Curso de Programación JAVA

Fco. Javier Alcalá Casado

Contenidos

Bibliografía

- *Java in a nutshell: a desktop quick reference*
  D. Flanagan. Ed. O'Reilly
  Muy completo
- *The Java tutorial: object-oriented programming for the Internet*
  M. Campione. Ed. Addison-Wesly
  Programación del lenguaje
- *Core packages*.
  J. Gosling. Ed. Addison-Wesley
  Manual de referencia
- *The Java language specification*
  J. Gosling. Ed. Addison-Wesley
  Lenguaje y manual de referencia
- *http://java.sun.com/j2se/1.4/docs/api/index.html*
  Referencia actualizada por SUN Microsystems

Introducción

Fco. Javier Alcalá Casado

Introducción (I)

- Creado en 1991 por Sun Microsystems para electrodomésticos:
  - Escasa potencia de cálculo
  - Poca memoria
  - Distintas CPUs
- Consecuencias:
  - Lenguaje sencillo que genera código reducido
  - Código neutro independiente de la CPU (máquina virtual)
- Lenguaje de programación para ordenadores desde 1995

Introducción (II)

- Sun describe Java como un lenguaje "*simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico*"
- Similar en sintaxis a C/C++ y en semántica a SmallTalk
- Ejecución de Java como:
  - aplicación independiente
  - *applet* (dentro del navegador al cargar la página *web*)
  - *servlet* (ejecutado en servidor de Internet, sin interfaz gráfica)
- JDK (*Java Development Kit*): programas y librerías para desarrollar, compilar y ejecutar programas Java

Características de Java

- Lenguaje de fácil uso orientado a objetos
- Lenguaje compilado e interpretado
- Facilita un entorno interpretado:
  - Velocidad de desarrollo (no de ejecución)
  - Portabilidad del código
- Ejecución multitarea
- Cambios dinámicos en tiempo de ejecución
- Seguridad del código

Máquina Virtual Java (JVM)

- La *Java Virtual Machine* es una máquina hipotética que emula por software a una máquina real. Contiene:
  - Conjunto de instrucciones máquina (C.O. + Operandos)
  - Registros
  - Pila
  - Memoria
  - ...
- El compilador genera *bytecodes* (instrucciones de código máquina para JVM)
- El intérprete ejecuta y traduce los *bytecodes* para cada máquina específica

Compilador e Intérprete de Java

- El **compilador** analiza la sintaxis del código fuente (con extensión `*.java`). Si no hay errores, genera *bytecodes*

  `> javac Nombre.java ⇒ Nombre.class`

- El **intérprete** es la Máquina Virtual Java que ejecuta los *bytecodes* (con extensión `*.class`) creados por el compilador

  `> java Nombre` (sin extensión .class)

- Aplicación con argumentos:

  `> java Nombre arg1 arg2` ...

Garbage Collector

- Debe liberarse la memoria reservada dinámicamente que no se vaya a utilizar más
- En otros lenguajes, esta liberación debe realizarla el propio programador
- La JVM dispone de un *thread* que rastrea las operaciones de memoria: el *Garbage Collector*, el cual:
  - Verifica y libera la memoria que no se necesita
  - Se ejecuta automáticamente
  - Puede variar según la implementación de la JVM

Seguridad del Código

- La JVM verifica los *bytecodes* asegurando que:
  - el código se ajusta a las especificaciones de la JVM
  - no hay violaciones de acceso restringido
  - el código no provoca desbordamientos de la pila
  - los tipos de los parámetros son correctos para todo el código
  - no existen conversiones ilegales de datos (p.e. convertir de enteros a punteros)
  - los accesos a los campos de los objetos están autorizados

Variables de entorno

- En versiones antiguas del JDK, es necesario incluir las siguientes líneas al final del *autoexec.bat*
- Para tener accesibles el compilador y el intérprete Java:

```
set PATH=%PATH%;C:\jdk1.2.2\bin
```

(el directorio dependerá de dónde se hayan instalado las JDK)

- Para acceder a las clases desarrolladas por uno mismo:

```
set CLASSPATH=.;%CLASSPATH%
```

Formato de los Ficheros Fuente

⇨ El fichero fuente contiene 3 elementos principales:

- Declaración de paquete (opcional)
- Sentencias de importación (opcional)
- Declaración de clase o de interfaz

Ejemplo: fichero fuente **`Empleado.java`**

```
package abc.financedept;
import java.lang.*;
import java.awt.*;
public class Empleado
{
      ...
}
```

"¡Hola Mundo!"

Fichero `HolaMundo.java`:

```
1   //
2   // Aplicación ejemplo HolaMundo
3   //
4   public class HolaMundo
5   {
6         public static void main (String args[])
7         {
8               System.out.println("¡Hola Mundo!");
9         }
10  }
```

```
> javac HolaMundo.java

> java HolaMundo
¡Hola Mundo!
```

Características de Java

Fco. Javier Alcalá Casado

Características del Lenguaje

- ⇨ Sensible a mayúsculas/minúsculas
- ⇨ Soporta comentarios
- ⇨ Lenguaje de formato libre
- ⇨ Permite identificadores
- ⇨ Incluye palabras reservadas
- ⇨ Permite variables y constantes
- ⇨ Convenciones de nomenclatura
- ⇨ Tiene reglas sobre los tipos de datos

Sensible a Mayúsculas/Minúsculas

- Se distingue entre mayúsculas y minúsculas
- Los identificadores Cat, cat y CAT son diferentes
- Todas las palabras reservadas del lenguaje van en minúsculas

Soporta Comentarios

- Existen tres formas de introducir comentarios:
  - Comentario en **una línea**

```
// Comentario de una línea
```

  - Comentario en **una o más líneas**

```
/* Comentario de
   más de una línea */
```

  - Comentario de **documentación**. Se usa con *javadoc*

```
/** Método XYZ:
    Realiza la labor X sobre los datos Y
    devolviendo Z */
```

```
> javadoc Fichero.java ⇒ Fichero.html
```

