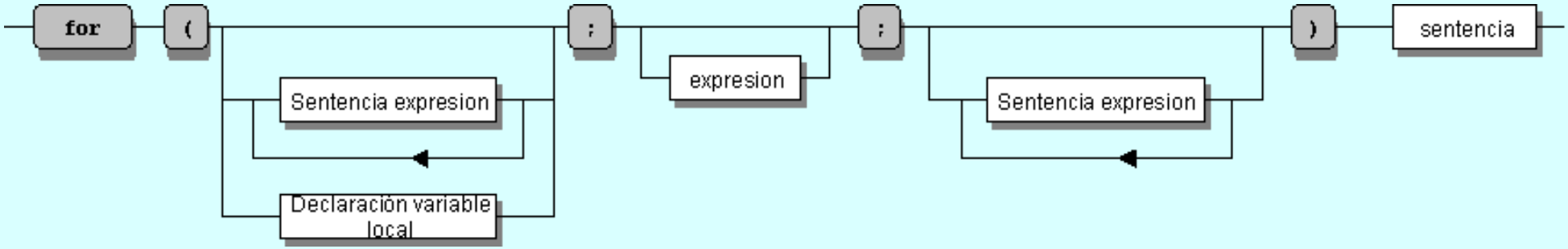
Sentencia if



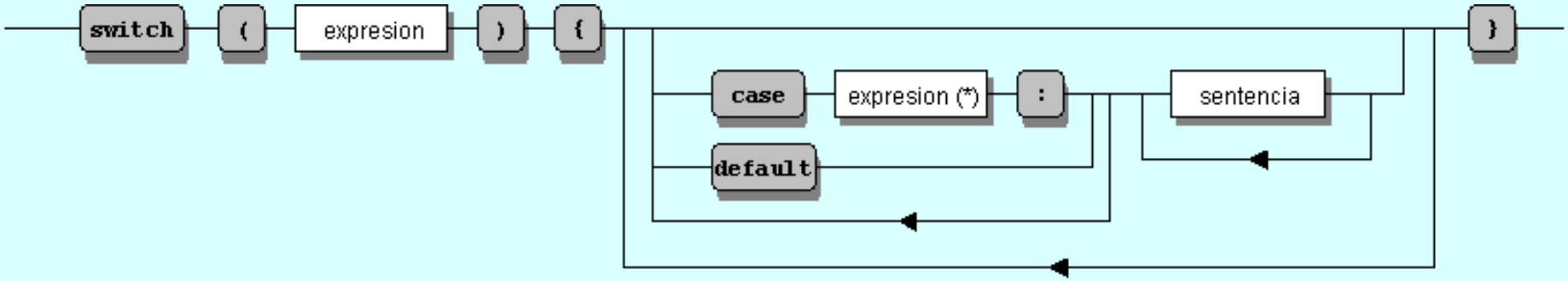
Sentencia while



Sentencia for

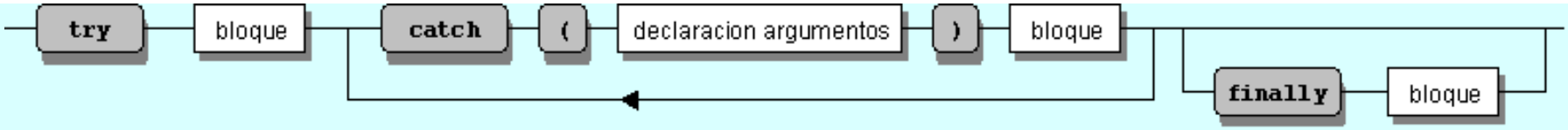


Sentencia switch

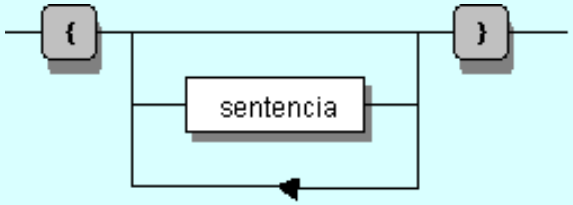


(*) expresion debe ser de tipo **`char, byte, short o int`**

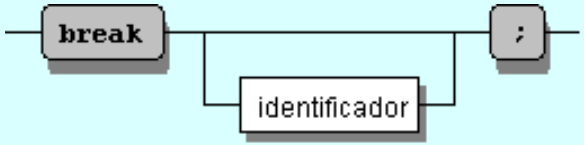
Sentencia try



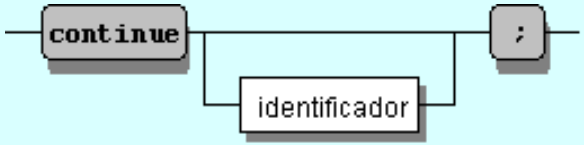
Bloque



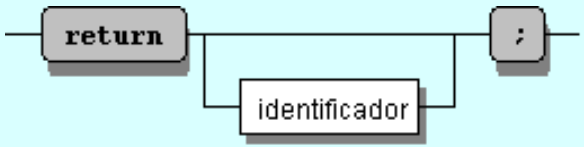
Sentencia break



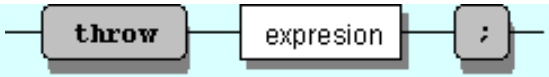
Sentencia continue



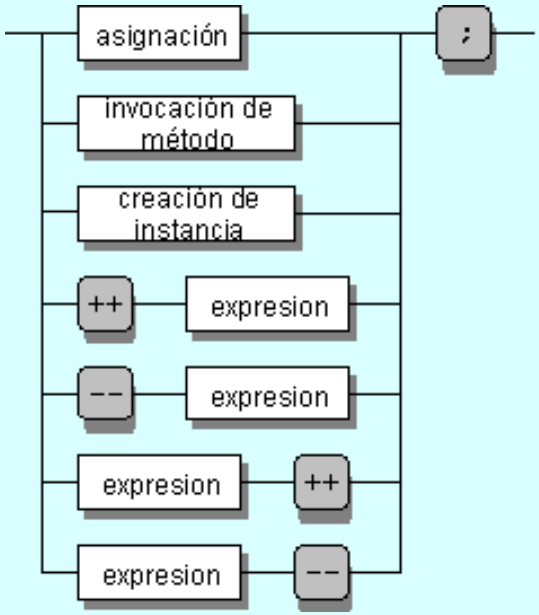
Sentencia return



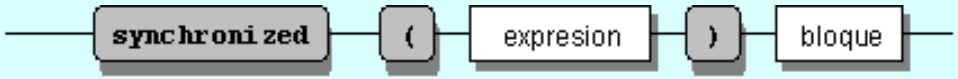
Sentencia throw



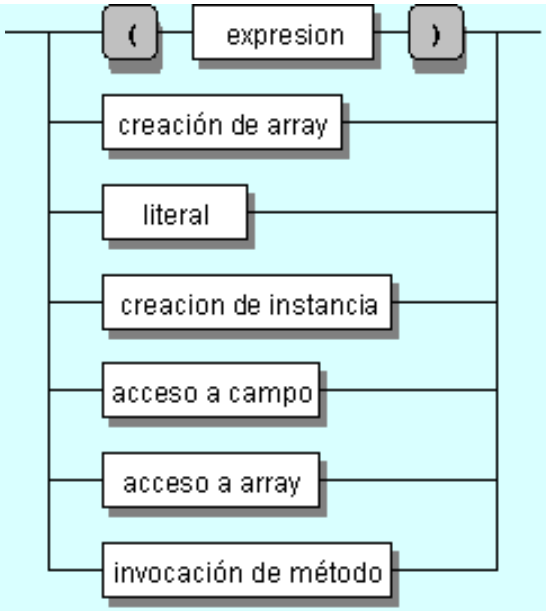
Sentencia expresión



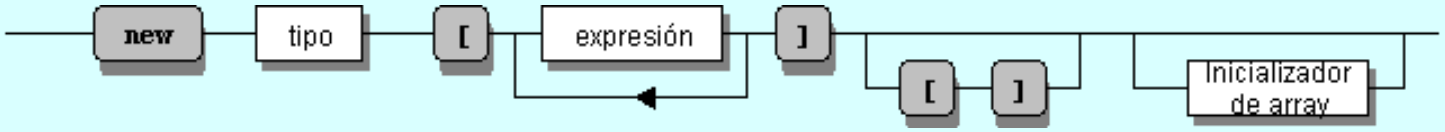
Sentencia synchronized



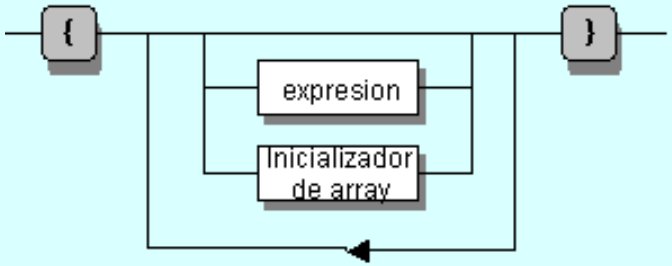
Expresión primaria



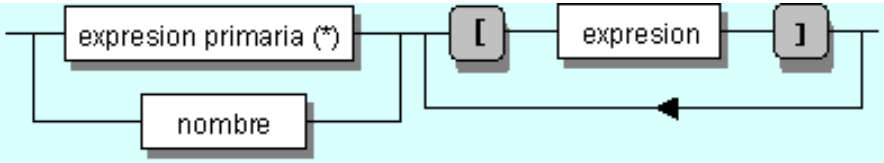
Creación de array



Inicializador de array

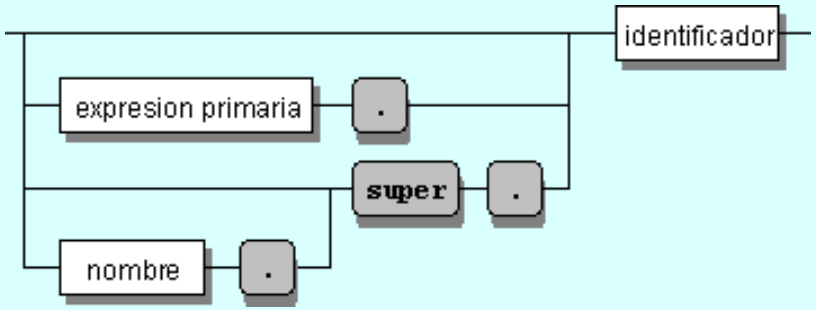


