Anexo III

Lenguaje JAVA.

Guion-resumen

1. **Introducción a JAVA**
2. **Comentarios**
3. **Identificadores**
4. **Variables**
5. **Operadores**
6. **Separadores**
7. **Sentencias o expresiones**
8. **Arrays y cadenas**
9. **Clases en JAVA**
10. **Entrada / Salida**
11. **Herencia**
12. **Clases abstractas**
13. **La clase Object**
14. **Interface**
15. **Paquetes**
16. **Excepciones**
17. **HILOs - Threads**
18. **Interfaz gráfico AWT (Abstract Window Toolkit)**
19. **Paseando por la Red**
20. **Los Sockets**
21. **El JAVA Development Kit**

1. Introducción a JAVA

JAVA es un lenguaje de programación desarrollado por un grupo de ingenieros de Sun Microsystems (1991-1995). **Es un lenguaje orientado a objetos**, a diferencia de otros lenguajes que son lenguajes modificados para poder trabajar con objetos. En un principio fue denominado "Oak", se le puso el nombre de JAVA en 1995.

La estructura de un programa realizado en cualquier lenguaje orientado a objetos (Object Oriented Programming) (OOP), y en particular en el lenguaje JAVA es una clase. En JAVA todo forma parte de una clase, es una clase o describe cómo funciona una clase. El conocimiento de las clases es fundamental para poder entender los programas JAVA. Todas las acciones de los programas JAVA se colocan dentro del bloque de una clase. Todos los métodos se definen dentro del bloque de la clase, JAVA no soporta funciones o variables globales. En todo programa nos encontramos con una clase que contiene el programa principal y algunas clases de usuario (las específicas de la aplicación que se está desarrollando) que son utilizadas por el programa principal.

Los ficheros fuente tienen la extensión *.java, mientras que los ficheros compilados tienen la extensión *.class. Un fichero fuente (*.java) puede contener más de una clase, pero solo una puede ser public. El nombre del fichero fuente debe coincidir con el de la clase public (con la extensión *.java), es decir, si por ejemplo en un fichero aparece la declaración (public class MiClase {...}) entonces el nombre del fichero deberá ser MiClase.java. Es importante que coincidan mayúsculas y minúsculas ya que MiClase.java y miclase.java serían clases diferentes para JAVA. Si la clase no es public, no es necesario que su nombre coincida con el del fichero. Una clase puede ser public o package (default), pero no private o protected.

De ordinario una aplicación está constituida por varios ficheros *.class. Cada clase realiza unas funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. Las clases de JAVA se agrupan en packages, que son librerías de clases. Si las clases no se definen como pertenecientes a un package, se utiliza un package por defecto (default) que es el directorio activo.

Es necesario entender y dominar la sintaxis utilizada en la programación; observemos nuestro primer programa en JAVA y un breve comentario de las partes que lo componen, para posteriormente pasar a estudiar la nomenclatura empleada y los elementos que empleamos para desarrollar nuestro lenguaje:

```
/*    lolo.java
          Autor NombreyApellido  2005
          Escribe en pantalla «¡Hola, mundo!»
*/
          class lolo {
          public static void main(String args[]) {
                    System.out.println(«¡Hola, mundo!»);
          }
}
```

Analicemos el programa anterior:

Comentario: /* Lolo.java

Autor NombreyApellido 2005

Escribe en pantalla «¡Hola, mundo!»

*/

Definición de clase Lolo: class Lolo.

Define método main: public static void main(String args[]).

Accesible para todos: public.

Método de clase: static.

Tipo devuelto. (No devuelve nada): void.

Punto de entrada de los programas JAVA: main.

Argumentos en línea de comandos: (String args[]).

Clase: System.

Objeto estático de System (salida estándar): out (flujo).

Método de "out" (escribe en línea): println.

JAVA es un lenguaje que ha sido diseñado para producir software:

— Confiable: minimiza los errores que se escapan a la fase de prueba.

— Multiplataforma: los mismos binarios funcionan correctamente en Windows, Unix y Power/Mac.

— Seguro: applets recuperados por medio de la red no pueden causar daño a los usuarios.

— Orientado a objetos: beneficioso tanto para el proveedor de bibliotecas de clases como para el programador de aplicaciones.

— Robusto: los errores se detectan en el momento de producirse, lo que facilita la depuración.

Entre las características que nombramos nos referimos a la robustez. JAVA no permite el manejo directo del hardware ni de la memoria. El intérprete siempre tiene el control. El compilador es suficientemente inteligente como para no permitir un montón de cosas que podrían traer problemas, como usar variables sin inicializarlas, modificar valores de punteros directamente, acceder a métodos o variables en forma incorrecta, utilizar herencia múltiple, etc. Además, JAVA implemen-

ta mecanismos de seguridad que limitan el acceso a recursos de las máquinas donde se ejecuta, especialmente en el caso de los applets (que son aplicaciones que se cargan desde un servidor y se ejecutan en el cliente). También está diseñado específicamente para trabajar sobre una red, de modo que incorpora objetos que permiten acceder a archivos en forma remota (via URL por ejemplo). Además, con el JDK (JAVA Development Kit) vienen incorporadas muchas herramientas, entre ellas un generador automático de documentación que, al poner los comentarios adecuados en las clases, crea inclusive toda la documentación de las mismas en formato HTML.

2. Comentarios

Los comentarios son aquellas aclaraciones que el programador introduce en el lenguaje dirigido al propio programador como recordatorio o a futuros programadores que deseen variar el programa, con el fin de facilitar la comprensión del mismo; el ordenador ignorará dichos comentarios.

```
// Comentarios para una sola línea
/* Comentarios de una o
      más líneas
*/
      /** Comentario que va a ir en ejecutable (Comentario de documentación)  */
```

3. Identificadores

Los identificadores se utilizan para nombrar variables, funciones, clases y objetos, o cualquier elemento que se necesite identificar o utilizar.

Los identificadores utilizados en JAVA comienzan con una letra, un subrayado (_) o un símbolo de dólar ($), nunca por un dígito u otro carácter gráfico (@, #, ...). El resto de los caracteres que componen los identificadores son o bien letras o bien dígitos o una combinación de ambos y no existe una longitud máxima. Serían identificadores válidos:

```
nombre
nombre_y_apellidos
Nombre_Usuarios
_variable_interna
$moneda
```

JAVA tiene una serie de palabras clave. Estas palabras no se pueden utilizar como identificadores. La siguiente lista de palabras incluye todas aquellas que son consideradas palabras clave por JAVA:

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

Además de estas palabras clave, JAVA se ha reservado otras ocho palabras que no pueden ser utilizadas como identificadores; se presentan a continuación y por el momento no tienen ningún cometido específico.

cast	**future**	**generic**	**inner**
operator	**outer**	**rest**	**var**

4. Variables

Son variables aquellos identificadores que a lo largo del programa pueden variar su valor, bien por el usuario, bien por el propio programa. JAVA utiliza cinco tipos de variables: enteros, reales en coma flotante, booleanos, caracteres y cadenas. Las variables se pueden definir y utilizar en cualquier parte del código. Cada variable define un tipo de elementos con un rango perfectamente definido, siempre dentro de la clase.

4.1. Declaración de variables locales

Las variables locales se declaran igual que los atributos de la clase:

Tipo NombreVariable [= Valor];

Ej:

int suma;

float precio;

Contador obj;

Solo que aquí no se declaran private, public, etc., sino que las variables definidas dentro del método solo son accesibles por él. Las variables pueden inicializarse al crearse:

Ej: int suma = 0;

float precio = 12.3;

Contador obj = new Contador ( );

4.2. Asignaciones a variables

Se asigna un valor a una variable mediante el signo =: Variable = Constante | Expresión;

Ej: suma = suma + 1;

*precio = 1.05 * precio;*

obj.cnt = 0;

Todas las variables en el lenguaje JAVA deben tener un tipo de dato. El tipo de la variable determina los valores que la variable puede contener y las operaciones que se pueden realizar con ella.

Existen dos categorías de datos principales en el lenguaje JAVA: los tipos primitivos y los tipos referenciados.

Los tipos primitivos contienen un solo valor e incluyen los tipos como los enteros, coma flotante, los caracteres, etc...

La tabla siguiente muestra todos los tipos primitivos soportados por el lenguaje JAVA, su formato, su tamaño y una breve descripción de cada uno:

TIPO	TAMAÑO/FORMATO	DESCRIPCIÓN
byte	8-bit complemento a 2	Entero de un Byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
float	32-bit IEEE 754	Coma flotante de precisión simple
double	64-bit IEEE 754	Coma flotante de precisión doble
char	16-bit Carácter	Un solo carácter
boolean	true o false	Un valor booleano (verdadero o falso)

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

Por convención, como ya hemos comentado, en JAVA los nombres de las variables empiezan con una letra minúscula (los nombres de las clases empiezan con una letra mayúscula).

5. Operadores

Los operadores utilizados por JAVA son parecidos a los utilizados en operaciones algebraicas y lógicas. A continuación se muestran los operadores que maneja JAVA por orden de precedencia:

.	[]	()				
++	—					
!	~	instanceof				
*	/	%				
+	-					
<<	>>	>>>				
<	>	<=	>=	==	!=	
&	^	|				
&&	||					
?	:					
=	op=	(*=	/=	%=	+=	-=

Los operadores con una precedencia más alta se evalúan primero. Por ejemplo, el operador división tiene una precedencia mayor que el operador suma, por eso, en la expresión anterior **x + y / 100**, el compilador evaluará primero **y / 100**.

El operador = hace copias de objetos, marcando los antiguos para borrarlo, y el garbage collector se encargará de devolver al sistema la memoria ocupada por el objeto eliminado.

Los operadores realizan algunas funciones en uno o dos operandos. Los operadores que requieren un operador se llaman operadores unarios. Por ejemplo, ++ es un operador unario que incrementa el valor operando en uno. Los operadores que requieren dos operandos se llaman operadores binarios. El operador = es un operador binario que asigna un valor del operando derecho al operando izquierdo. Los operadores unarios en JAVA pueden utilizar la notación de prefijo o de sufijo. La notación de prefijo significa que el operador aparece antes de su operando: **operador operando.**

La notación de sufijo significa que el operador aparece después de su operando: **operando operador** . Los operadores binarios de JAVA tienen la misma notación, es decir, aparecen entre los dos operandos: **op1 operator op2.**

Además de realizar una operación también devuelve un valor. El valor y su tipo dependen del tipo del operador y del tipo de sus operandos. Por ejemplo,

los operadores aritméticos (realizan las operaciones de aritmética básica como la suma o la resta) devuelven números, el resultado típico de las operaciones aritméticas. El tipo de datos devuelto por los operadores aritméticos depende del tipo de sus operandos: si sumas dos enteros, obtendrás un entero. Se dice que una operación evalúa su resultado.

Los operadores JAVA se dividen en las siguientes categorías: aritméticos, relacionales y condicionales, lógicos y de desplazamiento y de asignación.

A) Operadores aritméticos

OPERADOR	USO	DESCRIPCIÓN
+	op1 + op2	Suma op1 y op2
-	op1 - op2	Resta op2 de op1
*	op1 * op2	Multiplica op1 y op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Obtiene el resto de dividir op1 por op2

Nota: El lenguaje JAVA extiende la definición del operador+ para incluir la concatenación de cadenas.

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

OPERADOR	USO	DESCRIPCIÓN
+	+ op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de atajos aritméticos: ++ que incrementa en uno su operando, y — que decrementa en uno el valor de su operando.

OPERADOR	USO	DESCRIPCIÓN
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
—	op —	Decrementa op en 1; evalúa el valor antes de decrementar
—	— op	Decrementa op en 1; evalúa el valor después de decrementar

B) Operadores relacionales y condicionales

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, != devuelve true si los dos operandos son distintos.

OPERADOR	USO	DEVUELVE TRUE SI
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Aquí tiene tres operadores condicionales:

OPERADOR	USO	DEVUELVE TRUE SI
&&	op1 && op2	op1 y op2 son verdaderos
||	op1 || op2	uno de los dos es verdadero
!	! op	op es falso

El operador & se puede utilizar como un sinónimo de && si ambos operadores son booleanos. Similarmente, | es un sinónimo de || si ambos operandos son booleanos.

C) Operadores de desplazamiento

OPERADOR	USO	DESCRIPCIÓN
>>	op1 >> op2	Desplaza a la derecha op2 bits de op1
<<	op1 << op2	Desplaza a la izquierda op2 bits de op1
>>>	op1 >>> op2	Desplaza a la derecha op2 bits de op1 (sin signo)
&	op1 & op2	Bitwise and
|	op1 | op2	Bitwise or
^	op1 ^ op2	Bitwise xor
~	~ op	Bitwise complemento

D) Operadores de asignación

Puede utilizar el operador de asignación =, para asignar un valor a otro. Además del operador de asignación básico, JAVA proporciona varios operadores de asignación que permiten realizar operaciones aritméticas, lógicas o de bits y una operación de asignación al mismo tiempo.

6. Separadores

En la gramática de JAVA también hay definidos una serie de separadores simples, que definen la forma y función del código. Estos separadores son:

()paréntesis	Se utilizan para identificar listas de parámetros utilizados en la definición y llamada a métodos. También se utilizan para definir el orden de precedencia en expresiones, para delimitar expresiones en situaciones de control de flujo y rodear las conversiones de tipo.
{}llaves	Contienen los valores de las matrices inicializadas automáticamente. También se utiliza para definir un bloque de código, para clases, métodos y ámbitos locales.
[] corchetes	Para declarar tipos matriz. También se utiliza cuando se referencian valores de matriz.
;punto y coma	Separa sentencias.
,coma	Separa identificadores consecutivos en una declaración de variables. También se utiliza para encadenar sentencias dentro de una sentencia for.
.punto	Para separar nombres de paquete de subpaquetes y clases. También se utiliza para separar una variable o método de una variable de referencia.

7. Sentencias o expresiones

Las expresiones realizan el trabajo de un programa JAVA. Entre otras cosas, las expresiones se utilizan para calcular y asignar valores a las variables y para controlar el flujo de un programa JAVA. El trabajo de una expresión se divide en dos partes: realizar los cálculos indicados por los elementos de la expresión y devolver algún valor. Una expresión es, por tanto, una serie de variables, operadores y llamadas a métodos (construida de acuerdo a la sintaxis del lenguaje) que evalúa a un valor sencillo.

Si no se le indica explícitamente al compilador el orden en el que se quiere que se realicen las operaciones, él decide basándose en la **precedencia** asignada a los operadores y otros elementos que se utilizan dentro de una expresión.

Una expresión es un conjunto variables unidos por operadores. Son órdenes que se le dan al equipo para que realice una tarea determinada. Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo: i = 0; j = 5; x = i + j; // Tres sentencias.

7.1. Bifurcaciones

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

A) Bifurcación if

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

```
if (expresión_booleana) {
    SENTENCIAS;
}
```

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

B) Bifurcación if else

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

```
if (expresión_booleana) {
    SENTENCIAS 1;
} else {
    SENTENCIAS 2;
}
```

C) Bifurcación if elseif else

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al else.

```
,
if (expresión_booleana) instrucción_si_true;
[else instrucción_si_false;]

if (expresión_booleana) {
    instrucciones_si_true;
}
else {
    instrucciones_si_false;
}
```

```
if (expresión_booleana1) {
    SENTENCIAS 1;
} else if (expresión_booleana2) {
    SENTENCIAS 2;

} else if (expresión_booleana3) {
    SENTENCIAS 3;
} else {
    SENTENCIAS 4;
}
```

D) Sentencia switch

Se trata de una alternativa a la bifurcación if else cuando se compara la misma expresión con distintos valores. Permite ejecutar una serie de operaciones para el caso de que una variable tenga un valor entero dado. La ejecución salta todos los case hasta que encuentra uno con el valor de la variable, y ejecuta desde allí hasta el final del case o hasta que encuentre un break, en cuyo caso salta al final del case. El default permite poner una serie de instrucciones que se ejecutan en caso de que la igualdad no se dé para ninguno de los case.

```
switch (expresión) {
      case (valor1): instrucciones_1;
      [break;]
      case (valor2): instrucciones_2;
      [break;]
      .....
      case (valorN): instrucciones_N;
      [break;]
      default: instrucciones_por_defecto;
}
```

```
switch (expression) {
   case value1: SENTENCIAS; break;
   case value2: SENTENCIAS; break;
   case value3: SENTENCIAS; break;
   case value4: SENTENCIAS; break;
   case value5: SENTENCIAS; break;
   case value6: SENTENCIAS; break;
   [default: statements7;]
}
```

Las características más relevantes de switch son las siguientes:

1. Cada sentencia case se corresponde con un único valor de expresión. No se pueden establecer rangos o condiciones, sino que se debe comparar con valores concretos.

2. Los valores no comprendidos en ninguna sentencia case se pueden gestionar en default, que es opcional.

3. En ausencia de break, cuando se ejecuta una sentencia case se ejecutan también todas las case que van a continuación, hasta que se llega a un break o hasta que se termina el switch.