Lenguaje de Formato Libre

⇨ La disposición de los elementos dentro del código es libre

⇨ **Sentencias**: línea simple de código terminada en **;**

```
total = a + b + c + d ;
```

⇨ **Bloque de código**: conjunto de sentencias agrupadas entre llaves

```
{
   x=x+1;
   y=y+1;
}
```

⇨ Java permite espacios en blanco entre elementos del código

```
x1 =  y    * delta ;
x2 = (y+1) * delta ;
```

Identificadores

⇨ Son nombres de clases, variables o métodos

⇨ No tienen longitud máxima

⇨ El primer carácter del identificador debe ser: A-Z, a-z, _, $

⇨ El resto: A-Z, a-z, _, $, 0-9

⇨ No se permiten vocales acentuadas ni la letra eñe (ñ, Ñ)

⇨ No se permite utilizar palabras reservadas como identificador

Palabras Reservadas

⇨ Palabras con un significado especial para el compilador:

abstract	default	goto[1]	null[2]	synchronized
boolean	do	if	package	this
break	double	implements	private	throw
byte	else	import	protected	throws
case	extends	instanceof	public	transient
catch	false[2]	int	return	true[2]
char	final	interface	short	try
class	finally	long	static	void
const[1]	float	native	super	volatile
continue	for	new	switch	while

[1] palabras no usadas por el lenguaje, pero son reservadas
[2] realmente son constantes del lenguaje

Variables y Constantes

⇨ **Variables**: zona de memoria cuyos valores van a cambiar durante la ejecución
Declaración: `<tipo> <identificador> ;` ó

`<tipo> <identificador> , <identificador> ... ;`

Ejemplo: `int x, y, z ;`

⇨ **Constantes:** zona de memoria cuyos valores no cambian
Declaración:

```
final <tipo> <identificador> = <valor> ;
```

Ejemplo: `final double PI = 3.14159265 ;`

Asignación de Variables

- Se utiliza el operador asignación **=**

```
<variable> = <expresión> ;
```

- La parte izquierda siempre debe ser una variable
- La parte derecha puede ser un literal, una variable, una expresión, una función o una combinación de todos
- Se puede asignar un valor a una variable en el momento de declararla

  Ejemplo: `int i = 0 ;`

Convenciones de Nomenclatura (I)

- Los identificadores que proporciona Java siguen una convención según el elemento:
  - **Clases**: primera letra en mayúscula de cada palabra

    Ejemplo: `Empleado`, `LibroDeCuentas`, `String`
  - **Variables**: primera letra en minúscula y la primera letra de cada palabra en mayúscula

    Ejemplo: `contador`, `numeroTotalAccesos`, `string`
  - **Constantes**: todo en mayúsculas, separando cada palabra por el carácter "_"

    Ejemplo: `PI`, `ANCHO_IMAGEN`

Convenciones de Nomenclatura (II)

- **Métodos**: siguen el mismo formato que las variables seguidas de paréntesis “(“ “)”

  Ejemplo: `sumar()`, `obtenerResultado()`

- **Estructuras de control**: utilizar llaves englobando a todas las sentencias de una estructura de control, aunque sólo haya una sentencia

  Ejemplo:

```
if ( <condición> )
{
      // hacer algo
}
else
{
      // hacer otra cosa
}
```

Tipos de Datos y Operadores

Fco. Javier Alcalá Casado

Tipos de Datos

- Java define dos tipos de datos:
  - Tipos primitivos
  - Tipos referencia
- Los **tipos primitivos** son ocho agrupados en cuatro categorías:
  - lógico: boolean
  - texto: char
  - entero: byte, short, int, long
  - real: float, double
- Los **tipos referencia** son punteros a objetos

Tipo de Datos Lógico

- El tipo de datos `boolean` (8 bits) puede tomar dos valores posibles: `true` y `false`
- El tipo `boolean` no toma valores numéricos
- En Java *no* se considera cero como falso y distinto de cero como verdadero (como sucede en C/C++)
- No existe conversión entre tipos enteros y tipos lógicos

```
int i = 10 ;
if ( i )
{ ... }
```

Error de compilación

```
int i = 10 ;
if ( i != 0 )
{ ... }
```

Correcto

Tipo de Datos de Texto

- El tipo `char` (16 bits) representa sólo un carácter Unicode
- El código universal Unicode incluye el código ASCII y comprende los caracteres gráficos de prácticamente todos los idiomas (japonés, chino, braille...)
- El literal de texto debe ir entre comillas simples ‘ ’
- Utiliza la siguiente notación:
  - caracteres simples: ‘a’
  - caracteres especiales: ‘\t’, ‘\n’
  - caracteres Unicode (con 4 dígitos en hexadecimal): ‘\u00BF’

Tipo de Datos Entero

- Existen cuatro tipos de datos enteros: `byte` (8 bits), `short` (16 bits), `int` (32 bits) y `long` (64 bits)
- Todos los tipos tienen signo. El cero se considera positivo
- Los literales enteros se pueden representar con notación:
  - decimal: 2, 156, 56453645
  - octal: 077, 07700 (empezando con un cero)
  - hexadecimal: 0xABFF, 0xCC00 (empezando con 0x)
- Por defecto siempre se consideran de tipo `int`
- Seguido de L se considera `long`: 156L, 077L, 0xABFFL

Tipo de Datos Real

- Existen dos tipos de datos reales: `float` (32 bits) y `double` (64 bits)
- Un literal es de punto flotante si lleva:
  - un punto decimal: 3.14159, 2.0
  - una E ó e (valor exponencial): 105e25, 1.05E27
  - una F ó f (`float`): 279F, 2.79f
  - una D ó d (`double`): 279D, 2.79d
- Un literal real por defecto siempre se considera de tipo `double`, si no se indica explícitamente que es un `float`