Acceso a array

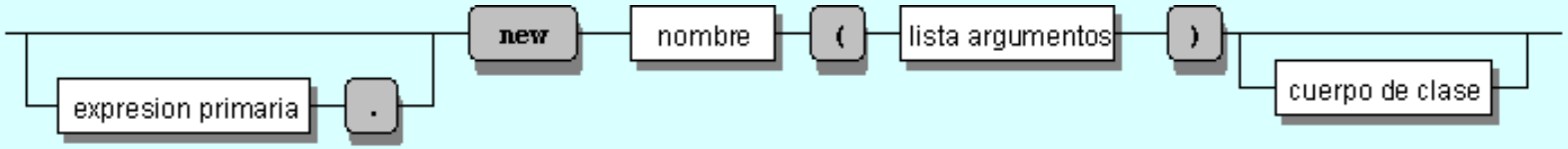


(*) excepto creación de array

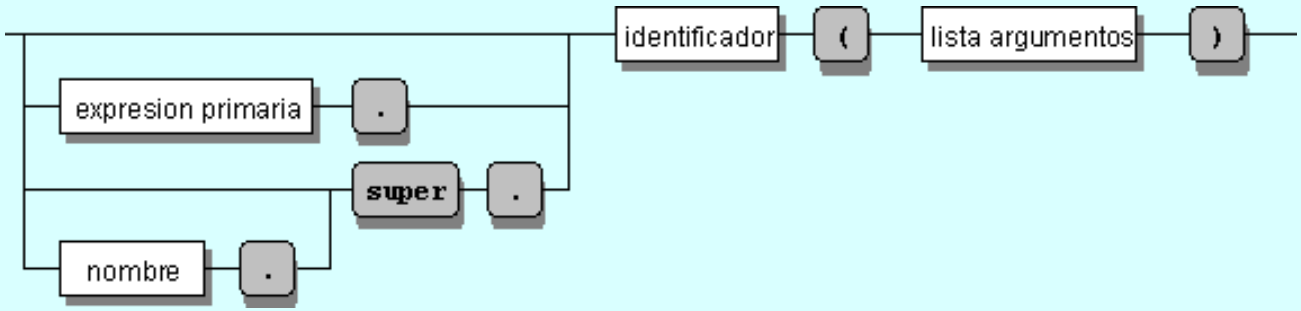
Acceso a campo



Creación de instancia



Invocación de método



Ultima actualización - 25-Marzo-2001

Antonio Bel Puchol - abelp@arrakis.es

22. Introducción API

22.1. Presentación API

El API de Java está formado por una amplísima jerarquía de clases que cubren una gran cantidad de aspectos relacionados con el desarrollo de software en general. Esta organizado en packages ordenados por temas. El J2SE (Java 2 Standard Edition) permite la utilización de todos estos packages en el desarrollo de programas Java y el JRE (Java Runtime Environment) permite la ejecución de programas que usan cualquiera de las clases del API.La documentación que acompaña al J2SE contiene un manual de referencia completo ordenado por packages y clases de todo el contenido del API. Su consulta resulta imprescindible para cualquier desarrollo.