7.2. Bucles

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

A) Bucle while

Permite ejecutar un grupo de instrucciones mientras se cumpla una condición dada:

```
while (expresión_booleana) {
    instrucciones...
}
```

Por ejemplo:

```
while ( linea != null) {
    linea = archivo.LeerLinea();
    System.out.println(linea);
}
```

B) Bucle for

La forma general del bucle for es la siguiente:

```
for (initialization; booleanExpression; increment) {
    SENTENCIAS;
}
```

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

```
initialization;
while (booleanExpression) {
    SENTENCIAS;
    incremento;
}
```

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

C) Bucle do while

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

tencias incluidas en el bucle, mientras que si la condición se evalúa a false finaliza el bucle. Este tipo de bucles se utiliza con frecuencia para controlar la satisfacción de una determinada condición de error o de convergencia.

```
do {
    instrucciones...
} while (expresión_booleana);
```

Por ejemplo:

```
do {
    linea = archivo.LeerLinea();
    if (linea != null) System.out.println(linea);
} while (linea != null);
```

7.3. Sentencias break, continue y return

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

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

A) Sentencias break y continue con etiquetas

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

```
lolo:
for ( int i = 0, j = 0; i < 100; i++){
    while ( true ) {
        if( (++j) > 5) { break lolo; }
        else { break; }
    }
}
```

B) Sentencia return

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

C) Sentencias de manejo de excepciones

Cuando ocurre un error dentro de un método JAVA, el método puede lanzar una excepción para indicar a su llamador que ha ocurrido un error y que el error está utilizando la sentencia **throw**. El método llamador puede utilizar las sentencias **try, catch**, y **finally** para capturar y manejar la excepción.

8. Arrays y cadenas

Al igual que otros lenguajes de programación, JAVA permite juntar y manejar múltiples valores a través de un objeto array (matriz). También se pueden manejar datos compuestos de múltiples caracteres utilizando el objeto String (cadena).

8.1. Arrays

Como otras variables, antes de poder utilizar un array primero se debe declarar. De nuevo, al igual que otras variables, la declaración de un array tiene dos componentes primarios: el tipo del array y su nombre. Un tipo de array incluye el tipo de dato de los elementos que va contener el array. Por ejemplo, el tipo de dato para un array que solo va a contener elementos enteros es un array de enteros. No puede existir un array de tipo de datos genérico en el que el tipo de sus elementos esté indefinido cuando se declara el array. Aquí tienes la declaración de un array de enteros: *int[] arrayDeEnteros*;

La parte **int**[] de la declaración indica que **arrayDeEnteros** es un array de enteros. La declaración no asigna ninguna memoria para contener los elementos del array. Si se intenta asignar un valor o acceder a cualquier elemento de **arrayDeEnteros** antes de haber asignado la memoria para él, el compilador dará un error.

Para asignar memoria a los elementos de un array, primero se debe ejemplarizar. Se puede hacer esto utilizando el operador **new** de JAVA (realmente, los pasos que se deben seguir para crear un array son similares a los se deben seguir para crear un objeto de una clase: declaración, ejemplarización e inicialización).

La siguiente sentencia asigna la suficiente memoria para que **arrayDeEnteros** pueda contener diez enteros.

int[] arraydeenteros = new int[10]

En general, cuando se crea un array, se utiliza el operador **new**, más el tipo de dato de los elementos del array, más el número de elementos deseados encerrado entre corchetes cuadrados ([y]).

TipodeElemento[ ] NombredeArray = new TipodeElementos[tamanoArray]

Los arrays pueden contener cualquier tipo de dato legal en JAVA incluyendo los tipos de referencia como son los objetos u otros array. Por ejemplo, el siguiente ejemplo declara un array que puede contener diez objetos String. *String[ ] arrayDeStrings = new String[10];*

Los elementos en este array son del tipo referencia, esto es, cada elemento contiene una referencia a un objeto String. En este punto, se ha asignado suficiente memoria para contener las referencias a los Strings, pero no se ha asignado memoria para los propios Strings. Si se intenta acceder a uno de los elementos de **arraydeStrings** obtendrá una excepción 'NullPointerException' porque el array está vacío y no contiene ni cadenas ni objetos String.

8.2. Strings

Una secuencia de datos del tipo carácter se llama un string (cadena) y en el entorno JAVA está implementada por la clase String (un miembro del paquete java.lang).

String[ ] args;

Este código declara explícitamente un array, llamado **args**, que contiene objetos del tipo String. Los corchetes vacíos indican que la longitud del array no se conoce en el momento de la compilación, porque el array se pasa en el momento de la ejecución.

El segundo uso de String es el uso de cadenas literales (una cadena de caracteres entre comillas "y"):

«Hola mundo!»

El compilador asigna implicitamente espacio para un objeto String cuando encuentra una cadena literal.

Los objetos String son inmutables —es decir, no se pueden modificar una vez que han sido creados—. El paquete java.lang proporciona una clase diferente, StringBuffer, que se podrá utilizar para crear y manipular caracteres. JAVA concatena cadenas fácilmente utilizando el operador **+**.

9. Clases en JAVA

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos. A estos datos y funciones pertenecientes a una clase se les denomina variables y métodos o funciones miembro. La programación orientada a objetos se basa en la programación de clases. Un programa se construye a partir de un conjunto de clases.

Una vez definida e implementada una clase, es posible declarar elementos de esta. Los elementos declarados se denominan objetos de la clase. De una clase se pueden declarar o crear numerosos objetos. La clase es lo genérico: es el patrón o modelo para crear objetos.

9.1. El cuerpo de la clase

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

La definición de una clase se realiza en la siguiente forma:

```
[elementos] class nombre_clase [elementos] {
    [lista_de_atributos]
    [lista_de_métodos]
}
```

El esqueleto de cualquier aplicación JAVA se basa en la definición de una clase. Todos los datos básicos, como los enteros, se deben declarar en las clases antes de hacer uso de ellos.

En la práctica son pocas las sentencias que se pueden colocar fuera del bloque de una clase. La palabra clave **import** (equivalente al **#include**) puede colocarse al principio de un fichero, fuera del bloque de la clase. Sin embargo, el compilador reemplazará esa sentencia con el contenido del fichero que se indique, que consistirá, como es de suponer, en más clases.

9.2. Tipos de clases

Los tipos de clases que podemos definir son:

- **abstract**. Una clase abstract tiene al menos un método abstracto. Una clase abstracta no se instancia, sino que se utiliza como clase base para la herencia.
- **final**. Una clase final se declara como la clase que termina una cadena de herencia. No se puede heredar de una clase final. Por ejemplo, la clase Math es una clase final.
- **public**. Las clases public son accesibles desde otras clases, bien sea directamente o por herencia. Son accesibles dentro del mismo paquete en el que se han declarado. Para acceder desde otros paquetes, primero tienen que ser importadas.
- **synchronizable**. Este modificador especifica que todos los métodos definidos en la clase son sincronizados, es decir, que no se puede acceder al mismo tiempo a ellos desde distintos threads; el sistema se encarga de colocar los flags necesarios para evitarlo. Este mecanismo hace que desde threads diferentes se puedan modificar las mismas variables sin que haya problemas de que se sobreescriban.

9.3. Características de las clases

1. Todas las variables y funciones de JAVA deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (extends) hereda todas sus variables y métodos.
3. JAVA tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.
4. Una clase solo puede heredar de una única clase (en JAVA no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de Object. La clase Object es la base de toda la jerarquía de clases de JAVA.
5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase public. Este fichero se debe llamar como la clase public que contiene con extensión *.java.
6. Si una clase contenida en un fichero no es public, no es necesario que el fichero se llame como la clase.
7. Los métodos de una clase pueden referirse de modo global al objeto de esa clase al que se aplican por medio de la referencia this.
8. Las clases se pueden agrupar en packages, introduciendo una línea al comienzo del fichero (package packageName;). Esta agrupación en packages está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

9.4. Declaración de la clase

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

[public] [final | abstract] class Clase [extends ClaseMadre] [implements Interfase1 [, Interfase2]...]

O bien, para interfaces:

[public] interface Interfase [extends InterfaseMadre1 [,InterfaseMadre2]...]

Como se ve, lo único obligatorio es class y el nombre de la clase. Las interfases son un caso de clase particular que veremos más adelante.

— **Extends.** La instrucción extends indica de qué clase desciende la nuestra. Si se omite, JAVA asume que desciende de la superclase Object. Cuando una clase desciende de otra, esto significa que hereda sus atributos y sus métodos (es decir que, a menos que los redefinamos, sus métodos son los mismos que los de la clase madre y pueden utili-

zarse en forma transparente, a menos que sean privados en la clase madre o, para subclases de otros paquetes, protegidos o propios del paquete).

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

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

En JAVA hay un montón de clases ya definidas y utilizables. Estas vienen en las bibliotecas estándar:

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

9.5. Variables miembro

Las variables declaradas en el cuerpo de la clase se dice que son miembros de la clase y son accesibles por todos los métodos de la clase; las variables miembro en la programación JAVA, son los datos o atributos que definen los objetos; cada objeto, es decir, cada ejemplar concreto de la clase, tiene su propia copia de las variables miembro.

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

Las variables miembro pueden también inicializarse explícitamente en la declaración, como las variables locales, por medio de constantes o llamadas a métodos. Por ejemplo:

```
long nDatos = 100;
```

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

Cada objeto que se crea de una clase tiene su propia copia de las variables miembro. Por ejemplo, cada objeto de la clase Figura tiene sus propias coordenadas del centro "x" e "y", y su propio valor del radio "r".

Los métodos de objeto se aplican a un objeto concreto poniendo el nombre del objeto y luego el nombre del método, separados por un punto. A este objeto se le llama argumento implícito. Por ejemplo, para calcular el área de un objeto de la clase Figura llamado obj1 se escribirá: obj1.area();. Las variables miembro del argumento implícito se acceden directamente o precedidas por la palabra this y el operador punto.

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

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

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

9.6. Variables miembro de clase (static)

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama variables de clase o variables static. Las variables static se suelen utilizar para definir constantes comunes para todos los objetos de la clase o variables que solo tienen sentido para toda la clase. Las variables de clase son lo más parecido que JAVA tiene a las variables globales de C/C++.

Las variables de clase se crean anteponiendo la palabra static a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Si no se les da valor en la declaración, las variables miembro static se inicializan con los valores por defecto para los tipos primitivos (false para Boolean, el carácter nulo para char y cero para los tipos numéricos), y con null si es una referencia.

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

9.7. Variables Finales

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

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

Además de las variables miembro, también las variables locales y los propios argumentos de un método pueden ser declarados final. Declarar como final un objeto miembro de una clase hace constante la referencia, pero no el propio objeto, que puede ser modificado a través de otra referencia. En JAVA no es posible hacer que un objeto sea constante.

9.8. Métodos o funciones miembros

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

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

```
public Figura elMayor(Figura c) {
   if (this.r>=c.r)  return this;
   else  return c;
}
```

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

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

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

9.9. Paso de argumentos a métodos

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

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

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

Si un método devuelve this (es decir, un objeto de la clase) o una referencia a otro objeto, ese objeto puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así sucesivamente. En este caso aparecerán varios métodos en la misma sentencia unidos por el operador punto (.), por ejemplo:

String numeroComoString =”8.978”;

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

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

String numeroComoString = ”8.978”;

float f = float.valueOf(numeroComoString);

float p = f.floatValue();

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

9.10. Métodos de clase (static)

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

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

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

9.11. Llamadas a métodos

```
public final class MiClase extends Number {
  // atributos:
  private float x;
  private float y;
```

```
    // constructor:
    public MiClase(float rx, float iy) {x = rx;y = iy;}
    // métodos:
    public float Norma() {return (float)Math.sqrt(x*x+y*y);}
    public double doubleValue() {return (double)Norma( );}
    public float floatValue() {return Norma();}
    public int intValue() {return (int)Norma();}
    public long longValue() {return (long)Norma();}
    public String toString() {return «(«+x+»)+i(«+y+»)»;}
}
```

Nombre_del_Objeto< .>Nombre_del_Método(parámetros)

```
import java.io.*;
public class Lolo {
    public static void main(String args[]) {
        MiClase obj = new MiClase(4,-3);
        System.out.println(obj.toString());
        System.out.println(obj.Norma());
    }
}
```

En la clase MiClase tenemos también un ejemplo de un llamado a un método de clase, o sea static:

*return (float)Math.sqrt(x*x+y*y);*

Como el método es de clase, no hace falta llamarlo para un objeto en particular. En ese caso, en lugar del nombre de un objeto existente se puede utilizar directamente el nombre de la clase:

Nombre_de_la_Clase<punto>Nombre_del_Método(parámetros)

9.12. Constructores en JAVA

Un punto clave de la programación orientada objetos es el evitar información incorrecta por no haber sido correctamente inicializadas las variables. JAVA no permite que haya variables miembro que no estén inicializadas. JAVA inicializa siempre con valores por defecto las variables miembro de clases y objetos. El segundo paso en la inicialización correcta de objetos es el uso de constructores. Un constructor es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del constructor es reservar memoria e inicializar las variables miembro de la clase.

Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase. Su argumento implícito es el objeto que se está creando. De ordinario una clase tiene varios constructores, que se diferencian por el tipo y número de sus argumentos. Se llama constructor por defecto al constructor que no tiene argumentos. El programador debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un constructor de una clase puede llamar a otro constructor previamente definido en la misma clase por medio de la palabra this. En este contexto, la palabra this solo puede aparecer en la primera sentencia de un constructor. El constructor de una sub-clase puede llamar al constructor de su súper-clase por medio de la palabra súper, seguida de los argumentos apropiados entre paréntesis. De esta forma, un constructor solo tiene que inicializar por sí mismo las variables no heredadas. El constructor es tan importante que, si el programador no prepara ningún constructor para una clase, el compilador crea un constructor por defecto, inicializando las variables de los tipos primitivos a su valor por defecto, y los Strings y las demás referencias a objetos a null. Si hace falta, se llama al constructor de la súper-clase para que inicialice las variables heredadas. Al igual que los demás métodos de una clase, los constructores pueden tener también los modificadores de acceso public, private, protected y package. Si un constructor es private, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos public y static (factory methods) que llamen al constructor y devuelvan un objeto de esa clase. Dentro de una clase, los constructores solo pueden ser llamados por otros constructores o por métodos static. No pueden ser llamados por los métodos de objeto de la clase.

9.13. Destrucción de objetos

En JAVA no hay destructores como en C++. El sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han perdido la referencia, esto es, objetos que ya no tienen ningún nombre que permita acceder a ellos, por ejemplo por haber llegado al final del bloque en el que habían sido definidos, porque a la referencia se le ha asignado el valor null o porque a la referencia se le ha asignado la dirección de otro objeto. A esta característica de JAVA se le llama garbage collection (recogida de basura). En JAVA es normal que varias variables de tipo referencia apunten al mismo objeto. JAVA lleva internamente un contador de cuantas referencias hay sobre cada objeto. El objeto podrá ser borrado cuando el número de referencias sea cero. Como ya se ha dicho, una forma de hacer que un objeto quede sin referencia es cambiar esta a null, haciendo por ejemplo: *ObjetoRef = null;*

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

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

```
public class Contador { // Se define la clase Contador
   // Atributos
   int cnt;
   // Constructor (un metodo igual que otro cualquiera)
   public Contador() {
      cnt = 0;
   }
   // Métodos
   public int incCuenta() {
      cnt++;
      return cnt;
   }
   public int getCuenta() {
      return cnt;
   }
}
```

Cuando, desde una aplicación u otro objeto, se crea una **instancia** de la clase **Contador** mediante la instrucción **new Contador()** el compilador busca un método con el mismo nombre de la clase y que se corresponda con la llamada en cuanto al tipo y número de parámetros. Dicho método se llama Constructor, y una clase puede tener más de un constructor (no así un objeto o instancia, ya que una vez que fue creado no puede recrearse sobre sí mismo). En tiempo de ejecución, al encontrar dicha instrucción, el intérprete reserva espacio para el objeto/instancia, crea su estructura y llama al constructor. O sea que el efecto de new Contador() es, precisamente, reservar espacio para el contador e inicializarlo en cero.

En cuanto a los otros métodos, se pueden llamar desde otros objetos (lo que incluye a las aplicaciones) del mismo modo que se llama una función desde C. Por ejemplo, usemos nuestro contador en un programa bien sencillo que nos muestre cómo evoluciona:

```
import java.io.*;
public class Lolo1 {
   static int n;
   static Contador obj;
   public static void main ( String args[] ) {
      System.out.println («Cuenta... «);
      obj = new Contador();
      System.out.println
      (obj.getCuenta());
      n = obj.incCuenta();
      System.out.println (n);
      obj.incCuenta();
      System.out.println
      (obj.getCuenta());
      System.out.println
      (obj.incCuenta());
   }
}
```

```
import java.applet.*;    import java.awt.*;
public class Lolo2 extends Applet {
   static int n;
   static Contador obj;
   public Lolo2 () {
      obj = new Contador();
   }
  public void paint (Graphics g) {
        g.drawString («Cuenta...», 20, 20);
g.drawString (String.valueOf(obj.getCuenta()), 20, 35 );
        n = obj.incCuenta();
g.drawString (String.valueOf(n), 20, 50 );
        obj.incCuenta();
g.drawString (String.valueOf(obj.getCuenta()), 20, 65 );
g.drawString (String.valueOf(obj.incCuenta()), 20, 80 );
   }
}
```

Ahora es necesario crear una página HTML para poder visualizarlo. Para esto, crear y luego cargar el archivo Lolo2.htm con un browser que soporte JAVA (o bien ejecutar en la ventana DOS: «appletviewerLolo2.htm»):

```
<HTML>
   <HEAD>
      <TITLE>Lolo 2 - Applet Contador</TITLE>
   </HEAD>
   <BODY>
      <applet code=»Lolo2.class» width=170 height=150> </applet>
</BODY>
</HTML>
```

Observemos las diferencias entre la aplicación standalone y el applet:

— La aplicación usa un método main, desde donde arranca.