Resumen de Tipos Primitivos

- El tamaño de cada tipo de dato primitivo se mantiene invariable, independientemente de la arquitectura de la máquina
- El valor por defecto se toma para las variables no inicializadas de los objetos

Tipo	Contiene	Valor por defecto	Tamaño	Rango de valores	
				Min	Max
`boolean`	True o false	false	8 bits	-	-
`char`	Carácter Unicode	\u0000	16 bits	\u0000	\uFFFF
`byte`	Entero con signo	0	8 bits	-128	127
`short`	Entero con signo	0	16 bits	-32768	32768
`int`	Entero con signo	0	32 bits	-2147483648	2147483647
`long`	Entero con signo	0	64 bits	-9223372036854775808	9223372036854775808
`float`	IEEE 754 estándar punto flotante	0.0	32 bits	±3.40282347E+38	±1.40239846E-45
`double`	IEEE 754 estándar punto flotante	0.0	64 bits	±1.79769313486231570E+308	±4.94065645841246544E-324

Tipo de Datos Referencia

- Un tipo referencia guarda un puntero a la dirección donde se ubica el objeto (32 bits)
- Sólo puede almacenar direcciones de objetos de su propio tipo

  Ejemplo:
  ```
  Ordenador pc , sun ;
  Usuario user ;
  pc = new Ordenador ( ) ;
  user = pc ;      ⇒ Error de compilación
  sun = pc ;       ⇒ Correcto
  ```

- Todas las clases son de tipo referencia
- El valor que toma por defecto una variable de tipo referencia es `null`

Cadenas de Caracteres

- La clase `String` permite manejar cadenas de caracteres
- El literal `String` debe ir entre comillas dobles " "
- Se puede crear una cadena de caracteres de dos formas:

  ```
  String nombre = new String("Pepe");
  String nombre = "Pepe";
  ```

- "a" ≠ 'a'
- Para concatenar dos cadenas se utiliza el operador **+**

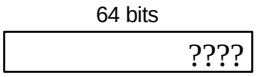
  "Pepe" + "Pérez" ⇒ "PepePérez"

- No se guarda el carácter de fin de cadena

Memoria Asignada a una Variable

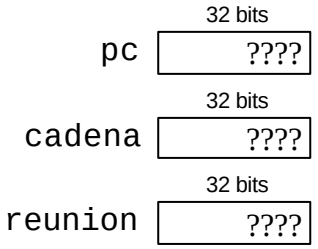
⇨ Tipo **primitivo**: se asigna la cantidad de memoria que requiere el tipo de la variable

Ejemplo: `long x ;`



⇨ Tipo **referencia**: se asigna el espacio correspondiente a una dirección de memoria (32 bits)

Ejemplo:
```
Computer pc ;
String cadena ;
Fecha reunion ;
```



Conversión de Tipos

⇨ La conversión de tipos (*casting*) se debe realizar entre tipos de la misma naturaleza: numéricos o referencia

⇨ Al convertir un tipo a un tamaño más pequeño se puede perder la información de los bits de mayor peso

⇨ La sintaxis es: `(<tipo>) <expresión>`

Ejemplo:
```
byte num8bits = (byte) 27 ;
int num32bits = 27 ;
num8bits = (byte) num32bits ;
```

Ejemplo:
```
short a , b , c ;
c = a + b ;
```
⇒ Error, + devuelve `int`

```
c = (short)(b + c) ;
```
⇒ Correcto

Operadores Java (I)

- Operadores **unarios**: +, -
- Operadores **aritméticos**: +, -, *, /, % (resto de la división)
- Operadores **de asignación**: =, +=, -=, *=, /=, %=
  `<var> += <expr>` ⇒ `<var> = <var> + <expr>`
- Operadores **incrementales**: ++, --
  - precediendo a la variable: `++<var>`, `--<var>`
  - siguiendo a la variable: `<var>++`, `<var>--`

```
i=6;          i=6;
j=i++;   ⇒    j=i;
              i=i+1;
 i=7 , j=6
```

```
i=6;          i=6;
j=++i;   ⇒    i=i+1;
              j=i;
        i=7 , j=7
```

Operadores Java (II)

- Operadores **relacionales**: == (igual), != (distinto), >, <, >=, <=
- Operadores **lógicos**: && (AND), || (OR), ! (NOT), & (AND), | (OR)
  - && y || realizan evaluación perezosa:
    - `op1 && op2` ⇒ si `op1` es `false`, no se evalúa `op2`
    - `op1 || op2` ⇒ si `op1` es `true`, no se evalúa `op2`
  - & y | siempre evalúan los dos operadores
- Operador **instanceof**: `<objeto> instanceof <clase>` determina si un objeto pertenece a una clase

Operadores Java (III)

⇨ Operador **condicional**: ? :
`<exprBooleana> ? <valor1> : <valor2>`
Permite bifurcaciones condicionales sencillas

⇨ Operadores **a nivel de bits**: >>, <<, >>>, &, |, ^, ~

- `op1 >> n` desplaza los bits de `op1` (con signo) a la derecha n posiciones
- `op1 << n` desplaza los bits de `op1` (con signo) a la izquierda n posiciones
- `op1 >>> n` desplaza los bits de `op1` (**sin** signo) a la derecha n posiciones
- `op1 & op2` Operador AND a nivel de bits
- `op1 | op2` Operador OR a nivel de bits
- `op1 ^ op2` Operador XOR a nivel de bits
- `~op1` Operador complemento (NOT a nivel de bits)

Precedencia de Operadores

⇨ Todos los operadores binarios se evalúan *de izquierda a derecha*, excepto los operadores de asignación

Tipo	**Operador**
Operadores sufijos	[] . (argumentos) expr++ expr--
Operadores unarios	++expr –expr +expr –expr ~ !
Creación y *casting*	new (tipo)expr
Multiplicativos	* / %
Aditivos	+ -
Desplazamiento	<< >> >>>
Relacional	< > <= >= instanceof
Igualdad	== !=
AND (bits)	&
OR exclusivo (bits)	^
OR inclusivo (bits)	|
AND lógico	&&
OR lógico	||
Condicional	? :
Asignación	= += -= *= /= %= **&= ^= |= <<= >>= >>>=**