El API de Java es vastísimo. La versión actual (1.3) contiene 76 packages y aproximadamente 2000 elementos entre clases, interfaces, excepciones, etc. Por tanto el conocimiento en profundidad del API no es tarea trivial, pero a la vez es imprescindible si se quieren abordar desarrollos extensos. La aproximación debe realizarse de forma progresiva, teniendo un conocimiento general de las capacidades globales del API, para ir adquiriendo una mayor especialización en función de las necesidades. Antes de intentar resolver cada problema de programación es conveniente reflexionar que packages del API pueden ayudarnos a resolverlo (si es que no está resuelto completamente).

La nomenclatura de los packages es uniforme y ayuda a categorizar las clases. El primer calificador es java o javax (para las clases dedicadas al interface gráfico). Escapan a esta norma las dedicadas a CORBA y materias afines. El segundo calificador da idea de la materia que cubre el package, como io (entrada/salida), math (funciones matemáticas). Hay temas que contienen varios subpackages, con un tercer calificador más específico (por ejemplo javax.sound.midi, que es bastante autoexplicativo).

El siguiente apartado da un somero repaso a los distintos grupos de packages del API, agrupándolos por temas.

22.2. Resumen del Contenido

La siguiente tabla muestra una descripción básica del contenido de los distintos packages que

constituyen el API de Java.

Package o grupo de packages	Descripción
java.applet	Proporciona las clases necesarias para crear applets, así como las clases que usa ul applet para comunicarse con su contexto.
java.awt.*	12 packages.El AWT (Abstract Windows Toolkit) proporciona el entorno base para todas las clases de manipulación del interfaz gráfico del usuario. El AWT apareció en la versión 1.0 del JDK y fue parcialmente sustuituido y sustancialmente mejorado en la versión 1.1 (con el conjunto de componentes conocido como swing). Actualmente se mantiene porque es la base del swing aunque muchos de sus elementos ya no se usan.
java.beans.*	2 packages. Contiene las clases relacionadas con el desarrollo de Java Beans. Los Java Beans representan la estrategia de desarrollo por componentes de Java.
java.io	Es un package fundamental. Proporciona los mecanismos para las operaciones de entrada/salida de flujos de datos (streams), así como la serialización (capacidad de los objetos para ser transformados en flujos de datos), y soporte para los sistemas de archivos.
java.lang.*	3 packages. Proporciona clases básicas para cualquier programa. (Threads, clases envoltorio, entrada/salida/error estándar, etc.). Asi como clases que proporcionan la caracteristica de 'reflexión', es decir la capacidad de las clases de averigurar como están construidas ellas mismas u otras clases.
java.math	Proporciona clases para realiza cálculos aritméticos de cualquier preción. Así como funciones matemáticas generales (Trigonometría, aleatorización, etc.)
java.net	Proporciona clases para implantar aplicaciones basadas en red. Sockets, URL's, direcciones IP, etc.
java.rmi.*	5 packages. RMI significa Remote Method Invocation. Proporciona clases que permite que los objetos puedan comunicarse con otros objetos en otros programas Java corriendo en otra máquina virtual conectada por red. Permite el desarrollo de aplicaciones distribuidas.
java.security.*	5 packages. Clases que implantan el esquema de seguridad de Java.
java.sql	Proporciona el API para el aceso y proceso de datos organizados en bases de datos relacionales y accediendo con mecanismos de lenguaje SQL. Esta API se denomina también JDBC (Java Data Base Conectivity)
java.text	Proporciona clases e interfaces para la manipulación de texto, fechas, números y mensajes de una forma independiente del idioma.

java.util.*	3 packages. Contiene la 'collections framework', o conjunto de clases para manipulación de conjuntos de objetos (colas, pilas, listas, diccionarios, árboles, tablas hash, etc.). Además tiene varios conjunts de utilidades para manipulación de fecha y hora, generación de números aleatorios y manipulación de ficheros comprimidos en formato ZIP y JAR. El formato JAR (Java ARchive) es una extensión del formato ZIP que permite empaquetar clases java compiladas para su ejecución.
javax.accesibility	Contiene un conjunto de clases para las tecnologías de asistencia relacionadas con los interfaces gráficos.
javax.naming.*	5 packages. Contiene clases e interfaces para proporcionar servicios de nombres y directorios.
javax.rmi.*	2 packages. Contiene clases relacionadas con el RMI-IIOP.
javax.sound.*	4 packages. Contiene clases para manipulación de sonidos.
javax.swing.*	16 packages. Conjunto extenso de clases para el configuación del interface gráfico de usuario. Reemplaza parcialmente al AWT. Su característica más importante es que es independiente de la plataforma.
org.omg.*	12 packages. Contiene clases relacionadas con la especificación CORBA. CORBA permite la comunicación entre programas de objetos escritos en diferentes lenguajes (Por ejemplo, Java y C++). Es equivalente a RMI, pero mientras que en este todos los participantes deben ser objetos Java con CORBA se posibilita la comunicación entre objetos multi-lenguaje.

Ultima actualización - 4-Febrero-2001

Antonio Bel Puchol - abelp@arrakis.es

23. Threads I

23.1. Qué es un Thread

La Máquina Virtual Java (JVM) es un sistema multi-thread. Es decir, es capaz de ejecutar varias secuencias de ejecución (programas) simultáneamente. La JVM gestiona todos los detalles, asignación de tiempos de ejecución, prioridades, etc, de forma similar a como gestiona un Sistema Operativo múltiples procesos. La diferencia básica entre un proceso de Sistema Operativo y un Thread Java es que los Threads corren dentro de la JVM, que es un proceso del Sistema Operativo y por tanto comparten todos los recursos, incluida la memoria y las variables y objetos allí definidos. A este tipo de procesos donde se comparte los recursos se les llama a veces 'procesos ligeros' (lightweight process).

Java da soporte al concepto de Thread desde el mismo lenguaje, con algunas clases e interfaces definidas en el package java.lang y con métodos específicos para la manipulación de Threads en la clase Object.