— El applet, en cambio, se arranca desde un constructor (método con el mismo nombre que la clase).

— En la aplicación utilizamos System.out.println para imprimir en la salida estándar.

— En el applet necesitamos «dibujar» el texto sobre un fondo gráfico, por lo que usamos el método g.drawString dentro del método paint

(que es llamado cada vez que es necesario redibujar el applet). Con poco trabajo se pueden combinar ambos casos en un solo objeto, de modo que **la misma** clase sirva para utilizarla de las dos maneras:

```
import java.applet.*;
import java.awt.*;
import java.io.*;
public class Lolo3 extends Applet {
    static int n;
    static Contador obj;
    public Lolo3 () {
        obj = new Contador();
    }
    public static void main(String args[]) {
        obj = new Contador();
        paint();
    }
    public static void paint () {
        System.out.println («Cuenta...»);
        System.out.println (obj.getCuenta());
        n = obj.incCuenta();
        System.out.println (n);
        obj.incCuenta();
        System.out.println (obj.getCuenta());
        System.out.println (obj.incCuenta());
    }
    public void paint (Graphics g) {
        g.drawString («Cuenta...», 20, 20);
        g.drawString (String.valueOf(obj.getCuenta()), 20, 35 );
        n = obj.incCuenta();
        g.drawString (String.valueOf(n), 20, 50 );
        obj.incCuenta();
        g.drawString (String.valueOf(obj.getCuenta()), 20, 65 );
        g.drawString (String.valueOf(obj.incCuenta()), 20, 80 );
    }
}
```

Esta clase puede ejecutarse tanto con «java Lolo3» en una ventana de MS-DOS, como cargarse desde una página HTML con:

<applet code=»Lolo3.class» width=170 height=150> </applet>

Notar que conviene probar el applet con el appletviewer («appletviewer Lolo3.htm»), ya que este indica en la ventana DOS si hay algún error durante la ejecución. Los browsers dejan pasar muchos errores, simplemente suprimiendo la salida a pantalla del código erróneo.

Fíjate que en todo este desarrollo de las clases Lolo1, Lolo2 y Lolo3, en ningún momento volvimos a tocar la clase Contador.

Los métodos, como las clases, tienen una declaración y un cuerpo. La declaración es del tipo:

[private |protected|public] [static] [abstract] [final] [native] [synchronized] *TipoDevuelto NombreMétodo ([tipo1 nombre1[, tipo2 nombre2]...]) [**throws** excepción1 [,excepción2]...]*

Básicamente, los métodos son como las funciones de C: implementan, a través de funciones, operaciones y estructuras de control, el cálculo de algún parámetro que es el que devuelven al objeto que los llama. Solo pueden devolver **un** valor (del tipo TipoDevuelto), aunque pueden no devolver ninguno (en ese caso TipoDevuelto es **void**). Como ya veremos, el valor de retorno se especifica con la instrucción **return**, dentro del método.

Los métodos pueden utilizar valores que les pasa el objeto que los llama (*parámetros*), indicados con *tipo1 nombre1, tipo2 nombre2...* en el esquema de la declaración.

Estos parámetros pueden ser de cualquiera de los tipos ya vistos. Si son tipos básicos, el método recibe el valor del parámetro; si son arrays, clases o interfases, recibe un puntero a los datos (*referencia*). Veamos un pequeño ejemplo:

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

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

— El método recibe un valor entero (*cantidad*),

— lo suma a la variable de instancia *cnt*,

— devuelve la suma (*return cnt*).

Supongamos que queremos hacer un método dentro de una clase que devuelva la posición del ratón (mouse).

Lo siguiente no sirve:

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

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

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

Luego utilizar esa clase en nuestro método:

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

9.14. La clase MiClase

```
public final class MiClase extends Number {
      private float x;
      private float y;

      public MiClase() {x = 0;y = 0;}
      public MiClase(float rx, float iy) {x = rx;y = iy;}

      public final float Norma() {return (float)Math.sqrt(x*x+y*y);}
      public final float Norma(MiClase c) {return (float)Math.sqrt(c.x*c.x+c.y*c.y);}
      public final MiClase Conjugado() {MiClase r = new MiClase(x,-y);return r;}
      public final MiClase Conjugado(MiClase c) {MiClase r = new MiClase(c.x,-
      c.y);return r;}
      public final double doubleValue() {return (double)Norma();}
      public final float floatValue() {return Norma();}
      public final int intValue() {return (int)Norma();}
      public final long longValue() {return (long)Norma();}
      public final String toString() {
```

```
    f (y<0)          return x+»-i»+(-y);
    else             return x+»+i»+y;   }
  public static final MiClase Suma(MiClase obj1, MiClase c2) {
    return new MiClase(obj1.x+c2.x,obj1.y+c2.y);          }
  public static final MiClase Resta(MiClase obj1, MiClase c2) {
    return new MiClase(obj1.x-c2.x,obj1.y-c2.y); }
  public static final MiClase Producto(MiClase obj1, MiClase c2) {
    return new MiClase(obj1.x*c2.x-obj1.y*c2.y,obj1.x*c2.y+obj1.y*c2.x);}
  public static final MiClase DivEscalar(MiClase c, float f) {return new MiCla-
  se(c.x/ f,c.y/f);}
  public static final MiClase Cociente(MiClase obj1, MiClase c2) {
    float x = obj1.x*c2.x+obj1.y*c2.y;        float y = -obj1.x*c2.y+obj1.y*c2.x;
    float n = c2.x*c2.x+c2.y*c2.y;  MiClase r = new MiClase(x,y);      return
    DivEscalar(r,n);}
}
```

Podemos hacer algunos comentarios:

1. No hay include aquí, ya que la única biblioteca que usamos es **java.lang** y se incluye automáticamente.

2. La clase es public final, lo que implica que cualquier clase en este u otros paquetes puede utilizarla, pero ninguna clase puede heredarla (o sea que es una clase estéril...).

Hagamos un resumen de los atributos y métodos de la clase:

```
// atributos:
private float x;
private float y;
```

Siendo privados, no podemos acceder a ellos desde el exterior. Como además la clase es final, no hay forma de acceder a "x" e "y". Además, al no ser static, cada instancia de la clase tendrá su propio "x" e "y".

```
// constructores:
public MiClase()
public MiClase(float rx, float iy)
```

La clase tiene dos constructores, que se diferencian por su «firma» (*signature*), o sea por la cantidad y tipo de parámetros. El primero nos sirve para crear un objeto de tipo **MiClase** y valor indefinido (aunque en realidad el método lo inicializa en cero); con el segundo, podemos definir el valor al crearlo.

```
// métodos:
public final float Norma()
public final float Norma(MiClase c)
public final MiClase Conjugado()
public final MiClase Conjugado(MiClase c)
```

Estos métodos también son duales; cuando los usamos sin parámetros devuelven la norma o el conjugado del objeto individual (instancia):

```
v = miMiClase.Norma(); // por ejemplo
otroMiClase = miMiClase.Conjugado();
Con parámetros, en cambio, devuelven la norma o el conjugado del parámetro:
v = unMiClase.Norma(miMiClase);
otroMiClase = unMiClase.Conjugado(miMiClase);
Notar que lo siguiente es inválido:
otroMiClase = MiClase.Norma(miMiClase); // NO SE PUEDE!
```

Porque el método no es static, por lo tanto **debe** llamarse para una instancia en particular (en este caso, *unMiClase*).

```
// obligatorios (pues son abstractos en Number):
public final double doubleValue()
public final float floatValue()
public final int intValue()
public final long longValue()
```

Estos métodos es obligatorio definirlos, ya que en la clase madre **Number** son métodos abstractos, o sea que debemos implementarlos aquí. Como todos los métodos de esta clase son final, o sea que no puede ser redefinido. No es importante en realidad puesto que la clase no puede tener descendientes.

```
public final String toString()
```

Este método nos sirve para representar el MiClase como una cadena de caracteres, de la forma x+iy.

```
// Operaciones matemáticas
public static final MiClase Suma(MiClase obj1, MiClase c2)
public static final MiClase Resta(MiClase obj1, MiClase c2)
public static final MiClase Producto(MiClase obj1, MiClase c2)
public static final MiClase DivEscalar(MiClase c, float f)
public static final MiClase Cociente(MiClase obj1, MiClase c2)
```

Aquí definimos varias operaciones matemáticas. Notar que se han definido como static, o sea que los métodos son únicos independientemente de las instancias. Esto permite que los podamos ejecutar sobre una instancia o directamente sobre la clase:

```
miMiClase = unMiClase.Suma(comp1,comp2); // vale
miMiClase = MiClase.Suma(comp1,comp2); // tambien seria correcto
```

Por ejemplo, la siguiente aplicación nos muestra cómo podemos usar algunos de estos métodos:

```
import java.io.*;

public class Lolo5 {

    public static void main(String args[]) {

        MiClase obj1 = new MiClase(4,-3);

        System.out.println(obj1+»\tNorma=»+obj1.Norma());

        MiClase c2 = new MiClase(-2,5);

        System.out.println(c2+»\tNorma=»+c2.Norma()+»\n»);

        System.out.println(«(«+obj1+»)/4 :»+MiClase.DivEscalar(obj1,4));

        System.out.println(«Suma : «+MiClase.Suma(obj1,c2));

        System.out.println(«Resta : «+MiClase.Resta(obj1,c2).toString());

        System.out.println(«Multip: «+MiClase.Producto(obj1,c2).toString());

        System.out.println(«Divis : «+MiClase.Cociente(obj1,c2).toString());

    }

}
```

MiClase obj1 = new MiClase(4,-3); obj1 es un objetos (instancias) de la clase MiClase.

El operador new ejemplariza una clase mediante la asignación de memoria para el objeto nuevo de ese tipo. new necesita un solo argumento: una llamada al método constructor. Los métodos constructores son métodos especiales proporcionados por cada clase JAVA que son reponsables de la inicialización de los nuevos objetos de ese tipo. El operador new crea el objeto, el constructor lo inicializa.

new Rectangle(0, 0, 100, 200);

En el ejemplo, Rectangle(0, 0, 100, 200) es una llamada al constructor de la clase Rectangle. El operador new devuelve una referencia al objeto recién creado. Esta referencia puede ser asignada a una variable del tipo apropiado.

Rectangle rect = new Rectangle(0, 0, 100, 200);

Recuerde que una clase escencialmente define un tipo de dato de referencia. Por eso, Rectangle puede utilizarse como un tipo de dato en los programas JAVA. El valor de cualquier variable cuyo tipo sea un tipo de referencia, es una referencia —un puntero— al valor real o conjunto de valores representado por la variable.

Como se dijo anteriormente, las clases porporcionan métodos constructores para incializar los nuevos objetos de ese tipo. Una clase podría proporcionar múltiples constructores para realizar diferentes tipos de inicialización en los nuevos objetos. Cuando vea la implementación de una clase, reconocerá los constructores porque tienen el mismo nombre que la clase y no tienen tipo de retorno. Recuerde la creacción del objeto Date en el sección inicial. El constructor utilizado no tenía ningún argumento: Date().

Un constructor que no tiene ningún argumento, como el mostrado arriba, es conocido como **constructor por defecto**. Al igual que Date, la mayoría de las clases tienen al menos un constructor, el constructor por defecto.

Si una clase tiene varios constructores, todos ellos tienen el mismo nombre pero se deben diferenciar en el número o el tipo de sus argmentos. Cada constructor inicializa el nuevo objeto de una forma diferente. Junto al constructor por defecto, la clase Date proporciona otro constructor que inicializa el nuevo objeto con un nuevo año, mes y día: *Date cumpleaños = new Date(1963, 8, 30)*;

El compilador puede diferenciar los constructores a través del tipo y del número de sus argumentos.

Para acceder a las variables de un objeto, solo se tiene que añadir el nombre de la variable al del objeto referenciado introduciendo un punto en el medio ('.').

objetoReferenciado.variable

Recuerde que el operador **new** devuelve una referencia a un objeto. Por eso, se puede utilizar el valor devuelto por **new** para acceder a las variables del nuevo objeto:

height = new Rectangle().height;

Llamar a un método de un objeto es similar a obtener una variable del objeto. Para llamar a un método del objeto, simplemente se añade al nombre del objeto referenciado el nombre del método, separados por un punto ('.'), y se proporcionan los argumentos del método entre paréntesis. Si el método no necesita argumentos, se utilizan los paréntesis vacios.

objetoReferenciado.nombreMétodo(listaArgumentos); o

objetoReferenciado.nombreMétodo();

Recuerde que una llamada a un método es un mensaje al objeto nombrado. El objeto referenciado en la llamada al método objetoReferenciado.

método() debe ser una referencia a un objeto. Como se puede utilizar un nombre de variable aquí, también se puede utilizar en cualquier expresión que devuelva una referencia a un objeto. Recuerda que el operador new devuelve una referencia a un objeto. Por eso, se puede utilizar el valor devuelto por new para acceder a las variables del nuevo objeto:

new Rectangle(0, 0, 100, 50).equals(anotherRect)

La expresión ***new Rectangle(0, 0, 100, 50)*** evalúa a una referencia a un objeto que se refiere a un objeto Rectangle. Entonces, como verás, se puede utilizar la notación, de punto ('.') para llamar al método **equals()** del nuevo objeto Rectangle para determinar si el rectangúlo nuevo es igual al especificado en la lista de argumentos de **equals()**.

9.15. Eliminar objetos no utilizados

Muchos otros lenguajes orientados a objetos necesitan que se siga la pista de los objetos que se han creado y luego se destruyan cuando no se necesiten. Escribir código para manejar la memoria de esta es forma es aburrido y propenso a errores. JAVA permite ahorrarse esto, permitiendo crear tantos objetos como se quiera (solo limitados por los que el sistema pueda manejar) pero nunca tienen que ser destruidos. El entorno de ejecución JAVA borra los objetos cuando determina que no se van a utilizar más. Este proceso es conocido como recolección de basura.

Un objeto es elegible para la recolección de basura cuando no existen más referencias a ese objeto. Las referencias que se mantienen en una variable desaparecen de forma natural cuando la variable sale de su ámbito. O cuando se borra explícitamente un objeto referencia mediante la selección de un valor cuyo tipo de dato es una referencia a null.

9.16. Recolector de Basura

El entorno de ejecución de JAVA tiene un recolector de basura que periódicamente libera la memoria ocupada por los objetos que no se van a necesitar más. El recolector de basura de JAVA es un **barredor de marcas** que escanea dinámicamente la memoria de JAVA buscando objetos, marcando aquellos que han sido referenciados. Después de investigar todos los posibles paths de los objetos, los que no están marcados (esto es, no han sido referenciados) se les conoce como basura y son eliminados.

El colector de basura funciona en un thread (hilo) de baja prioridad y funciona tanto síncrona como asíncronamente dependiendo de la situación y del sistema en el que se esté ejecutando el entorno JAVA. El recolector de basura se ejecuta síncronamente cuando el sistema funciona fuera de memoria o en respuesta a una petición de un programa JAVA. Un programa JAVA le puede pedir al recolector de basura que se ejecute en cualquier momento mediante una llamada a *System.gc()*.

Nota: *que se ejecute el recolector de basura no garantiza que los objetos sean recolectados.*

En sistemas que permiten que el entorno de ejecución JAVA note cuando un thread a empezado a interrumpir a otro thread (como Windows 95/NT), el recolector de basura de JAVA funciona asíncromamente cuando el sistema está ocupado. Tan pronto como otro thread se vuelva activo, se pedirá al recolector de basura que obtenga un estado consistente y termine.

9.17. Finalización

Antes de que un objeto sea recolectado, el recolector de basura le da una oportunidad para limpiarse él mismo mediante la llamada al método **finalize**() del propio objeto. Este proceso es conocido como finalización. Durante la finalización un objeto se podrían liberar los recursos del sistema como son los ficheros, etc., y liberar referencias en otros objetos para hacerse elegible por la recolección de basura. El método finalize() es un miembro de la clase java.lang.Object. Una clase debe sobreescribir el método finalize() para realizar cualquier finalización necesaria para los objetos de ese tipo.

9.18. This

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a las variables miembros del objeto. Sin embargo, algunas veces no se querrá tener ambigüedad sobre el nombre de la variable miembro y uno de los argumentos del método que tengan el mismo nombre. Por ejemplo, el siguiente constructor de la clase HSBColor inicializa alguna variable miembro de un objeto de acuerdo a los argumentos pasados al constructor. Cada argumento del constructor tiene el mismo nombre que la variable del objeto cuyo valor contiene el argumento.

```
class HSBColor {
    int luminosidad, saturacion, brillo;
    HSBColor (int luminosidad, int saturacion, int brillo) {
        this.luminosidad = luminosidad;
        this.saturacion = saturacion;
        this.brillo = brillo;
    }
```

Se debe utilizar **this** en este constructor para evitar la ambigüedad entre el argumento **luminosidad** y la variable miembro **luminosidad** (y así con el resto de los argumentos). Escribir **luminosidad = luminosidad** no tendría sentido. Los nombres de argumentos tienen mayor precedencia y ocultan a los nombres de las variables miembro con el mismo nombre. Para referirise a la variable miembro se debe hacer explicitamente a través del objeto actual—**this**. También se puede utilizar **this** para llamar a uno de los métodos del objeto actual. Esto solo es necesario si existe alguna ambigüedad con el nombre del método y se utiliza para intentar hacer el código más claro.