Control de Flujo

Fco. Javier Alcalá Casado

Control de Flujo

- Las sentencias de control del flujo de ejecución permiten tomar decisiones y realizar un proceso repetidas veces
- Hay dos tipos principales de sentencias de control de flujo:
  - Condicionales: `if`, `switch`
  - Bucles: `for`, `while`, `do while`
- Otras sentencias que permiten interrumpir el flujo normal de ejecución son `break` y `continue`

Sentencia if

⇨ Ejecuta un conjunto de sentencias en función del valor de la expresión de comparación (booleana)

⇨
```
if ( <exprBooleana> )
    <sentencia> ;
```

⇨
```
if ( <exprBooleana> )
{
    <grupoSentencias1> ;
}
else
{
    <grupoSentencias2> ;
}
```

⇨
```
if ( <exprBooleana> )
{
    <grupoSentencias> ;
}
```

⇨
```
if ( <exprBooleana1> )
{
    <grupoSentencias1> ;
}
else if ( <exprBoolean2> )
{
    <grupoSentencias2> ;
}
else
{
    <grupoSentencias3> ;
}
```

Sentencia switch

⇨ Comparación de igualdad múltiple con la misma variable

```
switch ( <variable> )
{
   case literal1: [<grupoSentencias1> ;]
                  [break;]
   case literal2: [<grupoSentencias2> ;]
                  [break;]
        ...
   case literalN: [<grupoSentenciasN> ;]
                  [break;]
   [default: <grupoSentencias> ;]
}
```

Ejemplo switch

Opciones de un menú:

1.- Abrir Fichero
2.- Cerrar Fichero
3.- Cerrar Fichero e Imprimir Datos
4.- Imprimir Datos
5.- Salir

```
switch ( opcion )
{
    case 1:     abrirFich ( ) ;
                break ;
    case 2:     cerrarFich ( ) ;
                break ;
    case 3:     cerrarFich ( ) ;
    case 4:     imprimirDatos ( ) ;
                break ;
    case 5:
    default:    terminarPrograma ( ) ;
}
```

Sentencia for

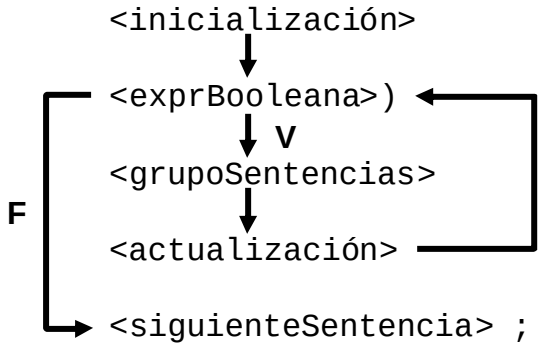
⇨ Permite la ejecución repetida de un grupo de sentencias con mayor control

```
for ( <inicial>; <exprBooleana>; <actual> )
{
   <grupoSentencias> ;
}
```

- <inicialización> asignación del valor inicial de las variables que intervienen en la expresión
- <exprBooleana> condición booleana
- <actualización> nuevo valor de las variables
- en <inicialización> y en <actualización> pueden ir más de una asignación separadas por comas

Funcionamiento de la sentencia `for`

⇨ Las partes del `for` siguen el siguiente orden de ejecución:



Ejemplo:

```
for ( i=0 , j=100 ; i < j ; i++ , j -= 3 )
      System.out.println ( i + “,” + j ) ;
```

Sentencia `while`

⇨ El grupo de sentencias se ejecuta mientras se cumpla la expresión booleana

```
while ( <exprBooleana> )
{
     <grupoSentencias> ;
}
```

Equivalencia for - while

```
for ( <inicial>; <exprBooleana>; <actual> )
{
   <grupoSentencias> ;
}
```

equivale a

```
<inicialización> ;
while ( <exprBooleana> )
{
   <grupoSentencias> ;
   <actualización> ;
}
```

Sentencia do while

⇨ El grupo de sentencias se ejecuta mientras se cumpla la expresión booleana

```
do
{
     <grupoSentencias> ;
}
while ( <exprBooleana> ) ;
```

⇨ El grupo de sentencias se ejecuta al menos 1 vez

Sentencias `break` y `continue`

- La sentencia `break` provoca la terminación inmediata de un bucle o sentencia `switch` (sin ejecutar el resto de sentencias)

  Válido para `for`, `while`, `do while` y `switch`

- La sentencia `continue` provoca la terminación inmediata de una iteración de un bucle

  Válido para `for`, `while` y `do while`

Programación Orientada a Objetos (POO)

Fco. Javier Alcalá Casado

Paradigmas de Programación

- **Paradigma estructurado o procedural**: los programas se dividen en *procedimientos* independientes con acceso total a los datos comunes

Algoritmos + Estructuras de Datos = Programas

- **Paradigma funcional**: el resultado de un cálculo es la entrada del siguiente, así hasta que se produce el valor deseado
- **Paradigma orientado a objetos**: los *datos* se consideran la parte más importante del programa, de modo que se agrupan en objetos.
  Los objetos modelan las características de los problemas del mundo real, su comportamiento ante estas características y su forma de interactuar con otros elementos

Objetos + Mensajes = Programas

Ejemplo

- Tomarse un café en una cafetería:

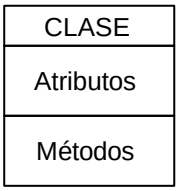
- Procedural:
  - el cliente entra en la cafetería
  - el cliente pasa detrás de la barra
  - el cliente prepara la cafetera
  - el cliente se sirve el café
  - el cliente se bebe el café