Desde el punto de vista de las aplicaciones los threads son útiles porque permiten que el flujo del programa sea divido en dos o más partes, cada una ocupándose de alguna tarea. Por ejemplo un Thread puede encargarse de la comunicación con el usuario, mientras otros actuan en segundo plano, realizando la transmisión de un fichero, accediendo a recursos del sistema (cargar sonidos, leer ficheros ...), etc. De hecho todos los programas con interface gráfico (AWT o Swing) son multithread porque los eventos y las rutinas de dibujado de las ventanas corren en un thread distinto al principal.

23.2. La Clase Thread

La forma más directa para hacer un programa multi-thread es extender la clase Thread, y redefinir el método run(). Este método es invocado cuando se inicia el thread (mediante una llamada al método start() de la clase thread). El thread se inicia con la llamada al método run y termina cuando termina éste. El ejemplo ilustra estas ideas:

```
public class ThreadEjemplo extends Thread {
    public ThreadEjemplo(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10 ; i++)
            System.out.println(i + " " +
getName());
        System.out.println("Termina thread " +
getName());
    }
    public static void main (String [] args) {
        new ThreadEjemplo("Pepe").start();
        new ThreadEjemplo("Juan").start();
        System.out.println("Termina thread
main");
    }
}
```

Compila y ejecuta el programa. La salida, será algo asi:

```
Termina thread main
0 Pepe
1 Pepe
2 Pepe
3 Pepe
0 Juan
4 Pepe
1 Juan
5 Pepe
2 Juan
6 Pepe
3 Juan
7 Pepe
4 Juan
8 Pepe
5 Juan
9 Pepe
6 Juan
Termina thread Pepe
7 Juan
8 Juan
9 Juan
Termina thread Juan
```

Ejecuta varias veces el programa. Verás que no siempre se ejecuta igual.

Notas sobre el programa:

- La clase Thread está en el package java.lang. Por tanto no es necesario el import.
- El constructor **`public`** `Thread(String str)` recibie un parámetro que es la identificación del Thread.
- El método run contiene el bloque de ejecución del Thread. Dentro de él, el método getName() devuelve el nombre del Thread (el que se ha pasado como argumento al constructor).
- El método main crea dos objetos de clase ThreadEjemplo y los inicia con la llamada al método start(). (el cual inicia el nuevo thread y llama al método run()).
- Observa en la salida el primer mensaje, de finalización del thread main. La ejecución de los threads es asíncrona. Realiza la llamada al método start(), éste le devuelve control y continua su ejecución, independiente de los otros threads.
- En la salida los mensajes de un thread y otro se van mezclando. La máquina virtual asigna tiempos a cada thread.

23.3. La Interface Runnable

La interface Runnable proporciona un método alternativo a la utilización de la clase Thread, para los casos en los que no es posible hacer que nuestra clase extienda la clase Thread. Esto ocurre cuando nuestra clase, que deseamos correr en un thread independiente deba extender alguna otra clase. Dado que no existe herencia múltiple, nuestra clase no puede extender a la vez la clase Thread y otra más. En este caso nuestra clase debe implantar la interface Runnable, variando ligeramente la forma en que se crean e inician los nuevos threads.

El siguiente ejemplo es equivalente al del apartado anterior, pero utilizando la interface Runnable:

```
public class ThreadEjemplo implements Runnable {
    public void run() {
        for (int i = 0; i < 5 ; i++)
            System.out.println(i + " " +
Thread.currentThread().getName());
        System.out.println("Termina thread " +
Thread.currentThread().getName());
    }
    public static void main (String [] args) {
        new Thread ( new ThreadEjemplo() ,
"Pepe").start();
        new Thread ( new ThreadEjemplo() ,
"Juan").start();
        System.out.println("Termina thread main");
    }
}
```

Observese en este caso:

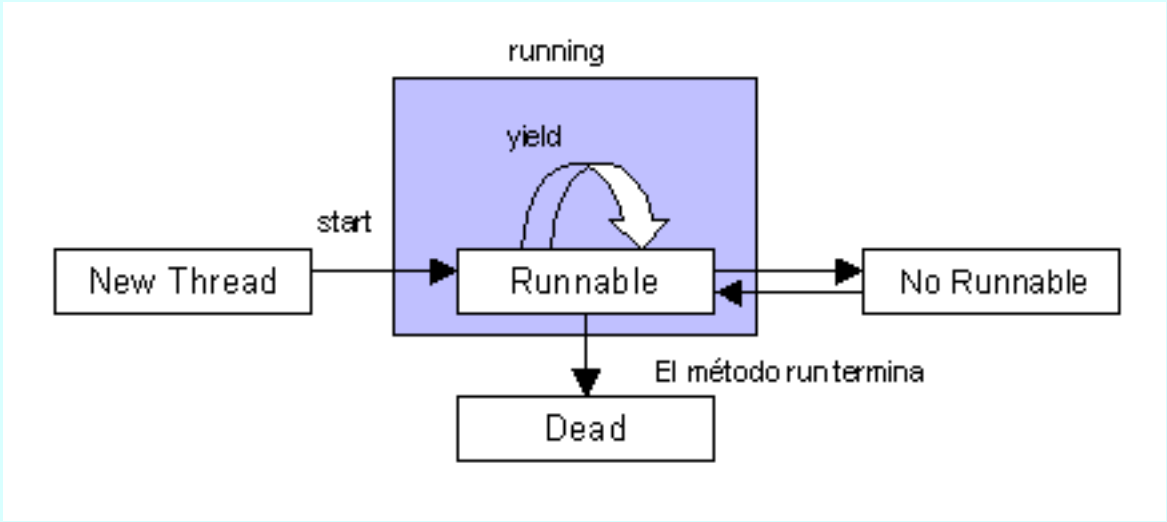
- Se implanta la interface Runnable en lugar de extender la clase Thread.
- El constructor que había antes no es necesario.
- En el main observa la forma en que se crea el thread. Esa expresión es equivalente a:

```
ThreadEjemplo ejemplo = new ThreadEjemplo();
Thread thread = new Thread ( ejemplo , "Pepe") ;
thread.start();
```

    - Primero se crea la instancia de nuestra clase.
    - Después se crea una instancia de la clase Thread, pasando como parámetros la referencia de nuestro objeto y el nombre del nuevo thread.
    - Por último se llama al método start de la clase thread. Este método iniciará el nuevo thread y llamará al método run() de nuestra clase.
- Por útlimo, obsérvese la llamada al método getName() desde run(). getName es un método de la clase Thread, por lo que nuestra clase debe obtener una referencia al thread propio. Es lo que hace el método estático currentThread() de la clase Thread.