9.19. Super

Si el método oculta una de las variables miembro de la superclase, se puede referir a la variable oculta utilizando **super**. De igual forma, si el método sobreescribe uno de los métodos de la superclase, se puede llamar al método sobreescrito a través de **super**.

Consideremos esta clase:

```
class MiClase {
        boolean unaVariable;
        void unMetodo() {
                unaVariable = true;
        }
}
```

y una subclase que oculta **unaVariable** y sobreescribe **unMetodo()**:

```
class OtraClase extends MiClase {
        boolean unaVariable;
        void unMetodo() {
                unaVariable = false;
                super.unMetodo();
                System.out.println(unaVariable);
                System.out.println(super.unaVariable);
        }
}
```

Primero **unMetodo()** selecciona **unaVariable** (una declarada en OtraClase que oculta a la declarada en MiClase) a **false**. Luego **unMetodo()** llama a su método sobreescrito con esta sentencia:

```
super.unMetodo();
```

Esto selecciona la versión oculta de **unaVariable** (la declarada en MiClase) a **true**. Luego **unMetodo** muestra las dos versiones de **unaVariable** con diferentes valores: false/true.

9.20. Miembros de la clase y del ejemplar

Las variables de ejemplar están en contraste con las *variables de clase* (que se declaran utilizando el modificador **static**). El sistema asigna espacio para las variables de clase una vez por clase, sin importar el número de ejemplares creados. Todos los objetos creados de esta clase comparten la misma copia de las variables de clase de la clase, se puede acceder a estas variables a través de un ejem-

plar o través de la propia clase. Los métodos son similares: una clase puede tener métodos de ejemplar y métodos de clase. Los métodos de ejemplar operan sobre las variables de ejemplar del objeto actual pero también pueden acceder a las variables de clase. Por otro lado, los métodos de clase no pueden acceder a las variables del ejemplar declarados dentro de la clase (a menos que se cree un objeto nuevo y acceda a ellos através del objeto). Los métodos de clase también pueden ser invocados desde la clase, no se necesita un ejemplar para llamar a los métodos de la misma.

Para especificar que una variable miembro es una variable de clase, se utiliza la palabra clave **static**. Por ejemplo, cambiemos la clase UnEnteroLlamadoX para que su variable **x** sea ahora una variable de clase:

```
class UnEnteroLlamadoX {
    static int x;
    public int x() {    return x     }
    public void setX(int newX) {  x = newX;     }
}
```

Ahora veamos el mismo código mostrado anteriormente que crea dos ejemplares de UnEnteroLlamadoX, selecciona sus valores de x, y muestra esta salida diferente: miX.x = 2 otroX.x = 2

La salida es diferente porque **x** ahora es una variable de clase por lo que solo hay una copia de la variable y es compartida por todos los ejemplares de UnEnteroLlamadoX incluyendo **miX** y **otroX**.

Cuando se llama a **setX()** en cualquier ejemplar, cambia el valor de **x** para todos los ejemplares de UnEnteroLlamadoX. Las variables de clase se utilizan para aquellos puntos en lo que se necesite una sola copia que debe estar accesible para todos los objetos heredados por la clase en la que la variable fue declarada. Por ejemplo, las variables de clase se utilizan frecuentemente con **final** para definir constantes (esto es más eficiente en el consumo de memoria, ya que las constantes no pueden cambiar y solo se necesita una copia).

Similarmente, cuando se declare un método, se puede especificar que es un método de clase en vez de un método de ejemplar. Los métodos de clase solo pueden operar con variables de clase y no pueden acceder a las variables de ejemplar definidas en la clase.

Para especificar que un método es de clase, se utiliza la palabra clave **static** en la declaración de método. Cambiemos la clase UnEnteroLlamadoX para que su variable miembro x sea de nuevo una variable de ejemplar y sus dos métodos sean ahora métodos de clase:

```
class UnEnteroLlamadoX {
    private int x;
    static public int x() {
        return x;
```

```
    }
    static public void setX(int newX) {
        x = newX;
    }
}
```

Cuando se intente compilar esta versión de UnEnteroLlamadoX, se obtendrán errores de compilación:

UnEnteroLlamadoX.java:4: Can't make a static reference to nonstatic variable x in class UnEnteroLlamadoX.

```
    return x;
    ^
```

UnEnteroLlamadoX.java:7: Can't make a static reference to nonstatic variable x in class UnEnteroLlamadoX.

```
    x = newX;
    ^
```

2 errors

Esto es porque los métodos de clase no pueden acceder a variables de ejemplar a menos que el método haya creado un ejemplar de UnEnteroLlamadoX primero y luego acceda a la variable a través de él. Construyamos de nuevo UnEnteroLlamadoX para hacer que su variable **x** sea una variable de clase:

```
class UnEnteroLlamadoX {
    static private int x;
    static public int x() {   return x;    }
    static public void setX(int newX) {     x = newX;   }
}
```

Ahora la clase se compilará y el código anterior que crea dos ejemplares de UnEnteroLlamadoX, selecciona sus valores **x**, y muestra en su salida los valores de **x**: miX.x = 2 otroX.x = 2

De nuevo, cambiar **x** a través de **miX** también lo cambia para los otros ejemplares de UnEnteroLlamadoX.

Otra diferencia entre miembros del ejemplar y de la clase es que los miembros de la clase son accesibles desde la propia clase. No se necesita ejemplarizar la clase para acceder a los miembros de clase. Reescribamos el código anterior para acceder a **x()** y **setX()** directamente desde la clase UnEnteroLlamadoX:

```
. . .
UnEnteroLlamadoX.setX(1);
System.out.println(«UnEnteroLlamadoX.x = « + UnEnteroLlamadoX.x());
. . .
```

Observe que ya no se tendrá que crear **miX** u **otroX**. Se puede seleccionar **x** y recuperarlo directamente desde la clase UnEnteroLlamadoX. No se puede hacer esto con miembros del ejemplar. Solo se puede invocar métodos de ejemplar a través de un objeto y solo puede acceder a las variables de ejemplar desde un objeto. Se puede acceder a las variables y métodos de clase desde un ejemplar de la clase o desde la clase misma.

9.21. Permisos de Acceso

Cuando se crea una nueva clase en JAVA, se puede especificar el nivel de acceso que se quiere para las variables de instancia y los métodos definidos en la misma. Ciertas informaciones y peticiones contenidas en la clase, las soportadas por los métodos y variables accesibles públicamente en su objeto son correctas para el consumo de cualquier otro objeto del sistema. Otras peticiones contenidas en la clase son solo para el uso personal de la clase. Estas otras soportadas por la operación de la clase no deberían ser utilizadas por objetos de otros tipos. Se querría proteger esas variables y métodos personales a nivel del lenguaje y prohibir el acceso desde objetos de otros tipos.

En JAVA se puede utilizar los especificadores de acceso para proteger tanto las variables como los métodos de la clase cuando se declaran. El lenguaje JAVA soporta cuatro niveles de acceso para las variables y métodos miembros: private, protected, public, y, todavía no especificado, acceso de paquete.

— **Public.** Cualquier clase desde cualquier lugar puede acceder a las variables y métodos de instacia públicos.

```
public void CualquieraPuedeAcceder(){}
```

— **Protected**. Solo las subclases de la clase y nadie más puede acceder a las variables y métodos de instancia protegidos.

```
protected void SoloSubClases(){}
```

— **Private.** Las variables y métodos de instancia privados solo pueden ser accedidos desde dentro de la clase. No son accesibles desde las subclases.

```
private String NumeroDelCarnetDeIdentidad;
```

— **Friendly** (sin declaración específica). Por defecto, si no se especifica el control de acceso, las variables y métodos de instancia se declaran friendly, lo que significa que son accesibles por todos los objetos dentro del mismo paquete, pero no por los externos al paquete. Es lo mismo que protected.

void MetodoDeMiPaquete(){}

Los métodos protegidos (protected) pueden ser vistos por las clases derivadas, como en C++, y también en JAVA, por los paquetes (packages). Todas las clases de un paquete pueden ver los métodos protegidos de ese paquete. Para evitarlo, se deben declarar como **private protected**, lo que hace que ya funcione como en C++ en donde solo se puede acceder a las variables y métodos protegidos de las clases derivadas. La siguiente tabla le mues-

ESPECIFICADOR	CLASE	SUBCLASE	PAQUETE	MUNDO
private	X			
protected	X	X*	X	
public	X	X	X	X
package	X		X	

tra los niveles de acceso pemitidos por cada especificador:

La primera columna indica si la propia clase tiene acceso al miembro definido por el especificador de acceso. La segunda columna indica si las subclases de la clase (sin importar dentro de que paquete se encuentren estas) tienen acceso a los miembros. La tercera columna indica si las clases del mismo paquete que la clase (sin importar su parentesco) tienen acceso a los miembros. La cuarta columna indica si todas las clases tienen acceso a los miembros.

Observa que la intersección entre protected y subclase tiene un '*', este caso de acceso particular tiene una explicación en más detalle más adelante.

A) Private

El nivel de acceso más restringido es private. Un miembro privado es accesible solo para la clase en la que está definido. Se utiliza este acceso para declarar miembros que solo deben ser utilizados por la clase. Esto incluye las variables que contienen información que si se accede a ella desde el exterior podría colocar al objeto en un estado de inconsistencia, o los métodos que llamados desde el exterior pueden poner en peligro el estado del objeto o del programa donde se está ejecutando. Los miembros privados son como secretos, nunca deben contársele a nadie.

B) Protected

El siguiente especificador de nivel de acceso es protected que permite a la propia clase, las subclases (con la excepción a la que nos referimos anteriormente), y todas las clases dentro del mismo paquete que accedan a los miembros. Este nivel de acceso se utiliza cuando es apropiado para una subclase da la clase tener acceso a los miembros, pero no las clases no relacionadas. Los miembros protegidos son como secretos familiares –no importa que toda la familia lo sepa, incluso algunos amigos allegados pero no se quiere que los extraños lo sepan.

C) Public

El especificador de acceso más sencillo es public. Todas las clases, en todos los paquetes tienen acceso a los miembros públicos de la clase. Los miembros públicos se declaran solo si su acceso no produce resultados indeseados si un extraño los utiliza. Aquí no importa que lo sepa todo el mundo.

D) Acceso de paquete (friendly)

Y finalmente, el último nivel de acceso es el que se obtiene si no se especifica ningún otro nivel de acceso a los miembros. Este nivel de acceso permite que las clases del mismo paquete que la clase tengan acceso a los miembros. Este nivel de acceso asume que las clases del mismo paquete son amigas de confianza.

10. Entrada / Salida

En JAVA hay muchas clases para leer y escribir archivos (u otros dispositivos de E/S). Están reunidos en la biblioteca **java.io**. Vamos a empezar como siempre con un pequeño ejemplo funcional y en seguida nos meteremos en el necesario camino de las excepciones...

```
import java.io.*;
public class Lolo9 {
    public static void main(String args[]) throws FileNotFoundException,IOException {
            FileInputStream fptr;
            DataInputStream f;
            String linea = null;
            fptr = new FileInputStream(«Lolo9.java»);
            f = new DataInputStream(fptr);
            do {
               linea = f.readLine();
               f (linea!=null) System.out.println(linea);
            } while (linea != null);
            fptr.close();
    }
}
```

El programa de ejemplo simplemente lee un archivo de texto y lo muestra en pantalla, algo así como el type del DOS o el cat de Unix.

Dejemos por ahora el throws FileNotFoundException,IOException y vamos al código.

fptr = new FileInputStream(«Lolo9.java»);

La clase FileInputStream (descendiente de InputStream) nos sirve para referirnos a archivos o conexiones (sockets) de una máquina. Podemos accederlos pasando un String como aquí, un objeto de tipo File o uno de tipo FileDescriptor, pero en esencia es lo mismo. Al crear un objeto de este tipo estamos «abriendo» un archivo, clásicamente hablando.

Si el archivo no existe (por ejemplo reemplacen «Lolo9.java» por alguna otra cosa, como «noexiste.txt»), al ejecutarlo nos aparece un error:

```
C:\java\curso>java Lolo9
java.io.FileNotFoundException: noexiste.txt
at java.io.FileInputStream.<init>(FileInputStream.java:51)
at Lolo9.main(Lolo9.java:9)
```

La clase DataInputStream nos permite leer, en forma independiente del hardware, tipos de datos de una «corriente» (stream) que, en este caso, es un archivo. Es descendiente de FilterInputStream e implementa DataInput, una interfaz.

Al crear un objeto de tipo DataInputStream lo referimos al archivo, que le pasamos como parámetro (fptr); esta clase tiene toda una serie de métodos para leer datos en distintos formatos. En nuestro programa usamos uno para leer líneas, que devuelve null cuando se llega al final del archivo o un String con el contenido de la línea:

```
do {
    linea = f.readLine();
    System.out.println(linea);
} while (linea != null);
```

Enseguida de leer la línea la imprimimos, y repetimos esto mientras no nos devuelva null.

Al final, cerramos el archivo:

```
fptr.close();
```

Tanto readLine como close pueden lanzar la excepción IOException, en caso de error de lectura o cierre de archivo.

En realidad, podríamos no haber usado un DataInputStream y trabajar en forma más directa:

```
import java.io.*;
public class Lolo10 {
    public static void main(String args[]) throws FileNotFoundEx-
    ception,IOException {
```

```
        FileInputStream fptr;
        int n;
        fptr = new FileInputStream(«Lolo9.java»);
        do {
            n = fptr.read();
            if (n!=-1) System.out.print((char)n);
        } while (n!=-1);
        fptr.close();
    }
}
```

Ya que la clase FileInputStream también dispone de métodos para leer el archivo. Solo que son unos pocos métodos que nos permiten leer un entero por vez o un arreglo de bytes. DataInputStream tiene métodos para leer los datos de muchas formas distintas, y en general resulta más cómodo.

11. Herencia

Herencia es el mecanismo por el que se crean nuevos objetos definidos en términos de objetos ya existentes. Por ejemplo, si se tiene la clase Mamífero, se puede crear la subclase Felino, que es una especialización de Mamífero.

```
class Felino extends Mamífero {
    int numero_de_patas;
}
```

La palabra cl Mamífero *extends* se usa para generar una subclase (especialización) de un objeto. Una Felino es una subclase de Mamífero. Cualquier cosa que contenga la definición de Mamífero será copiada a la clase Felino; además, en Felino se pueden definir sus propios métodos y variables de instancia. Se dice que Felino deriva o hereda de Mamífero.

Además, se pueden sustituir los métodos proporcionados por la clase base. Utilizando el ejemplo de MiClase, aquí hay un ejemplo de una clase derivada sustituyendo a la función *Suma_a_i()*:

```
import MiClase;
public class MiNuevaClase extends MiClase {
    public void Suma_a_i( int j ) {
        i = i + ( j/2 );
        }
    }
```

Ahora, cuando se crea una instancia de MiNuevaClase, el valor de "i" también se inicializa a 10, pero la llamada al método *Suma_a_i()* produce un resultado diferente:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

En JAVA no se puede hacer herencia múltiple. Por ejemplo, de la clase *aparato con motor* y de la clase *animal* no se puede derivar nada, sería como obtener el objeto toro mecánico a partir de una *máquina motorizada* (aparato con motor) y un toro (animal). En realidad, lo que se pretende es copiar los métodos, es decir, pasar la funcionalidad del toro de verdad al toro mecánico, con lo cual no sería necesaria la herencia múltiple sino simplemente la compartición de funcionalidad que se encuentra implementada en JAVA a través de interfaces.

En JAVA, como en otros lenguajes de programación orientados a objetos, las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama subclase. La clase de la que está derivada se denomina superclase. De hecho, en JAVA, todas las clases deben derivar de alguna clase. Lo que nos lleva a la cuestión ¿dónde empieza todo esto? La clase más alta, la clase de la que todas las demás descienden, es la clase Object, definida en java.lang. Object es la raíz de la herencia de todas las clases. Las subclases heredan el estado y el comportamiento en forma de las variables y los métodos de su superclase. La subclase puede utilizar los ítems heredados de su superclase tal y como son, o puede modificarlos o sobreescribirlos. Por eso, según se va bajando por el árbol de la herencia, las clases se convierten en más y más especializadas.

Una subclase es una clase que desciende de otra clase. Una subclase hereda el estado y el comportamiento de todos sus ancestros. El término superclase se refiere a la clase que es el ancestro más directo, así como a todas las clases ascendentes.

11.1. Crear subclases

Se declara que un clase es una subclase de otra clase dentro de la declaración de clase. Por ejemplo, supongamos que queremos crear una subclase llamada SubClase de otra clase llamada SuperClase. Se escribiría esto:

```
class SubClass extends SuperClass {
   . . .
}
```

Esto declara que SubClase es una subclase de SuperClase. Y también declara implícitamene que SuperClase es la superclase de SubClase. Una subclase también hereda variables y miembros de las superclases de su superclase, y así a lo largo del árbol de la herencia. Para hacer esta explicación un poco más sencilla, cuando este tutorial se refiere a la superclase de una clase significa el ancestro más directo de la clase así como a todas sus clases ascendentes.

Una clase JAVA solo puede tener una superclase directa. JAVA no soporta la herencia múltiple.

Crear una subclase puede ser tan sencillo como incluir la clausula **extends** en la declaración de la clase. Sin embargo, normalmente se deberá realizar alguna cosa más cuando se crea una subclase, como sobreescribir métodos, etc.

11.2. ¿Qué variables miembro hereda una subclase?

Una subclase hereda todas las variables miembros de su superclase que puedan ser accesibles desde la subclase (a menos que la variable miembro esté oculta en la subclase).