- Orientado a objetos:
  - el cliente entra en la cafetería
  - el cliente pide un café al camarero
  - el camarero prepara la cafetera
  - la cafetera hace el café
  - el camarero sirve el café al cliente
  - el cliente se bebe el café

Conceptos de la Orientación a Objetos

- **Clases**: patrones que indican cómo construir los objetos
- **Objetos**: instancias de las clases en tiempo de ejecución

  Ejemplo: plano de arquitecto *vs* edificios



- **Miembros** de la clase:
  - **Atributos**: características o propiedades de los objetos. Pueden ser variables numéricas o referencias a otros objetos
  - **Métodos**: comportamiento de los objetos. Son funciones que operan sobre los atributos de los objetos

Características de la Orientación a Objetos

- Cada objeto tiene características reconocibles
  Ejemplo: un *Empleado* tiene *Nombre*, *DNI*, *Salario*...
- Cada objeto es único
  Ejemplo: el *Empleado1* es *Pepe Pérez* con *DNI 12345678* cobra *18.000 €*
- El código fuente orientado a objetos define clases
- En tiempo de ejecución, el programa crea objetos a partir de cada clase
- Los objetos almacenan información
- Los objetos realizan operaciones sobre sus atributos

Mínimo Programa Orientado a Objetos

Fichero fuente `MinProgOO.java`:

```
public class MinProgOO
{
    public static void main (String args[])
    {
          Objeto obj = new Objeto();
          obj.saluda();
    }
}
```

Fichero fuente `Objeto.java`:

```
public class Objeto
{
    public void saluda()
    {
          System.out.println("¡Hola Mundo!");
    }
}
```

Definición de Clase

⇨ Sintaxis:

```
class <NombreClase>
{
    // Declaración de atributos
    <tipo> <variable> ;

    // Declaración de métodos
    <tipo> <nombreMétodo> ( <argumentos> )
    { ... }
}
```

⇨ El nombre del fichero Java debe coincidir con el de la clase definida en él ⇒ `<NombreClase>.java`

⇨ Se recomienda definir una clase por cada fichero Java

Ejemplo de Clase

⇨ Clase que almacena una fecha ⇒ `Fecha.java`

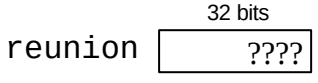
```
class Fecha
{
// Atributos
   int dia ;
   int mes ;
   int anno ;
// Métodos
   void asignarDia ( int d ) {...}
   String darFormato ( ) {...}
}
```

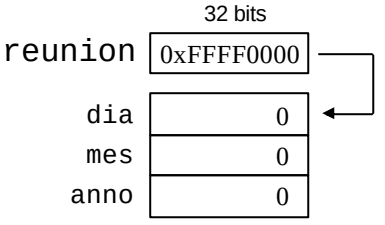
Creación de un Objeto

⇨ Se utiliza la palabra reservada `new`

```
<refObjeto> = new <NombreClase>() ;
```

⇨ Ejemplo:
```
Fecha reunion ;
reunion = new Fecha ( ) ;
```





Acceso a los Miembros de un Objeto

⇨ A través del operador **punto** (**.**) se puede acceder tanto a los atributos como a los métodos

```
<refObjeto>.<atributo> ó <refObjeto>.<método>()
```

Ejemplo:

```
Fecha reunion = new Fecha ( ) ;
reunion.dia = 15 ;
reunion.mes = 12 ;
reunion.anno = 2000 ;
reunion.darFormato() ;
```

Métodos

⇨ Los métodos son bloques de código (subprogramas) definidos dentro de una clase

⇨ Un método tiene acceso a **todos** los atributos de su clase

⇨ Pueden ser llamados o invocados desde cualquier sitio

⇨ Un método puede invocar a otros métodos

⇨ Los métodos que se invocan a sí mismos son *recursivos*

⇨ En Java no se puede definir un método dentro de otro

⇨ La ejecución de todos los programas se inician con el método `main`

Definición de Métodos (I)

```
<tipoRetorno> <nombreMétodo> ( <listaArgumentos> )
{
    <bloqueCódigo>
}
```

⇨ `<tipoRetorno>`: tipo de dato que retorna el método (primitivo o referencia)
Si no devuelve ningún valor, debe ser `void`

⇨ `<nombreMétodo>`: identificador del método

⇨ `<listaArgumentos>`: el método admite que le pasen argumentos separados por comas con el formato:

```
[<tipo> <argumento> [, <tipo> <argumento>...]]
```

Definición de Métodos (II)

⇨ `<bloqueCódigo>`: conjunto de sentencias que implementan la tarea que debe realizar el método

Si devuelve un valor, debe utilizar la sentencia `return`

```
return <valor> ;
```

`<valor>` debe ser del mismo `<tipoRetorno>` con que se ha declarado el método

El código se ejecuta hasta alcanzar la sentencia `return` (si devuelve un valor) o hasta el final del método (si no devuelve nada)

Se pueden declarar variables locales si son necesarias

Ejemplos de Métodos

```
⇨ double tangente ( double x )
  {
      return Math.sin(x) / Math.cos(x) ;
  }

⇨ void imprimirHola ( )
  {
      System.out.println ( “Hola” ) ;
  }

⇨ String darFormato ( int dia , int mes , int anno )
  {
      String s ;

      s = dia + “/” + mes + “/” + anno ;
      return s ;
  }
```

Paso de Argumentos

⇨ Java sólo permite pasar argumentos por valor

⇨ El método recibe una copia de los argumentos

⇨ El valor de los argumentos de tipo *primitivo* no cambia fuera del método

⇨ El valor de los argumentos de tipo *referencia* (un puntero a un objeto) tampoco cambia fuera del método, pero el contenido del objeto referenciado sí se puede cambiar dentro del método

⇨ Tipo *primitivo* ⇒ paso por valor
Tipo *referencia* ⇒ paso por referencia

Ámbito de las Variables (I)