23.4. El ciclo de vida de un Thread

El gráfico resume el ciclo de vida de un thread:



Cuando se instancia la clase Thread (o una subclase) se crea un nuevo Thread que está en en su estado inicial ('New Thread' en el gráfico). En este estado es simplemente un objeto más. No existe todavía el thread en ejecución. El único método que puede invocarse sobre él es el método start.

Cuando se invoca el método start sobre el thread el sistem crea los recursos necesarios, lo planifica (le asigna prioridad) y llama al método run. En este momento el thread está corriendo.

Si el método run invoca internamente el método sleep o wait o el thread tiene que esperar por una

operación de entrada/salida, entonces el thread pasa al estado 'no runnable' (no ejecutable) hasta que la condición de espera finalice. Durante este tiempo el sistema puede ceder control a otros threads activos.

Por último cuando el método run finaliza el thread termina y pasa a la situación 'Dead' (Muerto).

Ultima actualización - 18-Febrero-2001

Antonio Bel Puchol - abelp@arrakis.es

24. Threads II

24.1. Threads y prioridades

Aunque un programa utilice varios threads y aparentemente estos se ejecuten simultaneamente, el sistema ejecuta una sóla instrucción cada vez (esto es particularmente cierto en sistemas con una sola CPU), aunque realizado a velocidad suficiente para proporcionar la ilusión de simultaneidad. El mecanismo por el cual un sistema controla la ejecución concurrente de procesos se llama planificación (scheduling). Java soporta un mecanismo simple denominado planificacion por prioridad fija (fixed priority scheduling). Esto significa que la planificación de los threads se realiza en base a la prioridad relativa de un thread frente a las prioridades de otros.

La prioridad de un thread es un valor entero (cuanto mayor es el número, mayor es la prioridad), que puede asignarse con el método setPriority. Por defecto la prioridad de un thread es igual a la del thread que lo creó. Cuando hay varios threads en condiciones de ser ejecutados (estado runnable), la máquina virtual elige el thread que tiene una prioridad más alta, que se ejecutará hasta que:

- Un thread con una prioridad más alta esté en condiciones de ser ejecutado (runnable), o
- El thread termina (termina su metodo run), o
- Se detiene voluntariamente o
- Alguna condición hace que el thread no sea ejecutable (runnable), como una operación de entrada/salida o, si el sistema operativo tiene planificación por división de tiempos (time slicing), cuando expira el tiempo asignado.

Si dos o más threads están listos para ejecutarse y tienen la misma prioridad, la máquina virtual va cediendo control de forma cíclica (round-robin).

El hecho de que un thread con una prioridad más alta interrumpa a otro se denomina se denomina 'planificación con derecho preferente' (preemptive scheduling).

Cuando un thread entra en ejecución y no cede voluntariamente el control para que puedan ejecutarse otros threads, se dice que es un thread egoista (selfish thread). Algunos Sistemas Operativos, como Windows, combaten estas actitudes con una estrategia de planificación por división de tiempos (time-slicing), que opera con threads de igual prioridad que compiten por la CPU. En estas condiciones el Sistema Operativo asigna tiempos a cada thread y va cediendo el control consecutivamente a todos los que compiten por el control de la CPU, impidiendo que uno de ellos se apropie del sistema durante un intervalo de tiempo prolongado.

Este mecanismo lo proporciona el sistema operativo, no Java.

24.2. Sincronización de Threads

El ejemplo del capítulo anterior muestra un programa que ejecuta varios threads de forma asíncrona. Es decir, una vez que es iniciado, cada thread vive de forma independiente de los otros, no existe ninguna relación entre ellos, ni tampoco

ningún conflicto, dado que no comparten nada. Sin embargo, hay ocasiones que distintos threads en un programa si necesitan establecer alguna relación entre sí, o compartir objetos. Se necesita entonces algún mecanismo que permita sincronizar threads así como establecer unas 'reglas del juego' para acceder a recursos (objetos) compartidos.

Un ejemplo típico en que dos procesos necesitan sincronizarse es el caso en que un thread produzca algún tipo de información que es procesada por otro thread. Al primer thread le denominaremos productor y al segundo, consumidor. El productor podría tener el siguiente aspecto:

```
public class Productor extends Thread {
    private Contenedor contenedor;

    public Productor (Contenedor c) {
        contenedor = c;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            contenedor.put(i);
            System.out.println("Productor. put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

Productor tiene una variables miembro: contenedor es una referencia a un objeto Contenedor, que sirve para almacenar los datos que va produciendo. El método run genera aleatoriamente el dato y lo coloca en el contenedor con el método put. Después espera una cantidad de tiempo aleatoria (hasta 100 milisegundos) con el método sleep. Productor no se preocupa de si el dato ya ha sido consumido o no. Simplemente lo coloca en el contenedor.

El consumidor, por su parte podría tener el siguiente aspecto:

```
public class Consumidor extends Thread {
        private Contenedor contenedor;

        public Consumidor (Contenedor c) {
                contenedor= c;
        }

        public void run() {
                int value = 0;
                for (int i = 0; i < 10; i++) {
                        value = contenedor.get();
                        System.out.println("Consumidor. get: " + value);
                }
        }
}
```

El constructor es equivalente al del Productor. El método run, simplemente recupera el dato del contenedor con el método get y lo muestra en la consola. Tampoco el consumidor se preocupa de si el dato está ya disponible en el contenedor o no.

Productor y Consumidor se usarían desde un método main de la siguiente forma:

```
public class ProducTorConsumidorTest {
    public static void main(String[] args) {
        Contenedor c = new Contenedor ();
        Productor produce = new Productor (c);
        Consumidor consume = new Consumidor (c);

        produce.start();
        consume.start();
    }
}
```

Simplemente se crean los objetos, contendor, productor y consumidor y se inician los threads de estos dos últimos.

La sincronización que permite a productor y consumidor operar correctamente, es decir que hace que consumidor espere hasta que haya un dato disponible, y que productor no genere uno nuevo hasta que haya sido consumido esta en la clase Contenedor, que tiene el siguiente aspecto:

```
public class CubbyHole {
        private int dato;
        private boolean hayDato = false;

        public synchronized int get() {
                while (hayDato == false) {
                        try {
                                // espera a que el productor coloque un valor
                                wait(); }
                        catch (InterruptedException e) { }
                }
                hayDato = false;
                // notificar que el valor ha sido consumido
                notifyAll();
                return dato;
        }

        public synchronized void put(int valor) {
                while (hayDato == true) {
                        try {
                                // espera a que se consuma el dato
                                wait();
                        } catch (InterruptedException e) { }
                }
                dato = valor;
                hayDato = true;
                // notificar que ya hay dato.
                notifyAll();
        }
}
```

La variable miembro dato es la que contiene el valor que se almacena con put y se devuelve con get. La variable miembro hayDato es un flag interno que indica si el objeto contiene dato o no.

En el método put, antes de almacenar el valor en dato hay que asegurarse de que el valor anterior ha sido consumido. Si todavía hay valor (hayDato es true) se suspende la ejecución del thread mediante el método wait. Invocando wait (que es un método de la clase Object) se suspende el thread indefinidamente hasta que alguien le envíe una 'señal' con el método notify o notifyAll. Cuando esto se produce (veremos que el notify lo produce el método get) el método continua, asume

que el dato ya se ha consumido, almacena el valor en dato y envia a su vez un notifyAll para notificar a su vez que hay un dato disponible.

Por su parte, el método get chequea si hay dato disponible (no lo hay si hayDato es false) y si no lo hay espera hasta que le avisen (método wait). Una vez ha sido notificado (desde el método put) cambia el flag y devuelve el dato, pero antes notifica a put de que el dato ya ha sido consumido, y por tanto se puede almacenar otro.

La sincronización se lleva a cabo pues usando los métodos wait y notifiAll.

Existe además otro componente básico en el ejemplo. Los objetos productor y consumidor utilizan un recurso compartido que es el objeto contenedor. Si mientras el productor llama al método put y este se encuentra cambiando las variables miembro dato y hayDato, el consumidor llamara al método get y este a su vez empezara a cambiar estos valores podrían producirse resultados inesperados (este ejemplo es sencillo pero fácilmente pueden imaginarse otras situaciones más complejas).

Interesa, por tanto que mientras se esté ejecutando el método put nadie más acceda a las variables miembro del objeto. Esto se consigue con la palabra synchronized en la declaración del método. Cuando la máquina virtual inicia la ejecución de un método con este modificador adquiere un bloqueo en el objeto sobre el que se ejecuta el método que impide que nadie más inicie la ejecución en ese objeto de otro método que también esté declarado como syncrhonized. En nuestro ejemplo cuando comienza el método put se bloquea el objeto de tal forma que si alguien intenta invocar el método get o put (ambos son synchronized) quedará en espera hasta que el bloqueo se libere (cuando termine la ejecución del método). Este mecanismo garantiza que los objetos compartidos mantienen la consistencia.

Este método de gestionar los bloqueos implica que:

- Es responsabilidad del programador pensar y gestionar los bloqueos (A veces es una pesada responsabilidad).
- Los métodos synchronized son más costosos en el sentido de que adquirir y liberar los bloqueos consume tiempo (este es el motivo por el que no están sincronizados por defecto todos los métodos).
- Conviene evitar en lo posible el uso de objetos compartidos. Resultan difíciles de manejar.

Nota: El magnifico ejemplo que ilustra este capítulo no es, desgraciadamente, mio. Esta extraido del 'Java Tutorial', que contiene muy buenos ejemplos sobre muchas materias y es un punto de referencia obligada en la documentación sobre Java. Puedes consultarlo online o descargarlo en http://java.sun.com/docs/books/tutorial/

Ultima actualización - 17-Junio-2001

Antonio Bel Puchol - abelp@arrakis.es

Contenido

¿Qué es StarMap?

StarMap es un programa que presenta un mapa a escala del espacio próximo alrededor de nuestro Sol, en un radio aproximado de 80 años luz. Contiene las posiciones, nombres y características más importantes de unas 3800 estrellas. Con StarMap se puede:

- desplazar el visor del mapa en las dimensiones,
- averiguar distancias entre dos estrellas cualquiera,
- ver los datos del catálogo de cada estrella,
- obtener las estrellas más próximas a una dada,
- buscar estrellas por su nombre,
- personalizar los nombres de las estrellas, etc.

StarMap muestra las estrellas por sus coordenadas reales cartesianas (X, Y y Z) y no por sus posiciones aparentes desde la Tierra. El sistema de coordenadas está centrado en el Sol (sus coordenadas son 0, 0, 0). Es decir no es una carta estelar al uso, que pueda usarse para observar las estrellas desde la Tierra; es más bien un 'mapa de carreteras', con el que podríamos responder a preguntas tales como '¿Por donde debo pasar si quiero llegar a Epsilón Eridani y mi nave debe repostar cada 6 años luz?'.

StarMap está basado en el catálogo estelar Gliese-3 (Preliminary Version of the Third Catalogue of Nearby Stars. Gliese W., Jahreiss H. Astron. Rechen-Institut, Heidelberg, 1991), que contiene los datos fundamentales de todas las estrellas conocidas en un radio de 25 parsecs (81.5 años luz).