Esto es, las subclases:

- Heredan aquellas variables miembros declaradas como **public** o **protected**.
- Heredan aquellas variables miembros declaradas sin especificador de acceso (normalmente conocidas como «friendly») siempre que la subclases esté en el mismo paquete que la clase.
- No heredan las variables miembros de la superclase si la subclase declara una variable miembro que utiliza el mismo nombre. La variable miembro de la subclase se dice que oculta a la variable miembro de la superclase.
- No heredan las variables miembro **private**.

11.3. Ocultar variables miembro

Como se mencionó en la sección anterior, las variables miembros definidas en la subclase ocultan las variables miembro que tienen el mismo nombre en la superclase. Como esta característica del lenguaje JAVA es poderosa y conveniente, puede ser una fuente de errores: ocultar una variable miembro puede hacerse deliberadamente o por accidente. Entonces, cuando se nombren variables miembro se ha de ser cuidadoso y ocultar solo las variables miembro que realmente se desean ocultar. Una característica interesante de las variables miembro en JAVA es que una clase puede acceder a una variable miembro oculta a través de su superclase. Considere este pareja de superclase y subclase:

```
class Super {
    Number unNumero;
}
class Sub extends Super {
    Float unNumero;
}
```

La variable **unNumero** de Sub oculta a la variable **unNumero** de Super. Pero se puede acceder a la variable de la superclase utilizando: super.unNumero

Super es una palabra clave del lenguaje JAVA que permite a un método referirse a las variables ocultas y métodos sobreescritos de una superclase (ya la hemos estudiado en un apartado anterior en profundidad).

11.4. ¿Qué métodos hereda una subclase?

La regla que especifica los métodos heredados por una subclase es similar a la de las variables miembro. Una subclase hereda todos los métodos de sus superclase que son accesibles para la subclase (a menos que el método sea sobreescrito por la subclase).

Esto es, una subclase:

- Hereda aquellos métodos declarados como **public** o **protected**.
- Hereda aquellos métodos sin especificador de acceso, siempre que la subclase esté en el mismo paquete que la clase.
- No hereda un método de la superclase si la subclase declara un método que utiliza el mismo nombre. Se dice que el método de la subclase sobreescribe al método de la superclase.
- No hereda los métodos **private**.

11.5. Sobreescribir métodos

La habilidad de una subclase para sobreescribir un método de su superclase permite a una clase heredar de su superclase aquellos comportamientos «más cercanos» y luego suplementar o modificar el comportamiento de la superclase.

Una subclase puede sobreescribir completamente la implementación de un método heredado o puede mejorar el método añadiendole funcionalidad.

11.6. Redefinir el método de una superclase

Algunas veces, una subclase querría reemplazar completamente la implementación de un método de su superclase. De hecho, muchas superclases proporcionan implementaciones de métodos vacías con la esperanza de que la mayoría, si no todas, de sus subclases reemplacen completamente la implementación de ese método.

Un ejemplo de esto es el método **run()** de la clase thread. La clase thread proporciona una implementación vacía (el método no hace nada) para el método **run()**, porque por definición este método depende de la subclase. La clase thread posiblemente no puede proporcionar una implementación medianamente razonable del método **run()**.

Para reemplazar completamente la implementación de un método de la superclase, simplemente se llama a un método con el mismo nombre

que el del método de la superclase y se sobreescribe el método con la misma firma que la del método sobreescrito:

```
class ThreadSegundoPlano extends Thread {
   void run() {
      . . .
   }
}
```

La clase ThreadSegundoPlano sobreescribe completamente el método **run()** de su superclase y reemplaza completamente su implementación.

11.7. Añadir implementación a un método de la superclase

Otras veces una subclase querrá mantener la implememtación del método de su superclase y posteriormente ampliar algún comportamiento específico de la subclase. Por ejemplo, los métodos constructores de una subclase lo hacen normalmente —la subclase quiere preservar la inicialización realizada por la superclase, pero proporciona inicialización adicional específica de la subclase.

Supongamos que queremos crear una subclase de la clase Windows del paquete java.awt. La clase Windows tiene un constructor que requiere un argumento del tipo Frame que es el padre de la ventana:

```
public Window(Frame parent)
```

Este constructor realiza alguna inicialización en la ventana para que trabaje dentro del sistema de ventanas. Para asegurarnos de que una subclase de Window también trabaja dentro del sistema de ventanas, deberemos proporcionar un constructor que realice la misma inicialización. Mucho mejor que intentar recrear el proceso de inicialización que ocurre dentro del constructor de Windows, se podría utilizar lo que la clase Windows ya hace. Se puede utilizar el código del constructor de Windows llamándolo desde dentro del constructor de la subclase Window:

```
class Ventana extends Window {
      public Ventana(Frame parent) {
            super(parent);
            . . .
                // Ventana especifica su inicialización aquí
             . . .
      }
}
```

El constructor de **Ventana** llama primero al constructor de su superclase, y no hace nada más. Típicamente, este es el comportamiento deseado de los constructores —las superclases deben tener la oportunidad de realizar sus tareas de inicialización antes que las de su subclase—. Otros tipos de métodos podrían llamar al constructor de la supeclase al final del método o en el medio.

11.8. Métodos que una subclase no puede sobrescibir

Una subclase no puede sobrescribir métodos que hayan sido declarados como **final** en la superclase (por definición, los métodos finales no pueden ser sobrescritos). Si intentamos sobrescribir un método final, el compilador mostrará un mensaje similar a este y no compilará el programa:

```
FinalTest.java:7: Final methods can't be overriden. Method void iamfinal() is
final in class ClassWithFinalMethod.
void iamfinal() {
     ^
1 error
```

Para una explicación sobre los métodos finales, puede ver: escribir clases y métodos finales (epig. 11.10).

Una subclase tampoco pude sobrescribir métodos que se hayan declarado como **static** en la superclase. En otras palabras, una subclase no puede sobrescribir un método de clase.

11.9. Métodos que una subclase debe sobrescribir

Las subclases deben sobreescribir aquellos métodos que hayan sido declarados como **abstract** en la superclase, o la propia subclase debe ser abstracta.

11.10. Escribir clases y métodos finales

A) Clases finales

Se puede declarar que una clase sea final; esto es, que la clase no pueda tener subclases. Existen (al menos) dos razones por las que se querría hacer esto: razones de seguridad y de diseño.

Un mecanismo que los hackers utilizan para atacar sistemas es crear subclases de una clase y luego sustituirla por el original. Las subclases parecen y sienten como la clase original pero hacen cosas bastante diferentes, probablemente causando daños u obteniendo información privada. Para prevenir esta clase de subversión, se puede declarar que la clase sea final y así prevenir que se cree cualquier subclase.

La clase String del paquete java.lang es una clase final solo por esta razón. La clase String es tan vital para la operación del compilador y del intérprete que el sistema JAVA debe garantizar que siempre que un método o un objeto utilicen un String, obtenga un objeto java.lang.String y no algún otro string. Esto asegura que ningún string tendrá propiedades extrañas, inconsistentes o indeseables.

Si se intenta compilar una subclase de una clase final, el compilador mostrará un mensaje de error y no compilará el programa. Además, los bytescodes verifican que no está teniendo lugar una subversión al nivel de byte, comprobando que una clase no es una subclase de una clase final.

Otra razón por la que se podría querer declarar una clase final son razones de diseño orientado a objetos. Se podría pensar que una clase es «perfecta» o que, conceptualmente hablando, la clase no debería tener subclases.

Para especificar que una clase es una clase final, se utiliza la palabra clave **final** antes de la palabra clave **class** en la declaración de la clase. Por ejemplo, si quisieramos declarar AlgoritmodeAjedrez como una clase final (perfecta), la declaración se parecería a esto:

```
final class AlgoritmodeAjedrez {
    . . .
}
```

Cualquier intento posterior de crear una subclase de AlgoritmodeAjedrez resultará en el siguiente error del compilador:

```
Chess.java:6: Can't subclass final classes: class AlgoritmodeAjedrez
class MejorAlgoritmodeAjedrez extends AlgoritmodeAjedrez {
    ^
1 error
```

B) Métodos Finales

Si la creacción de clases finales parece algo dura para nuestras necesidades, y realmente lo que se quiere es proteger algunos métodos de una clase para que no sean sobrescritos, se puede utilizar la palabra clave **final** en la declaración de método para indicar al compilador que este método no puede ser sobrescrito por las subclases.

Se podría desear hacer que un método fuera final si el método tiene una implementación que no debe ser cambiada y que es crítica para el estado consistente del objeto. Por ejemplo, en lugar de hacer AlgoritmodeAjedrez como una clase final, podríamos hacer **siguienteMovimiento()** como un método final:

```
class AlgoritmodeAjedrez {
    . . .
    final void siguienteMovimiento(Pieza piezaMovida, PosicionenTablero nuevaPo-
    sicion) {
    }
    . . .
}
```

12. Clases abstractas

Algunas veces, una clase que se ha definido representa un concepto abstracto y, como tal, no debe ser ejemplarizado. Por ejemplo, la comida en la vida real. ¿Ha visto algún ejemplar de comida? No. Lo que ha visto son ejemplares de manzanas, pan y chocolate. Comida representa un concepto abstracto de cosas que son comestibles. No tiene sentido que exista un ejemplar de comida.

Similarmente, en la programación orientada a objetos se podría modelar conceptos abstractos pero no querer que se creen ejemplares de ellos. Por ejemplo, la clase Number del paquete java.lang representa el concepto abstracto de número. Tiene sentido modelar números en un programa, pero no tiene sentido crear un objeto genérico de números. En su lugar, la clase Number solo tiene sentido como superclase de otras clases como Integer y Float que implementan números de tipos específicos. Las clases como Number, que implementan conceptos abstractos y no deben ser ejemplarizadas, son llamadas clases abstractas. Una clase abstracta es una clase que solo puede tener subclases —no puede ser ejemplarizada—.

Para declarar que una clase es un clase abstracta, se utiliza la palabra clave **abstract** en la declaración de la clase.

```
abstract class Number {
    . . .
}
```

Si se intenta ejemplarizar una clase abstracta, el compilador mostrará un error similar a este y no compilará el programa:

```
AbstractTest.java:6: class AbstractTest is an abstract class. It can't be instantiated.
    new AbstractTest();
    ^
1 error
```

12.1. Métodos abstractos

Una clase abstracta puede contener *métodos abstractos*, esto es, métodos que no tienen implementación. De esta forma, una clase abstracta puede definir una interfaz de programación completa, incluso porporciona a sus subclases la

declaración de todos los métodos necesarios para implementar la interfaz de programación. Sin embargo, las clases abstractas pueden dejar algunos detalles o toda la implementación de aquellos métodos a sus subclases.

Veamos un ejemplo de cuándo sería necesario crear una clase abstracta con métodos abstractos: en una aplicación de dibujo orientada a objetos, se pueden dibujar círculos, rectángulos, líneas, etc. Cada uno de esos objetos gráficos comparten ciertos estados (posición, caja de dibujo) y comportamiento (movimiento, redimensionado, dibujo). Podemos aprovecharnos de esas similitudes y declararlos todos a partir de un mismo objeto padre-ObjetoGrafico. Sin embargo, los objetos gráficos también tienen diferencias sustanciales: dibujar un círculo es bastante diferente a dibujar un rectángulo. Los objetos gráficos no pueden compartir estos tipos de estados o comportamientos. Por otro lado, todos los ObjetosGraficos deben saber cómo dibujarse a sí mismos; se diferencian en cómo se dibujan unos y otros. Esta es la situación perfecta para una clase abstracta.

Primero se debe declarar una clase abstracta, ObjetoGrafico, para proporcionar las variables miembro y los métodos que van a ser compartidos por todas las subclases, como la posición actual y el método **moverA()**.

También se deberían declarar métodos abstractos como **dibujar(),** que necesita ser implementado por todas las subclases, pero de manera completamente diferente (no tiene sentido crear una implementación por defecto en la superclase). La clase ObjetoGrafico se parecería a esto:

```
abstract class ObjetoGrafico {
    int x, y;
    . . .
    void moverA(int nuevaX, int nuevaY) {
        . . .
    }
    abstract void dibujar();
}
```

Todas las subclases no abstractas de ObjetoGrafico como son Circulo o Rectangulo deberán proprocionar una implementación para el método **dibujar()**.

```
class Circulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
class Rectangulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
```

Una clase abstracta no necesita contener un método abstracto. Pero todas las clases que contengan un método abstracto o no proporcionen implemenación para cualquier método abstracto declarado en sus superclases debe ser declarada como una clase abstracta.

13. La clase Object

La clase Object está situada en la parte más alta del árbol de la herencia en el entorno de desarrollo de JAVA. Todas las clases del sistema JAVA son descendentes (directos o indirectos) de la clase Object. Esta clase define los estados y comportamientos básicos que todos los objetos deben tener, como la posibilidad de compararse unos con otros, de convertirse a cadenas, de esperar una condición variable, de notificar a otros objetos que la condición variable ha cambiado y devolver la clase del objeto.

13.1. El método equals()

Equals() se utiliza para comparar si dos objetos son iguales. Este método devuelve **true** si los objetos son iguales, o **false** si no lo son. Observe que la igualdad no significa que los objetos sean el mismo objeto. Consideremos este código que compara dos enteros:

```
Integer uno = new Integer(1), otroUno = new Integer(1);
if (uno.equals(otroUno))
    System.out.println(«Los objetos son Iguales»);
```

Este código mostrará **Los objetos son Iguales** aunque **uno** y **otroUno** referencian a dos objetos distintos. Se les considera iguales porque su contenido es el mismo valor entero.

Las clases deberían sobrescribir este método proporcionando la comprobación de igualdad apropiada. Un método **equals()** debería comparar el contenido de los objetos para ver si son funcionalmente iguales y devolver **true** si es así.

13.2. El método getClass()

El método **getClass()** es un método final (no puede sobrescribirse) que devuelve una representación en tiempo de ejecución de la clase del objeto. Este método devuelve un objeto Class al que se le puede pedir varia información sobre la clase, como su nombre, el nombre de su superclase y los nombres de los interfaces que implementa. El siguiente método obtiene y muestra el nombre de la clase de un objeto:

```
void PrintClassName(Object obj) {
    System.out.println(«La clase del Objeto es « + obj.getClass().get-
    Name());
}
```

Un uso muy manejado del método **getClass()** es crear un ejemplar de una clase sin conocer la clase en el momento de la compilación. Este método de ejemplo, crea un nuevo ejemplar de la misma clase que **obj** que puede ser cualquier clase heredada desde Object (lo que significa que podría ser cualquier clase):

```
Object createNewInstanceOf(Object obj) {
    return obj.getClass().newInstance();
}
```

13.3. El método toString()

Este método devuelve una cadena de texto que representa al objeto. Se puede utilizar **toString** para mostrar un objeto. Por ejemplo, se podría mostrar una representación del Thread actual de la siguiente forma:

```
System.out.println(Thread.currentThread().toString());
System.out.println(new Integer(44).toString());
```

La representación de un objeto depende enteramente del objeto. El String de un objeto entero es el valor del entero mostrado como texto. El String de un objeto Thread contiene varios atributos sobre el thread, como su nombre y prioridad. Por ejemplo, las dos líneas anteriores darían la siguiente salida:

```
Thread[main,5,main]
4
```

El método **toString()** es muy útil para depuración y también puede sobreescribir este método en todas las clases.

14. Interface

Un interface es una colección de definiciones de métodos (sin implementaciones) y de valores constantes.

Los interfaces se utilizan para definir un protocolo de comportamiento que puede ser implementado por cualquier clase del árbol de clases.

Los interfaces son útiles para:

- Capturar similitudes entre clases no relacionadas sin forzar una relación entre ellas.
- Declarar métodos que una o varias clases necesitan implementar.
- Revelar el interface de programación de un objeto sin recelar sus clases (los objetos de este tipo son llamados objetos anónimos y

pueden ser útiles cuando compartas un paquete de clases con otros desarrolladores).

En JAVA, un interface es un tipo de dato de referencia y, por tanto, puede utilizarse en muchos de los sitios donde se pueda utilizar cualquier tipo (como en un argumento de métodos y una declaración de variables).

14.1. Los Interfaces no proporcionan herencia múltiple

Algunas veces se tratra a los interfaces como una alternativa a la herencia múltiple en las clases. A pesar de que los interfaces podrían resolver algunos problemas de la herencia múltiple, son animales bastantes diferentes.

En particular:

- No se pueden heredar variables desde un interface.
- No se pueden heredar implementaciones de métodos desde un interface.
- La herencia de un interface es independiente de la herencia de la clase —las clases que implementan el mismo interface pueden o no estar relacionadas a través del árbol de clases—.

14.2. Definir un Interface

Para crear un Interface, se debe escribir tanto la declaración como el cuerpo del mismo:

```
declaracionde Interface {
    cuerpode Interface
}
```

La **Declaración de Interface** declara varios atributos del interface, como su nombre o si se extiende desde otro interface. El **Cuerpo de Interface** contiene las constantes y las declaraciones de métodos del Interface.

14.3. La declaración de interface

Como mínimo, una declaración de interface contiene la palabra clave **interface** y el nombre del interface que se va a crear:

```
"interface" Contable {
    . . .
}
```

Por convención, los nombres de interfaces empiezan con una letra mayúscula al igual que las clases. Frecuentemente los nombres de interfaces terminan en «able» o «ible».