- En Java se dispone de tres tipos de variables:
  - Variables miembro pertenecientes a una clase
  - Argumentos de un método de la clase
  - Variables locales de un método de la clase
- Los argumentos trabajan como variables locales

```
class Ejemplo
{
    int x ;                  // variable miembro
    void metodo ( int y )    // argumento
    {
          int z ;            // variable local

          x = y + z ;
    }
}
```

Ámbito de las Variables (II)

- Las variables miembro son visibles desde cualquier parte de la clase
- Los argumentos y variables locales sólo son visibles dentro del método al que pertenecen. Dejan de existir al finalizar el método
- Dentro de un método, si coincide el identificador de un argumento o variable local con el de una variable miembro, sólo se accede a la variable del método

```
class A
{
    int x ;
    void metodo ( int y )
    {
        int x = 2 ;
        y = 3*x + y - x ;
        ...println(y);  þ 8
    }
}
```

```
... main(...)
{
  int arg = 4 ;
  A obj = new A();
  obj.x = 1 ;
  obj.metodo(arg);
  ...println(arg);    þ 4
  ...println(obj.x);  þ 1
}
```

El puntero this

- Se emplea para apuntar al objeto actual dentro de un método
- Con `this` se hace accesible una variable miembro cuyo identificador coincide con una variable local

```
class A
{
    int x ;
    void metodo ( int y )
    {
       int x = 2 ;
       y = 3*this.x + y - x ;
       ...println(y);  Þ 5
    }
}
```

```
... main(...)
{
  int arg = 4 ;
  A obj = new A();
  obj.x = 1 ;
  obj.metodo(arg);
  ...println(arg);   Þ 4
  ...println(obj.x); Þ 1
}
```

Sobrecarga de Métodos

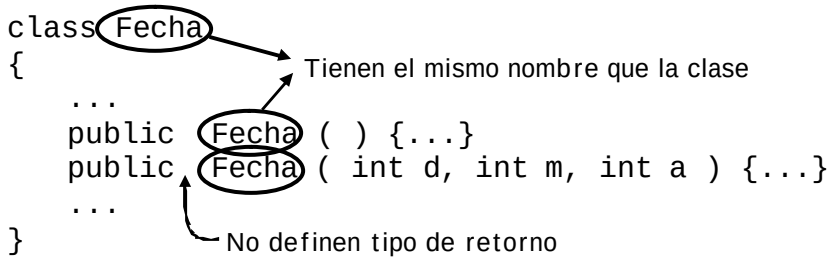
- A veces se necesitan varios métodos que hagan la misma tarea pero con argumentos de tipos distintos
- Java permite utilizar un mismo nombre para diferentes métodos, siempre que se pueda identificar cada método
- Un método se identifica por su nombre, el tipo de retorno, el número de argumentos que tiene y el tipo de cada uno de ellos

```
void mostrarInt(int i)        void mostrar(int i)
void mostrarLong(long l)      void mostrar(long l)
void mostrarFloat(float f)    void mostrar(float f)
```

- Esta característica se llama *sobrecarga de métodos*

Constructores (I)

⇨ Un *constructor* es un tipo especial de método que permite construir un objeto de una clase
Ejemplo:



⇨ Se utilizan junto con la palabra reservada `new`

```
Fecha f = new Fecha ( 10, 12, 2000 ) ;
```

Constructores (II)

⇨ Los constructores se pueden sobrecargar y son opcionales

⇨ Si no se define ningún constructor, Java proporciona uno por defecto. Incorpora el siguiente código a la clase:

```
class <NombreClase>
{
    ...
    public <NombreClase> ( ) { }
    ...
}
```

⇨ Si se define un constructor con argumentos, se pierde el constructor por defecto

⇨ Normalmente, en el constructor se inicializan las variables miembro

Destructores

- En Java no hay destructores de objetos como en C++
- El *garbage collector* es el encargado de liberar la memoria:
  - Cuando detecta objetos no referenciados
  - Al final de un bloque que haya utilizado objetos

Arrays

Fco. Javier Alcalá Casado

Arrays

- Los *arrays* son estructuras de memoria que almacenan en una variable múltiples valores del *mismo tipo*
- Los *arrays* son **objetos** ⇒ se crean con `new`
- Se utilizan los corchetes, `[]`, para declarar el *array* y para acceder a sus elementos
- Pueden ser de cualquier tipo (primitivo o referencia)
- Declaración de *arrays*:

```
<tipo> <variable>[];  ó  <tipo>[] <variable>;
int a[];  equivale a  int[] a;
```

`int a[], b, c;` (a es un *array*; b y c son enteros)
`int[] a, b, c;` (a, b y c son *arrays*) ⇒ RECOMENDADO

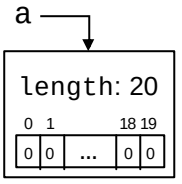
Instanciación de *Arrays*

- Creación de objetos *array*:

```
<variable> = new <tipo> [<tamaño>];
```

- Al crear el objeto, el número de elementos (`<tamaño>`) se guarda en un atributo llamado `length`
- El primer elemento del *array* está en la posición 0 y el último, en la posición `length-1`

```
int[] a = new int[20];
a[0] = 15;
int i = a[0];
System.out.println(a.length); ⇒ 20
System.out.println(i); ⇒ 15
```



Inicialización de *Arrays*

- Cuando se instancia un objeto *array*, sus elementos se inicializan al valor por defecto del tipo correspondiente
- Si se conocen los valores iniciales de cada elemento, se pueden inicializar con los valores entre llaves y separados por comas (a la vez que se declara)

```
int[] cuadrados = {0, 1, 4, 9};
```

equivale a

```
int[] cuadrados = new int[4];
cuadrados[0] = 0;
cuadrados[1] = 1;
cuadrados[2] = 4;
cuadrados[3] = 9;
```