StarMap es un programa escrito en Java que puede funcionar como applet o como programa independiente, siendo su funcionamiento equivalente (casi). Dado que está escrito en Java 2 requiere tener instalado, bien el Java Plug-in 1.2 o 1.3, para funcionar como applet o el JRE 1.2 o 1.3 para funcionar como programa independiente. Ambos programas se instalan conjuntamente y pueden obtenerse gratuitamente en http://java.sun.com/j2se/1.3/jre/download-windows.html. (Versión Windows). Para Solaris o Linux puedes dirigirte a http://java.sun.com/products/jdk/1.2/jre/index.html

---

Lanzar StarMap (Applet)

Para usar StasrMap como Applet debes tener instalado el Java Plug-in 1.2 o 1.3. Si no lo tienes verás un mensaje de aviso al intentar iniciar el Applet. Puedes descargarlo en http://java.sun.com/j2se/1.3/jre/download-windows.html . (Versión Windows). El proceso de instalación es muy sencillo. El Applet ocupa unas 300 Kb. por lo que tardará un poco en iniciarse. Si quieres usarlo regularmente te recomiendo que te descargues el programa. Pulsa en la imagen para iniciar el Applet, que se ejecutará en una ventana distinta a la del navegador. ¡Buen viaje!

Descargar StarMap (Programa)

Para usar StarMap como un programa independiente, sin necesidad de estar conectado a Internet haz lo siguiente:

1. Descarga el comprimido starmap.zip

2. Crea un directorio. Por ejemplo starmap.

3. Descomprime starmap.zip en ese directorio utilizando alguna Winzip, pkzip o algún otro producto similar.

4. Abre una ventana DOS y cambiate al directorio en cuestión (CD C:\STARMAP si has creado el directorio en el disco C)

5. Inicia StarMap con starmap.bat

Uso de StarMap

Cuando se activa StarMap aparece la Ventana Principal del programa. Puedes ver una imagen de la Ventana Principal aquí. Actua como contenedor para el resto de las ventanas que son las siguientes:

Ventana del Mapa
Ventana de Control
Ventana de Información
Ventana de Distancias
Ventana de Búsqueda
Ventana de Asignación de Nombres
Ventana de Ayuda

Acerca del Autor

StarMap ha sido concebido, diseñado y programado por Antonio Bel en sus ratos libres como parte de un proceso de aprendizaje del lenguaje Java y tecnologías afines.

Me gustaría conocer tu opinión sobre el programa, incluso si no es buena, en abelp@arrakis.es

Algunas direcciones

Aquí tienes algunas direcciones que he visitado frecuentemente para escribir StarMap:

http://java.sun.com : Si estás interesado en algo relacionado con el mundo Java aquí tienes la respuesta. Seguro. Software, documentación, cursos, referencias. Imprescindible.

http://www.clark.net/pub/nyrath/starmap.html#contents : Todo sobre mapas estelares en 3 y 4D. Referencias sobre temas astronómicos, ciencia-ficción, catálogos estelares, cálculos de coordenadas y mucho más.

ftp://adc.gsfc.nasa.gov/pub/adc/archives/catalogs/5/5070A/catalog.dat.gz Aquí puedes encontrar el Catálogo Gliese-3 usado en StarMap.

http://wdvl.internet.com/Authoring/Tutorials/ : Documentaciones variadas sobre HTML, Java, JavaScript, etc.

http://www.stsci.edu/astroweb/net-www.html : Muchas, muchas direcciones sobre temas de astronomía.

http://www.w3.org/ : Información sobre disintos temas relacionados con la WWW. Además puedes descargarte aquí Amaya, la herramienta de composición de páginas Web que he usado para escribir esto.

jEscoba

El juego de la Escoba.
Una aplicación Java multijugador para Internet.

http://www.arrakis.es/~abelp/escoba/indice.html

2000 - Antonio Bel Puchol
abelp@arrakis.es

jRabino - El juego del Rabino Francés

http://www.arrakis.es./~abelp/rabino/rabino.htm
2000 - Antonio Bel Puchol
abelp@arrakis.es

Colección de solitarios

jSol

Ultima modificación: 01 - Junio - 2002
Versión actual: 2.1

English version

Novedades

- 01 - Junio - 2002. Versión 2.1 disponible.
- 29 - Abril - 2002. Versión 2.0 disponible.
- 09 - Diciembre - 2001. Versión 1.2 disponible.
- 13 - Junio - 2001. Versión 1.1 disponible.
- 27 - Mayo -2001. Versión 1.0 disponible.
- 13 - Mayo -2001. Versión 0.2 disponible.
- 6 - Mayo -2001. Versión 0.1 disponible.
- 29 - Abril - 2001 . Versión inicial 0.0. Página Web creada

Resumen

jSol es una colección de solitarios de cartas. La versión actual es la 2.1, que dispone de 61 solitarios (Klondike, Free Cell, 40 ladrones, Yukon, Gypsy, Canfield y otros muchos). Sus características son.

- Agrupación de los solitarios por tipos o familias
- Posibilidades de deshacer / rehacer ilimitadas
- Ayuda con las reglas de cada juego.
- Estadísticas de solitarios completados por usuario.
- Solitarios configurables por el usuario. Posibilidades de crear nuevos

juegos.
- Versiones en español e inglés. (Detección automática del idioma).

jSol está escrito totalmente en Java, por lo que es instalable en cualquier plataforma que soporte este entorno. Consulta en el apartado Descarga instalación los detalles.

jSol es gratuito. Descarga, instala y diviértete !!!

Crear nuevos solitarios?

La posibilidad de crear nuevos solitarios es uno de los aspectos destacables de **jSol**. No solo puedes experimentar una amplia variedad de juegos, sino que además puedes crear tus propias versiones, o crear juegos enteramente nuevos. Consulta la documentación 'Creación de nuevos solitarios en jSol' para conocer como puedes definir tu mismo tus solitarios favoritos. Los solitarios se definen en un fichero de configuración con un sencillo lenguaje donde se establecen todos los aspectos del solitario, la disposición inicial, reglas de movimiento de cartas, condiciones de victoria, etc.