Una declaración de interface puede tener otros dos componentes: el especificador de acceso **public** y una lista de «superinterfaces». Un interface puede extender otros interfaces como una clase puede extender o subclasificar otra case. Sin embargo, mientras que una clase solo puede extender una superclase, los interfaces pueden extender de cualquier número de interfaces. Así, una declaración completa de interface se parecería a esto:

```
[public] interface Nombredenterface [extends listadeSuperInterfaces] {
    . . .
}
```

El especificador de acceso **public** indica que el interface puede ser utilizado por todas las clases en cualquier paquete. Si el interface no se especifica como público, solo será accesible para las clases definidas en el mismo paquete que el interface.

La clausula **extends** es similar a la utilizada en la declaración de una clase; sin embargo, un interface puede extender varios interfaces (mientras una clase solo puede extender una), y un interface no puede extender clases. Esta lista de superinterfaces es un lista delimitada por comas de todos los interfaces extendidos por el nuevo interface.

Un interface hereda todas las constantes y métodos de sus superinterfaces a menos que el interface oculte una constante con el mismo nombre o redeclare un método con una nueva declaración.

14.4. El cuerpo del Interface

El cuerpo del interface contiene las declaraciones de métodos para los métodos definidos en el interface. Implementar Métodos muestra cómo escribir una declaración de método. Además de las declaraciones del métodos, un interface puede contener declaraciones de constantes.

Las declaraciones de miembros en un interface no permiten el uso de algunos modificadores y desaconsejan el uso de otros. No se podrán utilizar transient, volatile, o synchronized en una declaración de miembro en un interface. Tampoco se podrá utilizar los especificadores private y protected cuando se declaren miembros de un interface.

Todos los valores constantes definidos en un interfaces son implicitamente públicos, estáticos y finales. El uso de estos modificadores en una declaración de constante en un interface está desaconsejado por falta de estilo. Similarmente, todos los métodos declarados en un interface son implícitamente públicos y abstractos.

Este código define un nuevo interface llamado Coleccion que contiene un valor constante y tres declaraciones de métodos:

```
interface Coleccion {
    int MAXIMO = 500;

    void añadir(Object obj);
    void borrar(Object obj);
    Object buscar(Object obj);
    int contadorActual();
}
```

El interface anterior puede ser implementado por cualquier clase que represente una colección de objetos como pueden ser pilas, vectores, enlaces, etc...

Observa que cada declaración de método está seguida por un punto y coma (;) porque un interface no proporciona implementación para los métodos declarados dentro de él.

14.5. Implementar un Interface

Para utilizar un interface se debe escribir una clase que lo implemente. Una clase declara todos los interfaces que implementa en su declaración de clase. Para declarar que una clase implementa uno o más interfaces, se utiliza la palabra clave **implements** seguida por una lista delimitada por comas con los interfaces implementados por la clase.

Por ejemplo, consideremos el interface Coleccion presentado en la página anterior. Ahora, supongamos que queremos escribir una clase que implemente un pila FIFO (primero en entrar, primero en salir). Como una pila FIFO contiene otros objetos tiene sentido que implemente el interface Coleccion. La clase PilaFIFO declara que implementa el interface Coleccion de esta forma:

```
class PilaFIFO implements Coleccion {
    . . .
    void añadir(Object obj) {
        . . .
    }
    void borrar(Object obj) {
        . . .
    }
    Object buscar(Object obj) {
        . . .
    }
```

```
    int contadorActual() {
        . . .
    }
}
```

Así se garantiza que proporciona implementación para los métodos **añadir(), borrar(), buscar() y contadorActual()**.

Por convención, la cláusula **implements** sigue a la cláusula **extends** si es que esta existe.

Observe que las firmas de los métodos del interface Coleccion implementados en la clase PilaFIFO debe corresponder exactamente con las firmas de los métodos declarados en la interface Coleccion.

14.6. Utilizar un Interface como un tipo

Como se mencionó anteriormente, cuando se define un nuevo interface, en esencia se está definiendo un tipo de referencia. Se pueden utilizar los nombres de interface en cualquier lugar donde se usaría un nombre de dato de tipos primitivos o un nombre de datos del tipo de referencia.

Por ejemplo, supongamos que se ha escrito un programa de hoja de cálculo que contiene un conjunto tabular de celdas y cada una contiene un valor. Querríamos poder poner cadenas, fechas, enteros, ecuaciones, en cada una de las celdas de la hoja. Para hacer esto, las cadenas, las fechas, los enteros y las ecuaciones tienen que implementar el mismo conjunto de métodos. Una forma de conseguir esto es encontrar el ancestro común de las clases e implementar ahí los métodos necesarios. Sin embargo, esto no es una solución práctica porque el ancestro común más frecuente es Object. De hecho, los objetos que puede poner en las celdas de su hoja de cálculo no están relacionadas entre sí, solo por la clase Object. Pero no puede modificar Object.

Una aproximación podría ser escribir una clase llamada ValordeCelda que representara los valores que pudiera contener una celda de la hoja de cálculo. Entonces se podrían crear distintas subclases de ValordeCelda para las cadenas, los enteros o las ecuaciones. Además de ser mucho trabajo, esta aproximación arbitraria fuerza una relación entre esas clases que de otra forma no sería necesaria, y debería duplicar e implementar de nuevo clases que ya existen.

Se podría definir un interface llamado CellAble que se parecería a esto:

```
interface CellAble {
    void draw();
    void toString();
    void toFloat();
}
```

Ahora, supongamos que existen objetos Línea y Columna que contienen un conjunto de objetos que implementan el interface CellAble. El método **setObjectAt()** de la clase Línea se podría parecer a esto:

```
class Línea {
    private CellAble[ ] contents;
    . . .
    void setObjectAt(CellAble ca, int index) {
        . . .
    }
    . . .
}
```

Observe el uso del nombre del interface en la declaración de la variable miembro **contents** y en la declaración del argumento **ca** del método. Cualquier objeto que implemente el interface CellAble, sin importar que exista o no en el árbol de clases, puede estar contenido en el array **contents** y podría ser pasado al método **setObjectAt()**.

15. Paquetes

Los paquetes son grupos relacionados de clases e interfaces y proporcionan un mecanismo conveniente para manejar un gran juego de clases e interfaces y evitar los conflictos de nombres. Además de los paquetes de JAVA, se pueden crear paquetes propios y poner en ellos definiciones de clases y de interfaces utilizando la sentencia **package**.

Supongamos que se está implementando un grupo de clases que representan una colección de objetos gráficos como círculos, rectángulos, líneas y puntos. Además de estas clases se debería escribir un interface Draggable para que las clases que lo implementen puedan moverse con el ratón. Si se quiere que estas clases estén disponibles para otros programadores, puedes empaquetarlas en un paquete, digamos, **graphics** y entregar el paquete a los programadores (junto con alguna documentación de referencia como qué hacen las clases y los interfaces y qué interfaces de programación son públicos).

De esta forma, otros programadores pueden determinar fácilmente para qué es tu grupo de clases, cómo utilizarlos y cómo relacionarlos unos con otros, y con otras clases y paquetes. Los nombres de clases no tienen conflictos con los nombres de las clases de otros paquetes porque las clases y los interfaces dentro de un paquete son referenciados en términos de su paquete.

Se declara un paquete utilizando la sentencia package:

```
package graphics;
interface Draggable {    . . .
}
```

```
class Circle {    . . .
}
class Rectangle {    . . .
}
```

La primera línea del código anterior crea un paquete llamado **graphics**. Todas las clases e interfaces definidas en el fichero que contiene esta sentencia son miembros del paquete. Por lo tanto, Draggable, Circle,y Rectangle son miembros del paquete **graphics**.

Los ficheros **.class** generados por el compilador cuando se compila el fichero que contiene el fuente para Draggable, Circle y Rectangle debe situarse en un directorio llamado **graphics** en algún lugar se el path **CLASSPATH**. **CLASSPATH** es una lista de directorios que indica al sistema donde ha instalado varias clases e interfaces compiladas JAVA. Cuando busque una clase, el intérprete JAVA busca un directorio en su **CLASSPATH** cuyo nombre coincida con el nombre del paquete del que la clase es miembro. Los ficheros **.class** para todas las clases e interfaces definidas en un paquete deben estar en ese directorio de paquete.

Los nombres de paquetes pueden contener varios componentes (separados por puntos). De hecho, los nombres de los paquetes de JAVA tienen varios componentes: **java.util**, **java.lang**, etc... Cada componente del nombre del paquete representa un directorio en el sistema de ficheros. Así, los ficheros **.class** de **java.util** están en un directorio llamado util en otro directorio llamado **JAVA** en algún lugar del **CLASSPATH**.

Todas las clases e interfaces pertenecen a un paquete. Incluso si no especifica uno con la sentencia **package**. Si no se especifican las clases e interfaces se convierten en miembros del paquete por defecto, que no tiene nombre y que siempre es importado.

15.1. Utilizar Clases e Interfaces desde un paquete

Para importar una clase específica o un interface al fichero actual (como la clase Circle desde el paquete graphics creado en la sección anterior) se utiliza la sentencia de **import**:

```
import graphics.Circle;
```

Esta sentencia debe estar al principio del fichero antes de cualquier definición de clase o de interface y hace que la clase o el interface esté disponible para su uso por las clases y los interfaces definidos en el fichero.

Si se quiere importar todas las clases e interfaces de un paquete, por ejemplo, el paquete **graphics** completo, se utiliza la sentencia **import** con un caracter comodín, un asterisco '*'.

```
import graphics.*;
```

Si intenta utilizar una clase o un interface desde un paquete que no ha sido importado, el compilador mostrará este error:

testing.java:4: Class Date not found in type declaration.

Date date;

^

Observe que solo las clases e intefaces declarados como públicos pueden ser utilizados en clases fuera del paquete en el fueron definidos.

El paquete por defecto (un paquete sin nombre) siempre es importado. El sistema de ejecución también importa automáticamente el paquete **java.lang**.

Si, por suerte, el nombre de una clase de un paquete es el mismo que el nombre de una clase en otro paquete, se debe evitar la ambigüedad de nombres precediendo el nombre de la clase con el nombre del paquete. Por ejemplo, previamente se ha definido una clase llamada Rectangle en el paquete **graphics**. El paquete **java.awt** también contiene una clase Rectangle. Si estos dos paquetes son importados en la misma clase, el siguiente código sería ambigüo:

Rectangle rect;

En esta situación se tiene que ser más específico e indicar exactamente qué clase Rectangle se quiere:

graphics.Rectangle rect;

Se puede hacer esto anteponiendo el nombre del paquete al nombre de la clase y separando los dos con un punto.

15.2. Los paquetes de JAVA

A) El paquete de lenguaje JAVA

El paquete **java.lang**, contiene las clases principales de JAVA, y se importa automáticamente:

— **Object**. El abuelo de todas las clases —la clase de la que parten todas las demás. Esta clase se cubrió anteriormente en la lección La Clase Object.

— **Tipos de datos encubiertos**. Una colección de clases utilizadas para encubrir variables de tipos primitivos: Boolean, Character, Double, Float, Integer y Long. Cada una de estas clases es una subclase de la clase abstracta Number.

— **Strings**. Dos clases que implementan los datos de caracteres.

— **System y Runtime.** Estas dos clases permiten a los programas utilizar los recursos del sistema. System proporciona un interface de programación independiente del sistema para recursos del sistema y Runtime da acceso directo al entorno de ejecución específico de un sistema.

— **Thread**. Las clases Thread, ThreadDeath y ThreadGroup implementan las capacidades multitareas tan importantes en el lenguaje JAVA. El paquete java.lang también define el interface Runnable. Este interface es conveniente para activar la clase JAVA sin subclasificar la clase Thread.

— **Class**. La clase Class proporciona una descripción en tiempo de ejecución de una clase y la clase ClassLoader permite cargar clases en los programas durante la ejecución.

— **Math**. Una librería de rutinas y valores matemáticos como pi.

— **Exceptions, Errors y Throwable.** Cuando ocurre un error en un programa JAVA, el programa lanza un objeto que indica qué problema era y el estado del intérprete cuando ocurrió el error. Solo los objetos derivados de la clase Throwable puden ser lanzados. Existen dos subclasses principales de Throwable: Exception y Error. Exception es la forma que deben intentar capturar los programas normales. Error se utiliza para los errores catastróficos —los programas normales no capturan Errores—. El paquete java.lang contiene las clases Throwable, Exception y Error, y numerosas subclases de Exception y Error que representan problemas específicos.

— **Process**. Los objetos Process representa el proceso del sistema que se crea cuando se utiliza el sistema en tiempo de ejecución para ejecutar comandos del sistema. El paquete java.lang define e implementa la clase genérica Process.

B) El Paquete I/O de JAVA

El paquete I/O de JAVA (java.io) proporciona un juego de canales de entrada y salida utilizados para leer y escribir ficheros de datos y otras fuentes de entrada y salida. Las clases e interfaces definidos en **java.io** se cubren completamente en Canales de Entrada y Salida.

C) El paquete de utilidades de JAVA

Este paquete, **java.util**, contiene una colección de clases útiles. Entre ellas se encuentan muchas estructuras de datos genéricas (Dictionary, Stack, Vector, Hashtable), un objeto muy útil para dividir cadenas y otro para la manipualción de calendarios. El paquete **java.util** también contiene el interface Observer y la clase Observable que permiten a los objetos notificarse unos a otros cuando han cambiado.

D) El paquete de red de JAVA

El paquete **java.net** contiene definiciones de clases e interfaces que implementan varias capacidades de red. Las clases de este paquete incluyen una clase que implementa una conexión URL. Se puede utilizar estas clases

para implementar aplicaciones cliente-servidor y otras aplicaciones de comunicaciones. Conectividad y Seguridad del Cliente tiene varios ejemplos de utilización de estas clases, incluyendo un ejemplo cliente-servidor que utiliza datagramas.

E) El paquete applet

Este paquete contiene la clase applet —la clase que se debe subclasificar si se quiere escribir un applet—. En este paquete se incluye el interface AudioClip que proporciona una abstración de alto nivel para audio.

F) Paquetes de herramientas para ventanas abstractas

Tres paquetes componen las herramientas para ventanas abstractas: **java.awt, java.awt.image y java.awt.peer**.

— El paquete AWT. El paquete **java.awt** proporciona elementos GUI utilizados para obtener información y mostrarla en la pantalla como ventanas, botones, barras de desplazamiento, etc.

— El paquete AWT Image. El paquete **java.awt.image** contiene clases e interfaces para manejar imágenes de datos, como la selección de un modelo de color, el cortado y pegado, el filtrado de colores, la selección del valor de un píxel y la grabación de partes de la pantalla.

— El paquete AWT Peer. El paquete **java.awt.peer** contiene clases e interfaces que conectan los componentes AWT independientes de la plataforma a su implementación dependiente de la plataforma (como son los controles de Microsoft Windows).

16. Excepciones

En el lenguaje JAVA, una Exception es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. JAVA implementa excepciones dándole al usuario la oportunidad de corregir el error de forma que se pueden capturar y recuperar.

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

Algunas excepciones son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras excepciones, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pue-

den ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (definiendo por ejemplo un nuevo path del fichero no encontrado).

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

- **Bloque try {...} catch {...} finally {...}**

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

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

De esta forma se puede ir pasando la Exception de un método a otro hasta llegar al último método del programa, el método main().

```
import java.io.*;
public class Lolo {
   public static void main(String args[]) {
      FileInputStream fptr;
      DataInputStream f;
      String linea = null;
      try {
         fptr = new FileInputStream(args[0]);
         f = new DataInputStream(fptr);
            do {
            linea = f.readLine();
            if (linea!=null) System.out.println(linea);
         } while (linea != null);
         fptr.close();
      }
```

```
        catch (FileNotFoundException e) {
            System.out.println(«Hey, ese archivo no existe!\n»);
        }
        catch (IOException e) {
            System.out.println(«Error de E/S!\n»);
        }
    }
}
```

También hicimos un cambio para elegir el archivo a imprimir desde la línea de comandos, en lugar de entrarlo fijo, utilizando para eso el argumento del método main(arg[]), que consiste en una lista de Strings con los parámetros que se pasan en la línea a continuación de java nombre_programa. Por ejemplo, si llamamos a este programa con:

java Lolo archi.txt otro.xxx

arg[0] contendrá «archi.txt», arg[1] contendrá «otro.xxx», y así sucesivamente.

Por supuesto, si llamamos a Lolo sin parámetros se lanzará otra excepción al intentar accederlo:

C:\java\curso>java Lolo

java.lang.ArrayIndexOutOfBoundsException: 0

at Lolo.main(Lolo.java:10)

La cláusula **try** engloba una parte del programa donde se pueden lanzar excepciones. Si una excepción se produce, JAVA busca una instrucción **catch (nombre_de_la_excepción variable)**, y, si la encuentra, ejecuta lo que esta engloba. Si no encuentra un catch para esa excepción, para el programa y muestra el error que se produjo.

Por ejemplo, para evitar este último error bastaría con agregar:

```
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println(«Debe ingresar un nombre de archivo!»);
    System.out.println(«Ej.: java Lolo pepe.txt»);
}
```

Hay que notar que cuando se lanza una excepción el programa igual se detiene, porque el código que sigue al lanzamiento de la excepción no se ejecuta. Más adelante se verá cómo se comporta esto en un objeto que fue creado por otro, y cómo usar la instrucción finally para poner una parte de código que se ejecute pase lo que pase.