Ejemplos de *Arrays*

```
int[] digitos = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

String[] dias = {“lunes”,“martes”,“miércoles”,“jueves”,
                 “viernes”,“sábado”,“domingo”};

Fecha[] festivos = { new Fecha (  1,  1, 2000),
                     new Fecha ( 15,  5, 2000),
                     new Fecha ( 12, 10, 2000),
                     new Fecha (  6, 12, 2000),
                   }
```

Recorrido de una lista:

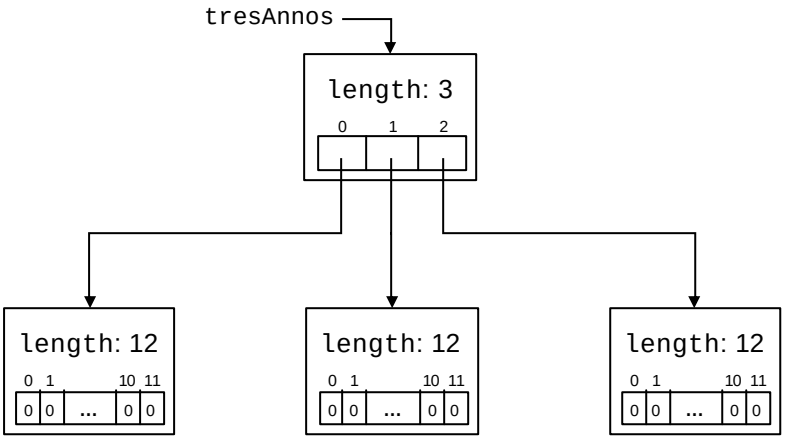
```
int[] lista = new lista[10];
for (int i = 0; i < lista.length; i++)
{
    System.out.println(lista[i]);
}
```

Arrays Multidimensionales

- En Java los *arrays* son todos de una dimensión.
  Un *array* bidimensional es un *array* de *arrays*
- Se necesita un conjunto de corchetes por cada dimensión

```
int[] unAnno = new int[12];
int[][] tresAnnos = new int[3][12];
```

Ejemplo de *Arrays* Multidimensionales



Arrays Bidimensionales No Rectangulares

⇨ Un *array* de 2 dimensiones se puede crear sin especificar el tamaño de su segunda dimensión

```
int[][] tresAnnos = new int[3][];
tresAnnos[0] = new int[12];
tresAnnos[1] = new int[12];
tresAnnos[2] = new int[12];
```

⇨ Si se indica sólo una dimensión, ésta debe ser la primera

```
int[][] tresAnnos = new int[][3]; ⇒ ERROR
```

⇨ Esta separación permite crear *arrays* no rectangulares

```
tresAnnos[0] = new int[12];
tresAnnos[1] = new int[5];
tresAnnos[2] = new int[9];
```

Inicialización de *Arrays* Multidimensionales

⇨ Se necesita un conjunto de datos entre llaves para cada dimensión

```
int[][] matriz = { {1, 2, 3},
                   {4, 5, 6}
                 };
```

equivale a

```
int[][] matriz = new int[2][3];
matriz[0][0] = 1;
matriz[0][1] = 2;
matriz[0][2] = 3;
matriz[1][0] = 4;
matriz[1][1] = 5;
matriz[1][2] = 6;
```

Características Avanzadas de la OO

Fco. Javier Alcalá Casado

Conceptos Avanzados de la OO

- Hay tres conceptos avanzados relacionados con la orientación a objetos:
  - **Encapsulación**: permite la protección de ciertas partes de un objeto del acceso desde otros objetos externos
  - **Herencia**: jerarquía de clases basada en la agrupación de atributos y/o de métodos comunes
  - **Polimorfismo**: tratamiento generalizado de todas las clases pertenecientes a una jerarquía de herencia

Encapsulación

- La *encapsulación* consiste en el agrupamiento de datos y su tratamiento en una misma estructura
- Permite la protección de la manipulación externa de algunas partes de los objetos
- Un objeto suele tener datos y código privados de acceso restringido
- Fuerza al usuario a utilizar una interfaz para acceder a los datos
- Hace que el código sea más fácil de mantener

Modificadores para Restringir el Acceso

- La definición de los miembros de una clase se puede ampliar añadiendo modificadores al principio:
  - `[<modificador>] <tipo> <identificador> ;`
  - `[<modificador>] <tipo> <nombre> ( <args> ) {...}`
- Los modificadores permiten acceder a los datos o al código de manera restringida:
  - `public`: el miembro de la clase es accesible desde cualquier parte del código
  - `private`: el miembro de la clase sólo es accesible desde código perteneciente a la propia clase

Ejemplo de Encapsulación (I)

```
class Fecha
{
    public int dia ;
    public int mes ;
    public int anno ;
}

...
Fecha f = new Fecha() ;
f.dia = 34 ;
f.mes = 14 ;
f.anno = 0 ;
...
```

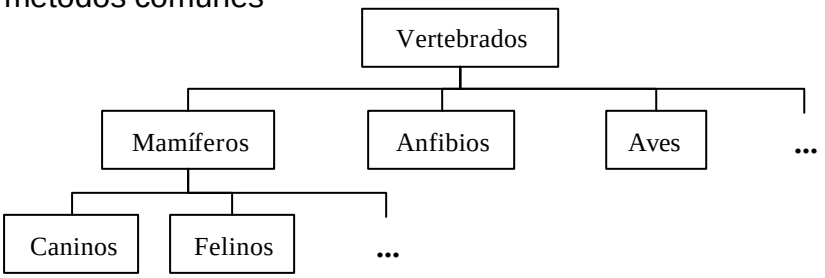
```
class Fecha
{
    private int dia ;
    private int mes ;
    private int anno ;
}

...
Fecha f = new Fecha() ;
f.dia = 34 ;   ⇒ ERROR
f.mes = 14 ;   ⇒ ERROR
f.anno = 0 ; ⇒ ERROR
...
int d ;
d = f.dia ; ⇒ ERROR
...
```