Envia tus definiciones de nuevos solitarios a abelp@arrakis.es. Serán incorporados en versiones sucesivas.

Descarga

Puedes descargar jSol en cualquiera de estos dos formatos:

- Instalable para Windows: jSol-windows-install.exe (791045 bytes.)
- Formato zip (Todas las plataformas): jSol.zip (506961 bytes)

Para usar jSol debes tener instalado en tu sistema soporte Java 2, que es proporcionado por el producto JRE (Java Runtime Environment).
Puedes descargar este producto gratuitamente en las siguientes direcciones, dependiendo de tu sistema operativo:

Entorno Windows:
http://www.softonic.com/informacion_extendida.phtml?n_id=9160&plat=1

Todos los entornos: http://java.sun.com/j2se/1.4/download.html

---

Instalación

1. Instala primero el **JRE**, si no lo tienes ya: En Windows, el JRE es un archivo ejecutable. Ejecútalo haciendo doble click sobre él y sigue las instrucciones. Con esto instalas soporte Java 2 en tu sistema, incluyendo la ejecución de applets Java 2 en tu navegador (Java plug-in). Este paso sólo lo tienes que hacer la primera vez.
2. Si tienes Windows y has descargado el ejecutable (jSol-windows-install.exe) haz doble click y sigue las instrucciones. El instalador te informará de todo lo necesario, creará iconos en el escritorio y registrará la aplicación para poder desinstalarla. Si tienes Windows es recomendable seguir este procedimiento, aunque el tamaño del descargable es ligeramente mayor.
3. Si has descargado el zip (jSol.zip) descomprimelo sobre un directorio de tu elección. Si no tienes winzip, puedes descargartelo gratuitamente en http://www.winzip.com. Puedes inicial **jSol** usando el archivo **jSol.bat** que hay en la carpeta bin.
4. Si tienes otro sistema operativo diferente de Windows deberás adaptar **jSol.bat** para hacerlo funcionar.

jSol intenta reconocer el lenguaje que utilizas. Si tu sistema usa inglés, la aplicación se inicia con los textos y mensajes en inglés. En cualquier otro caso la aplicación se inicia en español. Si prefieres elegir el idioma puedes utilizar los archivos **jSol_xx.bat** que están en la carpeta bin, donde xx es '**es**' para español y '**en**' para inglés.

Prueba online

Puedes probar jSol ahora mismo, en modo applet, sin necesidad de descargar e instalar el producto completo. Los juegos en modo applet tienen algunas limitaciones que no existen en el juego completo (corresponden a la versión 0.0 de jSol). En modo applet, hay dos juegos disponibles:

- Klondike
- Doble klondike (Igual que el anterior pero con dos barajas).

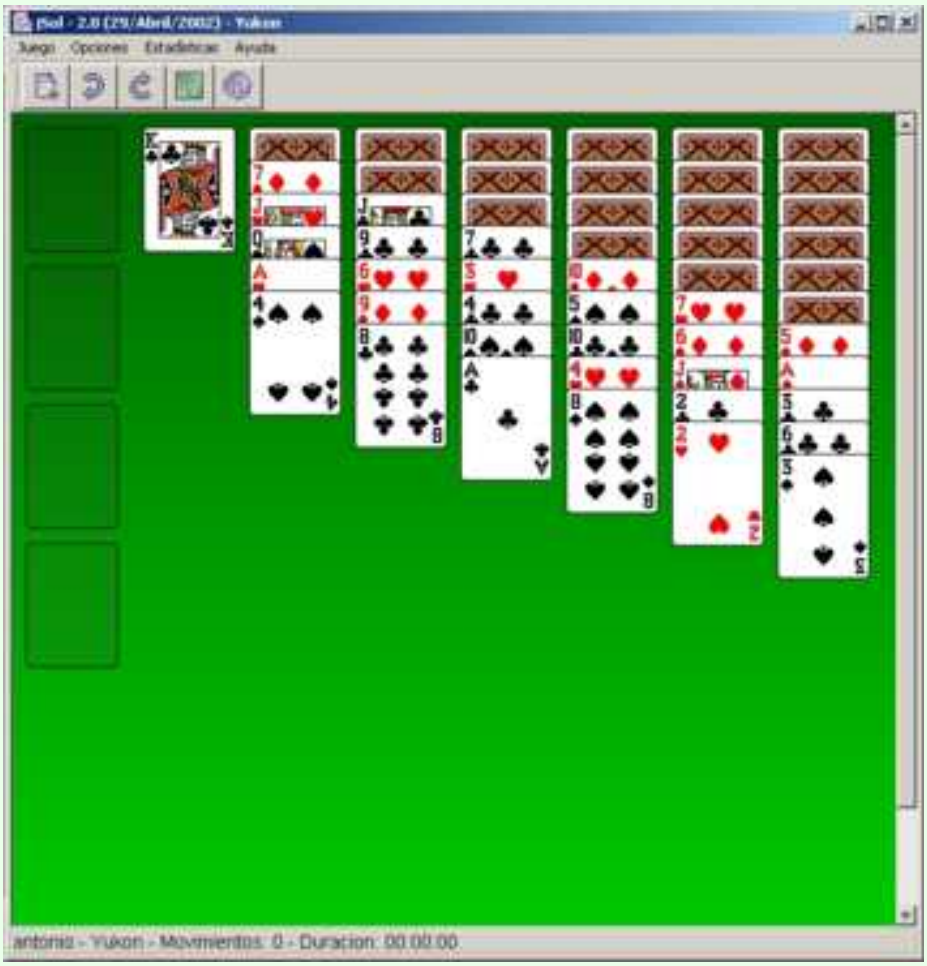
(Para que funcionen en modo applet necesitas tener instalado el JRE. Consulta la

sección descarga).

Nota: Probablemente los applets sólo te funcionaran si usas Microsoft Internet Explorer (cualquier versión) o Netscape Navigator (cualquier versión) con el JRE instalado.

Capturas de pantalla

Este es el aspecto de jSol. Seguramente te resultará familiar.



Colaboración

Puedes colaborar en el desarrollo de jSol de varias formas:

- Comunicando problemas, errores, manlfuncionamiento que detectes o haciendo sugerencias, comentarios o críticas que ayuden a mejorar jSol.
- Comentando solitarios que se puedan incorporar en jSol. Estoy

particularmente interesado en solitarios de origen español o sudamericano. Dime como se juega e intentaré incorporarlo a jSol.
- Contribuyendo con gráficos, juegos de cartas o imágenes que se puedan incorporar a jSol.
- Enviando tus definiciones de nuevos solitarios para incorporarlos en nuevas versiones.

Para cualquiera de estas posibilidades ponte en contacto conmigo en abelp@arrakis.es. ¡ Cualquier comentario será bien recibido !

Cambios

- Versión 2.1 (01-Junio-2002)

  - Mejora de la interfaz gráfica. Movimiento automático de las cartas al hacer doble click o al deshacer o rehacer para facilitar la jugabilidad.
  - Nuevo icono para la aplicación.
  - Corrección de errores detectados en la versión 2.0.

- Versión 2.0 (29-Abril-2002)

  - 21 nuevos solitarios: Tipo Canfield, Numérico y otros.
  - Estadísticas de solitarios resueltos por usuario.
  - Mejora de la presentación. Aspecto Windows y fondo en gradiente.
  - Visualización de la consola Java en una ventana independiente.
  - Instalador para Windows.

- Versión 1.2 (09-Diciembre-2001)

  - Nuevos tipos de solitarios: Golf, Baker's Dozen y Fan, hasta un total de 40 solitarios y variantes.
  - Documentación para la creación de solitarios en jSol.
  - Modificado diálogo inicial para selección de solitarios. (Lista de familias y lista de variantes).

- Versión 1.1 (13-Junio-2001)

  - Añadido soporte muli lenguaje.
  - Traducción de la aplicación, reglas y página Web al inglés.

- Versión 1.0 (27-Mayo-2001)

- 7 Solitarios nuevos. (Tipo 40 ladrones, Yukon, Gypsy y variantes).
- Parametrización interna completamente renovada. Definición de solitarios usando XML.

- Versión 0.2 (13-Mayo-2001)

  - 4 Solitarios nuevos. (Free Cell y algunas variantes).
  - Barra de herramientas incorporada.
  - Mejoras en la parametrización interna.

- Versión 0.1 (06-Mayo-2001)

  - 12 Solitarios nuevos. (Variaciones del Klondike).
  - Posibilidad de deshacer y rehacer ilimitada.
  - Ayuda con las reglas de cada solitario e índice general de solitarios.

- Versión 0.0 (29-Abril-2001)

  - Dos solitarios (Klondike y doble klondike)
  - Entorno mínimo para el juego: Selección de solitarios, reinicio y abandono.

Acerca del autor

jSol es desarrollado y mantenido por Antonio Bel Puchol. Puedes contactar conmigo en abelp@arrakis.es. Puedes ver otras páginas mías aquí.

2001. Antonio Bel Puchol
abelp@arrakis.es