16.1. Manejo de errores utilizando excepciones

Existe una regla de oro en el mundo de la programación: en los programas ocurren errores. Esto es sabido. Pero, ¿qué sucede realmente después de que ha ocurrido el error? ¿Cómo se maneja el error? ¿Quién lo maneja? ¿Puede recuperarlo el programa?

El lenguaje JAVA utiliza **excepciones** para proporcionar capacidades de manejo de errores. En este apartado se explicará qué es una excepción, cómo lanzar y capturar excepciones, qué hacer con una excepción una vez capturada, y cómo hacer un mejor uso de las excepciones heredadas de las clases proporcionadas por el entorno de desarrollo de JAVA.

Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.

Muchas clases de errores pueden utilizar excepciones —desde serios problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un array fuera de sus límites—. Cuando dicho error ocurre dentro de un método JAVA, el método crea un objeto 'exception' y lo maneja fuera, en el sistema de ejecución. Este objeto contiene información sobre la excepción, incluyendo su tipo y el estado del programa cuando ocurrió el error. El sistema de ejecución es el responsable de buscar algún código para manejar el error. En terminología JAVA, crear una objeto exception y manejarlo por el sistema de ejecución se llama **lanzar una excepción**.

Después de que un método lance una excepción, el sistema de ejecución entra en acción para buscar el manejador de la excepción. El conjunto de «algunos» métodos posibles para manejar la excepción es el conjunto de métodos de la pila de llamadas del método donde ocurrió el error. El sistema de ejecución busca hacia atrás en la pila de llamadas, empezando por el método en el que ocurrió el error, hasta que encuentra un método que contiene el «manejador de excepción» adecuado. Un manejador de excepción es considerado adecuado si el tipo de la excepción lanzada es el mismo que el de la excepción manejada por el manejador. Así, la excepción sube sobre la pila de llamadas hasta que encuentra el manejador apropiado y una de las llamadas a métodos maneja la excepción, se dice que el manejador de excepción elegido **captura la excepción**.

Si el sistema de ejecución busca exhaustivamente por todos los métodos de la pila de llamadas sin encontrar el manejador de excepción adecuado, el sistema de ejecución finaliza (y, consecuentemente, el programa JAVA también).

Mediante el uso de excepciones para manejar errores, los programas JAVA tienen las siguientes ventajas frente a las técnicas de manejo de errores tradicionales:

— Ventaja 1: Separar el Manejo de Errores del Código «Normal».

— Ventaja 2: Propagar los Errores sobre la Pila de Llamadas.

— Ventaja 3: Agrupar los Tipos de Errores y la Diferenciación de éstos.

16.2. La sentencia throw

Todos los métodos JAVA utilizan la sentencia **throw** para lanzar una excepción. Esta sentencia requiere un solo argumento: un objeto Throwable. En el sistema JAVA, los objetos lanzables son ejemplares de la clase Throwable definida en el paquete java.lang. Aquí tiene un ejemplo de la sentencia **throw**:

throw algunObjetoThrowable;

16.3. La cláusula throws

Habrá observado que la declaración del método **pop()** contiene esta cláusula:

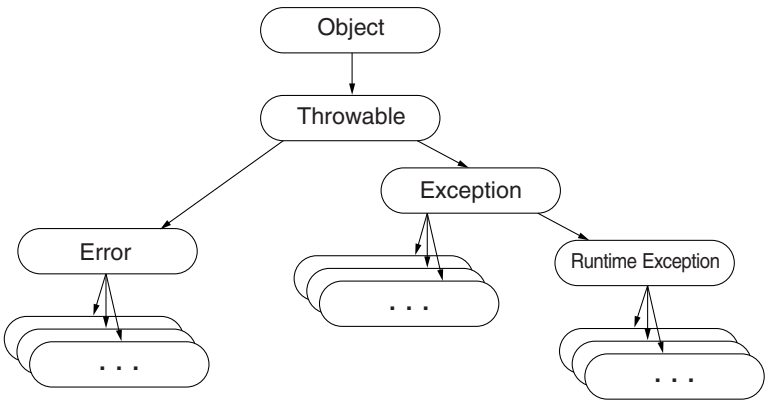
throws EmptyStackException

La cláusula **throws** especifica que el método puede lanzar una excepción EmptyStackException. Como ya sabe, el lenguaje JAVA requiere que los métodos capturen o especifiquen todas las excepciones chequeadas que puedan ser lanzadas dentro de su ámbito. Se puede hacer esto con la cláusula **throws** de la declaración del método.

16.4. La clase Throwable y sus subclases

Solo se pueden lanzar objetos que estén derivados de la clase Throwable, esto incluye descendientes directos (esto es, objetos de la clase Throwable) y descendiente indirectos (objetos derivados de hijos o nietos de la clase Throwable).

Este diagrama ilustra el árbol de herencia de la clase Throwable y sus subclases más importantes:



Como se puede ver en el diagrama, la clase Throwable tiene dos descendientes directos: Error y Exception.

A) Error

Cuando falla un enlace dinámico y hay algún fallo «hardware» en la máquina virtual, esta lanza un error. Típicamente los programas JAVA no capturan los errores, pero siempre lanzarán errores.

B) Exception

La mayoría de los programas lanzan y capturan objetos derivados de la clase Exception. Una excepción indica que ha ocurrido un problema, pero que el problema no es demasiado serio. La mayoría de los programas que se escribirán lanzarán y capturarán excepciones.

La clase Exception tiene muchos descendiente definidos en los paquetes JAVA. Estos descendientes indican varios tipos de excepciones que pueden ocurrir. Por ejemplo, IllegalAccessException señala que no se puede encontrar un método particular, y NegativeArraySizeException indica que un programa intenta crear un array con tamaño negativo.

Una subclase de Exception tiene un significado especial en el lenguaje JAVA: RuntimeException.

- **Excepciones en tiempo de ejecución**

La clase RuntimeException representa las excepciones que ocurren dentro de la máquina virtual JAVA (durante el tiempo de ejecución). Un ejemplo de estas excepciones es NullPointerException, que ocurre cuando un método intenta acceder a un miembro de un objeto a través de una referencia nula. Esta excepción puede ocurrir en cualquier lugar en que un programa intente desreferenciar una referencia a un objeto. Frecuentemente el coste de chequear estas excepciones sobrepasa los beneficios de capturarlas.

Como las excepciones en tiempo de ejecución están omnipresentes e intentar capturar o especificarlas todas en todo momento podrían ser un ejercicio infructuoso (y un código infructuoso, imposible de leer y de mantener), el compilador permite que estas excepciones no se capturen ni se especifiquen.

Los paquetes JAVA definen varias clases RuntimeException. Se pueden capturar estas excepciones al igual que las otras. Sin embargo, no se requiere que un método especifique que lanza excepciones en tiempo de ejecución. Además puedes crear sus propias subclases de RuntimeException.

17. HILOs - Threads

17.1. ¿Qué es un Thread?

Un thread, por sí mismo, no es un programa. No puede ejecutarse por sí mismo, pero sí con un programa.

Un thread es un flujo secuencial de control dentro de un programa. Proporciona la posibilidad de que un solo programa ejecute varios threads a la vez y que realicen diferentes tareas.

El navegador HotJAVA es un ejemplo de una aplicación multi-thread. Dentro del navegador HotJAVA puede moverse por la página mientras baja un applet o una imagen, se ejecuta una animación o escucha un sonido, imprime la página en segundo plano mientras descarga una nueva página, o ve cómo los tres algoritmos de ordenación alcanzan la meta.

Algunos textos utilizan el nombre proceso de poco peso en lugar de thread. Un thread es similar a un proceso real en el que un thread y un programa en ejecución son un solo flujo secuencial de control. Sin embargo, un thread se considera un proceso de poco peso porque se ejecuta dentro del contexto de un programa completo y se aprovecha de los recursos asignados por ese programa y del entorno de este.

Como un flujo secuencial de control, un thread debe conseguir algunos de sus propios recursos dentro de un programa en ejecución (debe tener su propia pila de ejecución y contador de programa, por ejemplo). El código que se ejecuta dentro de un Thread trabaja solo en este contexto. Así, algunos textos utilizan el término contexto de ejecución como un sinónimo para los threads.

Este ejemplo define dos clases: SimpleThread y TwoThreadsTest. Empecemos nuestra exploración de la aplicación con la clase SimpleThread — una subclase de la clase Thread, que es proporcionada por el paquete java.lang—:

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + « « + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println(«HECHO! « + getName());
    }
}
```

El primer método de esta clase es un constructor que toma una cadena como su único argumento. Este constructor está implementado mediante

una llamada al constructor de la superclase y es interesante para nosotros solo porque selecciona el nombre del Thread, que se usará más adelante en el programa.

El siguiente método es el método **run()**. Este método es el corazón de cualquier Thread y donde tiene lugar la acción del Thread. El método **run()** de la clase SimpleThread contiene un bucle **for** que itera diez veces. En cada iteración el método muestra el número de iteración y el nombre del Thread, luego espera durante un intervalo aleatorio de hasta 1 segundo. Después de haber terminado el bucle, el método **run()** imprime «HECHO!» con el nombre del Thread.

La clase TwoThreads proporciona un método **main()** que crea dos threads SimpleThread: uno llamado «Jamaica» y otro llamado «Fiji» (si no quiere decidir, dónde ir de vacaciones, puede utilizar este programa para ayudarte a elegir –ve a la isla cuyo threads imprima «HECHO!» primero–.

```
class TwoThreadsTest {
   public static void main (String[] args) {
      new SimpleThread(«Jamaica»).start();
      new SimpleThread(«Fiji»).start();
   }
}
```

El método **main()** también arranca cada uno de los threads inmediatamente después siguiendo su construcción con una llamada al método **start()**.

El programa daría una salida parecida a esta:

```
0 Jamaica       0 Fiji          1 Fiji          1 Jamaica
2 Jamaica       2 Fiji          3 Fiji          3 Jamaica
4 Jamaica       4 Fiji          5 Jamaica       5 Fiji
6 Fiji          6 Jamaica       7 Jamaica       7 Fiji
8 Fiji          9 Fiji          8 Jamaica       HECHO! Fiji
9 Jamaica       HECHO! Jamaica
```

Observe cómo la salida de cada uno de los threads se mezcla con la salida del otro. Esto es porque los dos threads SimpleThread se están ejecutando de forma concurrente. Así, los dos métodos **run()** se están ejecutando al mismo tiempo y cada thread está mostrando su salida al mismo tiempo que el otro.

0 Jamaica	*0 Fiji*	*1 Fiji*	*1 Jamaica*
2 Jamaica	*2 Fiji*	*3 Fiji*	*3 Jamaica*
4 Jamaica	*4 Fiji*	*5 Jamaica*	*5 Fiji*
6 Fiji	*6 Jamaica*	*7 Jamaica*	*7 Fiji*
8 Fiji	*9 Fiji*	*8 Jamaica*	*HECHO! Fiji*
9 Jamaica	*HECHO! Jamaica*		

17.2. Atributos de un Thread

Esta página presenta varias características específicas de los threads JAVA y proporciona enlaces a las páginas que explican cada característica con más detalle.

Los threads java están implementados por la clase Thread, que es una parte del paquete java.lang. Esta clase implementa una definición de threads independiente del sistema. Pero, bajo la campana, la implementación real de la operación concurrente la proporciona una implementación específica del sistema. Para la mayoría de las aplicaciones, la implementación básica no importa. Se puede ignorar la implementación básica y programar el API de los threads descrito en estas lecciones y en otra documentación proporcionada con el sistema JAVA.

- **Cuerpo del Thread**

Toda la acción tiene lugar en el cuerpo del thread — el método **run()**—. Se puede proporcionar el cuerpo de un Thread de una de estas dos formas: subclasificando la clase Thread y sobreescribiendo su método **run()**, o creando un thread con un objeto de la clase Runnable y su target.

- **Estado de un Thread**

A lo largo de su vida, un thread tiene uno o varios estados. El estado de un thread indica qué está haciendo el Thread y lo que es capaz de hacer durante su tiempo de vida: ¿se está ejecutando?, ¿está esperando? ¿o está muerto?

- **La prioridad de un Thread**

Una prioridad del Thread le dice al temporizador de threads de JAVA cuando se debe ejecutar este thread en relación con los otros.

- **Threads Daemon**

Estos threads son aquellos que proporcionan un servicio para otros threads del sistema. Cualquier thread JAVA puede ser un thread **daemon**.

- **Grupos de Threads**

Todos los threads pertenecen a un grupo. La clase ThreadGrpup, perteneciente al paquete java.lang define e implementa las capacidades de un grupo de thread relacionados.

— La mayoría de los ordenadores solo tienen una CPU, los threads deben compartir la CPU con otros threads. La ejecución de varios threads en un solo CPU, en cualquier orden, se llama programación. El sistema de ejecución JAVA soporta un algoritmo de programación determinístico que es conocido como programación de prioridad fija.

— A cada thread JAVA se le da una prioridad numérica entre MIN_PRIORITY y MAX_PRIORITY (constantes definidas en la clase Thread). En un momento dato, cuando varios threads están listos para ejecutarse, el thread con prioridad superior será el elegido para su ejecución. Solo cuando el thread para o se suspende por alguna razón, se empezará a ejecutar un thread con prioridad inferior.

— La programación de la CPU es totalmente preventiva. Si un thread con prioridad superior que el que se está ejecutando actualmente necesita ejecutarse, toma inmediatamente posesión del control sobre la CPU.

— El sistema de ejecución de JAVA no hace abandonar a un thread el control de la CPU por otro thread con la misma prioridad. En otras palabras, el sistema de ejecución de JAVA no comparte el tiempo. Sin embargo, algunos sistemas sí lo soportan, por lo que no se debe escribir código que esté relacionado con el tiempo compartido.

— Además, un thread cualquiera, en cualquier momento, puede ceder el control de la CPU llamando al método **yield().** Los threads solo pueden 'prestar' la CPU a otros threads con la misma prioridad que él –intentar cederle la CPU a un thread con prioridad inferior no tendrá ningún efecto–.

— Cuando todos los threads «ejecutables» del sistema tienen la misma prioridad, el programador elige a uno de ellos en una especie de orden de competición.

17.3. Programas con varios Threads

A) Sincronización de Threads

Frecuentemente, los threads necesitan compartir datos. Por ejemplo, supongamos que existe un thread que escribe datos en un fichero mientras, al mismo tiempo, otro está leyendo el mismo fichero. Cuando los threads comparten información necesitan sincronizarse para obtener los resultados deseados.

Existen muchas situaciones interesantes donde ejecutar threads concurrentes que compartan datos y deban considerar el estado y actividad de otros. Este conjunto de situaciones de programación son conocidos como escenarios 'productor/consumidor', donde el productor genera un canal de datos que es consumido por el consumidor.

Por ejemplo, se puede imaginar una aplicación JAVA donde un thread (el productor) escribe datos en un fichero mientras que un segundo thread (el consumidor) lee los datos del mismo fichero. O si se teclean caracteres en el teclado, el thread productor sitúa las pulsaciones en una pila de eventos y el consumidor lee los eventos de la misma pila. Estos dos ejemplos utilizan threads concurrentes que comparten un recurso común: el primero comparte un fichero y el segundo una pila de eventos. Como los threads comparten un recurso común, deben sincronizarse de alguna forma.

B) Imparcialidad, hambre y punto muerto

Si se escribe un programa en el que varios threads concurrentes deben competir por los recursos, se deben tomar las precauciones necesarias para asegurarse la justicia. Un sistema es justo cuando cada thread obtiene suficiente acceso a los recursos limitados como para tener un progreso razonable. Un sistema justo previene el hambre y el punto muerto. El hambre ocurre cuando uno o más threads de un programa están bloqueados por ganar el acceso a un recurso y así no pueden progresar. El punto muerto es la última forma de hambre; ocurre cuando dos o más threads están esperando una condición que no puede ser satisfecha. El punto muerto ocurre muy frecuentemente cuando dos (o más) threads están esperando a que el otro u otros hagan algo.

C) Volatile

Los programas pueden modificar variables miembros fuera de la protección de un método o un bloque sincronizados y puede declarar que la variable miembro es volatile.

Si una variable miembro es declarada como volatile, el sistema de ejecución JAVA utiliza esta información para asegurarse que la variable sea cargada desde la memoria antes de cada uso, y almacenada en la memoria después de utilizarla. Esto asegura que el valor de la variable es consistente y coherente a lo largo del programa.

D) Monitores

Los objetos, como el CubbyHole que son compartidos entre dos threads y cuyo acceso debe ser sincronizado son llamados condiciones variables. El lenguaje JAVA permite sincronizar threads alrededor de una condición variable mediante el uso de monitores. Los monitores previenen que dos threads accedan simultáneamente a la misma variable.

E) Los métodos notify() y wait()

En un nivel superior, el ejemplo Productor/Consumidor utiliza los métodos **notify()** y **wait()** del objeto para coordinar la actividad de los dos threads.

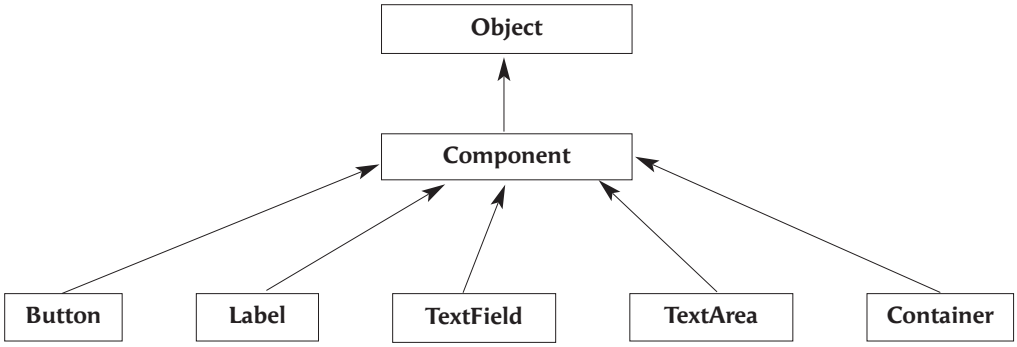
El objeto CubyHole utiliza notify() y wait() para asegurarse de que cada valor situado en él por el Productor es recuperado una vez y solo una por el Consumidor.

18. Interfaz gráfico AWT (Abstract Window Toolkit)

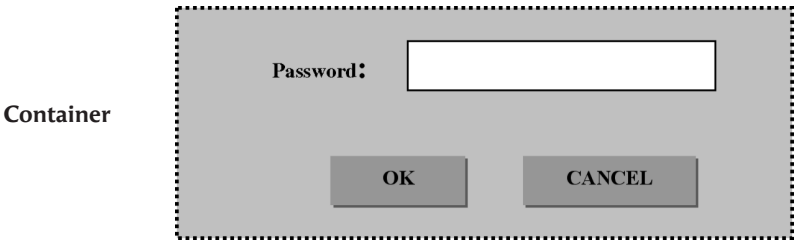
18.1. Introducción

El AWT (Abstract Windows Toolkit) es la parte de JAVA que se ocupa de construir interfaces gráficas de usuario. JAVA incluye una librería llamada **«Abstract Window Toolkit» (AWT)** que define los principales elementos para el desarrollo de una interfaz gráfica (GUI): **Button, Canvas, TextField, TextArea, Checkbox, Choice, Label, List, etc...**

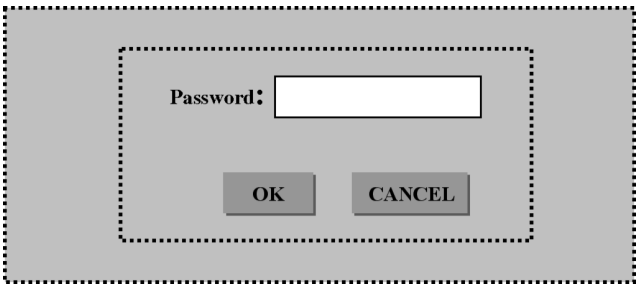
Todos estos **elementos** gráficos son objetos derivadas de una **superclase** común: **Component**.



Un objeto de la clase **Container** agrupa a los elementos gráficos. Para poder **mostrar** un **elemento** gráfico es necesario **añadirlo** a un objeto **Container**:



Un objeto **Container** es a su vez un **Component**, por lo que puede estar anidado en otro objeto **Container**:



Existen diferentes objetos de la clase **Container**, por lo general se utilizan objetos de la clase **Panel**.

18.2. Creación de una interfaz gráfica de usuario

Para construir una interfaz gráfica de usuario hace falta:

— Un contenedor o container, que es la ventana o parte de la ventana donde se situarán los componentes (botones, barras de desplazamiento, etc.) y donde se realizarán los dibujos. Se correspondería con un formulario o una picture box de Visual Basic.

— Los componentes: menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc. Se corresponderían con los controles de Visual Basic.

— El modelo de eventos. El usuario controla la aplicación actuando sobre los componentes, de ordinario con el ratón o con el teclado. Cada vez que el usuario realiza una determinada acción, se produce el evento correspondiente, que el sistema operativo transmite al AWT. El AWT crea un objeto de una determinada clase de evento, derivada de AWTEvent. Este evento es transmitido a un determinado método para que lo gestione.

En los siguientes apartados se verán con un cierto detalle estos tres aspectos del AWT. Hay que considerar que el AWT es una parte muy extensa y complicada de JAVA, sobre la que existen libros con muchos cientos de páginas.

18.3. Objetos event source y objetos event listener

El modelo de eventos de JAVA está basado en que los objetos sobre los que se producen los eventos (event sources) registran los objetos que habrán de gestionarlos (event listeners), para lo cual los event listeners habrán de disponer de los métodos adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los event listeners disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface Listener.

Las interfaces listener se corresponden con los tipos de eventos que se pueden producir.

Las capacidades gráficas del AWT resultan pobres y complicadas en comparación con lo que se puede conseguir con Visual Basic, pero tienen la ventaja de poder ser ejecutadas casi en cualquier ordenador y con cualquier sistema operativo.

18.4. Proceso a seguir para crear una aplicación interactiva (orientada a eventos)

Para avanzar un paso más, se resumen a continuación los pasos que se pueden seguir para construir una aplicación sencilla orientada a eventos, con interfaz gráfica de usuario:

— Determinar los componentes que van a constituir la interfaz de usuario (botones, cajas de texto, menús, etc.).

— Crear una clase para la aplicación que contenga la función main().

— Crear una clase Ventana, sub-clase de Frame, que responda al evento WindowClosing(). La función main() deberá crear un objeto de la clase Ventana (en el que se van a introducir las componentes seleccionadas) y mostrarla por pantalla con el tamaño y posición adecuados.

— Añadir al objeto Ventana todos los componentes y menús que deba contener. Se puede hacer en el constructor de la ventana o en el propio método main().

— Definir los objetos listener (objetos que se ocuparán de responder a los eventos, cuyas clases implementan las distintas interfaces listener) para cada uno de los eventos que deban estar soportados. En aplicaciones pequeñas, el propio objeto Ventana se puede ocupar de responder a los eventos de sus componentes. En programas más grandes se puede crear uno o más objetos de clases especiales para ocuparse de los eventos.

— Finalmente, se deben implementar los métodos de las interfaces Listener que se vayan a hacer cargo de la gestión de los eventos.

18.5. Relación entre componentes y eventos

En la siguiente tabla se relacionan los componentes del AWT y los eventos específicos de cada uno; hay que tener en cuenta que los eventos propios de una superclase de componentes pueden afectar a los componentes de una subclase.

COMPONENT	EVENTOS GENERADOS	SIGNIFICADO
Button	ActionEvent	Clicar en el botón
Checkbox	ItemEvent	Seleccionar o deseleccionar un ítem
CheckboxMenuItem	ItemEvent	Seleccionar o deseleccionar un ítem
Choice	ItemEvent	Seleccionar o deseleccionar un ítem
Component	ComponentEvent	Mover, cambiar tamaño, mostrar u ocultar un componente
	FocusEvent	Obtener o perder el focus
	KeyEvent	Pulsar o soltar una tecla
	MouseEvent	Pulsar o soltar un botón del ratón; entrar o salir de un componente; mover o arrastrar el ratón (tener en cuenta que este evento tiene dos Listener)
Container	ContainerEvent	Añadir o eliminar un componente de un "container"
List	ActionEvent	Hacer doble clic sobre un ítem de la lista
	ItemEvent	Seleccionar o deseleccionar un ítem de la lista
MunuItem	ActionEvent	Seleccionar un ítem de un menú
Scrollbar	AdjustementEvent	Cambiar el valor de la scrollbar
TextComponent	TextEvent	Cambiar el texto
TextField	ActionEvent	Terminar de editar un texto pulsando Intro
Window	WindowEvent	Acciones sobre una ventana: abrir, cerrar, iconizar, restablecer e iniciar el cierre

En la siguiente tabla se especifican también los eventos específicos de sus superclases:

AWT Components	Eventos que se pueden generar										
	Ation Event	Adjust. Event	Compn. Even	Contai. Even	Focus Even	Item Even	Key Even	Mouse Even	Mousm. Even	Text Even	Window Even
Button	*		*		*		*	*	*		
Canvas			*		*		*	*	*		
Checkbox			*		*	*	*	*	*		
Checkbox-MenuItem						*					
Choice			*		*	*	*	*	*		
Component			*		*		*	*	*		
Container			*	*	*		*	*	*		
Dialog			*	*	*		*	*	*		*
Frame			*	*	*		*	*	*		*
Label			*		*		*	*	*		
List	*		*		*	*	*	*	*		
MenuItem	*										
Panel			*	*	*		*	*	*		
Scrollbar		*	*		*		*	*	*		
TextArea			*		*		*	*	*	*	
TextField	*		*		*		*	*	*	*	
Window			*	*	*		*	*	*		*

18.6. Interfaces listener

Una vez vistos los distintos eventos que se pueden producir, conviene ver cómo se deben gestionar estos eventos. A continuación se detalla cómo se gestionan los eventos según el modelo de JAVA:

Cada objeto que puede recibir un evento (event source), registra uno o más objetos para que los gestionen (event listener). Esto se hace con un método que tiene la forma:

eventSourceObject.addEventListener(eventListenerObject);

Donde eventSourceObject es el objeto en el que se produce el evento, y eventListenerObject es el objeto que deberá gestionar los eventos. La relación entre ambos se establece a través de una interface Listener que la clase del eventListenerObject debe implementar. Esta interface proporciona la declaración de los métodos que serán llamados cuando se produzca el evento. La interface a implementar depende del tipo de evento.

EVENTO	INTERFACE LISTENER	MÉTODOS DE LISTENER
ActionEvent	ActionListener	actionPerformed()
AdjustementEvent	AdjustementListener	adjustementValueChanged()
ComponentEvent	ComponentListener	componentHidden(), componentMoved(), componentResized(), componentShown()
ContainerEvent	ContainerListener	componentAdded(), componentRemoved()
FocusEvent	FocusListener	focusGained(), focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed(), keyReleased(), keyTyped()
MouseEvent	MouseListener	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
	MouseMotionListener	mouseDragged(), mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated(), windowDeactivated(), windowClosed(), windowClosing(), windowIconified(), windowDeiconified(), windowOpened()

En la tabla están relacionados los distintos tipos de eventos, con la interface que se debe implementar para gestionarlos y los métodos declarados en cada interface.

Obsérvese que el nombre de la interface coincide con el nombre del evento, sustituyendo la palabra Event por Listener.

Una vez registrado el objeto que gestionará el evento, perteneciente a una clase que implemente la correspondiente interface Listener, se deben defi-

nir los métodos de dicha interface. Siempre hay que definir todos los métodos de la interface, aunque algunos de dichos métodos puedan estar vacíos.

18.7. Clases Adapter

JAVA proporciona ayudas para definir los métodos declarados en las interfaces Listener. Una de estas ayudas son las clases Adapter, que existen para cada una de las interfaces Listener que tienen más de un método. Su nombre se construye a partir del nombre de la interface, sustituyendo la palabra Listener por Adapter. Hay siete clases Adapter:

— ComponentAdapter.

— ContainerAdapter.

— FocusAdapter.

— KeyAdapter.

— MouseAdapter.

— MouseMotionAdapter.

— WindowAdapter.

Las clases Adapter derivan de Object, y son clases predefinidas que contienen definiciones vacías para todos los métodos de la interface. Para crear un objeto que responda al evento, en vez de crear una clase que implemente la interface listener, basta crear una clase que derive de la clase Adapter correspondiente, y redefina solo los métodos de interés.

19. Paseando por la red

Es muy sencillo acceder a archivos en la red utilizando JAVA. El paquete java.net dispone de varias clases e interfaces a tal efecto. En primer lugar, la clase URL nos permite definir un recurso en la red de varias maneras, por ejemplo:

URL url1 = new URL («http://www.rockar.com.ar/index.html»);

URL url2 = new URL («http», «www.rockar.com.ar», «sbits.htm»);

Por otra parte, podemos establecer una conexión a un URL dado mediante openConnection:

URLConnection conexion = url.openConnection();

Una vez lograda la conexión, podemos leer y escribir datos utilizando streams (corrientes de datos), como en el caso de manejo de archivos comunes (ver capítulo X). Un DataInputStream nos permite leer datos que llegan a través de la red, y un DataOutputStream nos permite enviar datos al host.

Por ejemplo:

DataInputStream datos = new DataInputStream (corrienteEntrada);

En nuestro caso, la corriente de entrada de datos proviene de la conexión al URL. El método getInputStream() del objeto URLConnection nos provee tal corriente:

DataInputStream datos = new DataInputStream(conex.getInputStream())

De este modo podemos escribir un pequeño programa para, por ejemplo, leer una página HTML desde una dirección arbitraria de internet. El programa, luego de compilarse mediante javac Lolo25.java, se ejecuta con java Lolo25 <url>; por ejemplo: java Lolo25 http://www.rockar.com.ar/index.html.

```
import java.io.*;          import java.net.*;
public class Lolo25 {
   public static void main(String argv[]) {
      String s;
      try {
         URL url = new URL (argv[0]);
         URLConnection conex = url.openConnection();
         System.out.println(«Cargando «+argv[0]);
         DataInputStream datos = new DataInputStream(conex.getInput  Stream());
         do {
            s = datos.readLine();
            if (s != null) System.out.println(s);
         } while (s != null);
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.out.println(«Sintaxis: java Lolo25 <url>»);
      }
      catch (UnknownHostException e) {
         System.out.println(«El host no existe o no responde»);
      }
      catch (Exception e) {e.printStackTrace();}
   }
}
```

Este programa muestra el HTML como texto en la pantalla, pero podríamos grabarlo a un archivo para guardarlo. Inclusive, podríamos procesarlo a medida que lo recibimos, identificar los tags <A HREF=url>, guardarlos en un vector, y seguidamente conectarnos y bajar los links que figuran en la página original hasta bajar un site completo.

Nótese que esto no solo sirve para establecer conexiones a páginas HTML. En realidad, un URL puede referirse también a otros protocolos, como gopher, ftp, etcétera; si bien según la implementación de JAVA puede haber problemas para conectarse a algunos tipos de URL.

20. Los sockets

Los sockets (zócalos, referido a los enchufes de conexión de cables) son mecanismos de comunicación entre programas a través de una red TCP/IP. De hecho, al establecer una conexión via Internet estamos utilizando sockets: los sockets realizan la interface entre la aplicación y el protocolo TCP/IP. Dichos mecanismos pueden tener lugar dentro de la misma máquina o a través de una red. Se usan en forma cliente-servidor: cuando un cliente y un servidor establecen una conexión, lo hacen a través de un socket. JAVA proporciona para esto las clases ServerSocket y Socket.

Los sockets tienen asociado un *port* (puerto). En general, las conexiones via Internet pueden establecer un puerto particular (por ejemplo, en http://www.rockar.com.ar:80/index.html, el puerto es el 80). Esto casi nunca se especifica porque ya hay definidos puertos por defecto para distintos protocolos: 20 para ftp-data, 21 para ftp, 79 para finger, etc. Algunos servers pueden definir otros puertos, e inclusive pueden utilizarse puertos disponibles para establecer conexiones especiales.

Justamente, una de las formas de crear un objeto de la clase URL permite especificar también el puerto:

URL url3 = new URL («http», «www.rockar.com.ar», 80,»sbits.htm»);

Para establecer una conexión a través de un socket, tenemos que programar por un lado el servidor y por otro los clientes.

En el servidor creamos un objeto de la clase ServerSocket y luego esperamos algún cliente (de clase Socket) mediante el método accept():

ServerSocket conexion = new ServerSocket(5000); // 5000 es el puerto en este caso

Socket cliente = conexion.accept(); // espero al cliente

Desde el punto de vista del cliente, necesitamos un socket al que le indiquemos la dirección del servidor y el número de puerto a usar:

Socket conexion = new Socket (direccion, 5000);

Una vez establecida la conexión, podemos intercambiar datos usando streams como en el ejemplo anterior. Como la clase URLConnection, la clase Socket dispone de métodos getInputStream y getOutputStream que nos dan respectivamente un InputStream y un OutputStream a través de los cuales transferir los datos.

21. El JAVA Development Kit

El kit de desarrollo de JAVA consiste en un compilador y herramientas de desarrollo para crear tanto programas independientes como Applets. Desarrollado por Sun Microsystems, los creadores de JAVA. Esencial para todo programador de JAVA. El entorno básico del JDK de JAVA que proporciona Sun está formado por herramientas en modo texto, que son: *java*, intérprete que ejecuta programas en byte-code. *javac*, compilador de JAVA que convierte el código fuente en byte-code. *javah*, crea ficheros de cabecera para implementar métodos para cualquier clase. javap, es un descompilador de byte-code a código fuente JAVA. *javadoc*, es un generador automático de documentos HTML a partir del código fuente JAVA. *javaprof*, es un profiler para aplicaciones de un solo thread. *HotJAVA*, es un navegador Web escrito completamente en JAVA.

El entorno habitual, pues, consiste en un navegador que pueda ejecutar Applets, un compilador que convierta el código fuente JAVA a byte-code y el intérprete JAVA para ejecutar los programas. Estos son los componenetes básicos para desarrollar algo en JAVA. No obstante se necesita un editor para escribir el código fuente, y no son estrictamente necesarias otras herramientas como el debugger, un entorno visual, la documentación o un visualizador de jerarquía de clases.

JDK es el acrónimo de «JAVA Development Kit», es decir Kit de desarrollo de JAVA. Se puede definir como un conjunto de herramientas, utilidades, documentación y ejemplos para desarrollar aplicaciones JAVA.

21.1. Componentes del JDK

JDK consta de una serie de aplicaciones y componentes, para realizar cada una de las tareas de las que es capaz de encargarse.

— Intérprete en tiempo de ejecución (JRE).

— Compilador.

— Visualizador de applets.

— Depurador.

— Desensamblador de archivo de clase.

— Generador de cabecera y archivo de apéndice.

— Generador de documentación.

— Applets de demostración.

— Código fuente la API.