Ejemplo de Encapsulación (II)

```
class Fecha
{
    private int dia ;
    private int mes ;
    private int anno ;
    public void setDia ( int d )
    {
          if ( (d > 0) && (d < 32) )
          {
                dia = d ;
          }
    }
    public int getDia ( )
    {
          return dia ;
    }
}
...
Fecha f = new Fecha() ;
f.setDia(34); ⇒ el método setDia() controla el acceso
int d = f.getDia() ;
...
```

Herencia (I)

⇨ Jerarquía de clases basada en agrupar atributos y/o métodos comunes



⇨ Las clases descendientes se llaman **subclases**

⇨ Las clases ascendientes se llaman **superclases**

⇨ Las subclases "heredan" *características* y *métodos* de las superclases (excepto los constructores)

Herencia (II)

⇨ Supongamos, por ejemplo, que tenemos la clase Jefe y la clase Secretaria definidas como sigue:

```
class Jefe
{
    int numEmpleado;
    String nombre;
    int numDepart;
    int salario;
    int numTrabajadores;
}
```

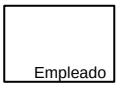
```
class Secretaria
{
    int numEmpleado;
    String nombre;
    int numDepart;
    int salario;
    Jefe trabajaPara;
}
```

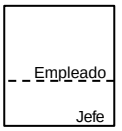
⇨ Las partes comunes se pueden agrupar en una misma clase, manteniendo las otras dos clases con las partes no comunes y heredando de esta nueva clase con la palabra reservada `extends`

Herencia (I I I)

```
class Jefe
{
    int numEmpleado;
    String nombre;
    int numDepart;
    int salario;

    int numTrabajadores;
}

class Secretaria
{
    int numEmpleado;
    String nombre;
    int numDepart;
    int salario;

    Jefe trabajaPara;
}
```

```
class Empleado
{
    int numEmpleado;
    String nombre;
    int numDepart;
    int salario;
}

class Jefe extends Empleado
{
    int numTrabajadores;
}

class Secretaria extends Empleado
{
    Jefe trabajaPara;
}
```





Relación “es-un”

⇨ Para saber si la relación de herencia es correcta, se plantea la pregunta “*¿la subclase* ***es-una*** *superclase?*”. La respuesta debe ser “sí”

⇨ ¿el Jefe **es-un** Empleado? ⇒ Sí
¿la Secretaria **es-un** Empleado? ⇒ Sí

```
class Bici
{
    int numRuedas;
    int numAsientos;
    int velocidadMax;
}
class Avion
 {
    int numRuedas;
    int numAsientos;
    int velocidadMax;
    int numAlas;
}
```

```
class Bici
{
    int numRuedas;
    int numAsientos;
    int velocidadMax;
}
class Avion extends Bici
{
    int numAlas;
}
```

¿Avion **es-una** Bici? ⇒ **NO**

Herencia Simple

- ⇨ Si una clase hereda de una única clase se considera **herencia simple**
- ⇨ Si una clase hereda de varias clases se considera **herencia múltiple**
- ⇨ En Java sólo se permite la herencia simple
- ⇨ La herencia simple hace que el código sea reutilizable

Relación de Contenido ("tiene-un")

- ⇨ Una clase puede contener referencias de objetos de otras clases
- ⇨ Se diferencia de la herencia en que es necesario instanciarlos por separado
- ⇨ Responde afirmativamente a la pregunta:
  ¿<Contenedor> ***tiene-un*** *<Contenido>?*

```
class Motor   class Chasis    class Coche
{             {               {
   ...           ...             Motor m;
}             }                  Chasis ch;
                                 }
```

¿un Coche **tiene-un** Motor? ⇒ Sí
¿un Coche **tiene-un** Chasis? ⇒ Sí

Sobreescritura de Métodos

⇨ También llamados **métodos virtuales**

⇨ Una subclase puede modificar los métodos que ha heredado de la superclase, manteniendo los mismos nombre, tipo de retorno y lista de argumentos

```
class Empleado
{
    ...
    int calcularVacaciones(){...}
}

class Jefe extends Empleado
{
    int numTrabajadores;
    int calcularVacaciones(){...}
}
```

Otras Características de la Herencia

⇨ Todas las clases proporcionadas por Java y las que defina el programador heredan de una clase común: la clase Object

El compilador añade `extends Object` a todas las clases que no heredan explícitamente de ninguna otra

```
class Fecha    ⇒    class Fecha extends Object
{                   {
    ...                 ...
}                   }
```

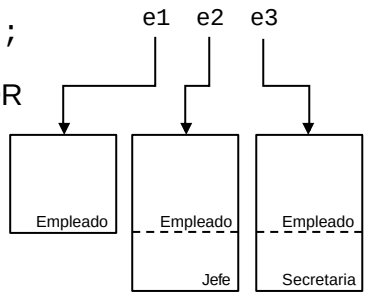
⇨ Los miembros de la clase se pueden proteger con otro modificador, `protected`, cuyo acceso queda restringido a la clase donde se define y a todas sus subclases

Polimorfismo

⇨ *Polimorfismo* indica “muchas formas”

⇨ Una clase sólo tiene una forma, pero una variable que hace referencia a la superclase de una jerarquía puede tener muchas formas (una por cada subclase)

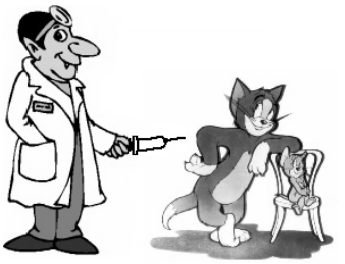
```
Empleado e1 = new Empleado();
Empleado e2 = new Jefe();
Empleado e3 = new Secretaria();

e2.numTrabajadores=15; ⇒ ERROR
((Jefe)e2).numTrabajadores=15;
```



⇨ Pueden utilizarse de dos maneras:
- Parámetros polimórficos
- Colecciones heterogéneas

Parámetros Polimórficos

```
class Mascota {...}
class Raton extends Mascota {...}
class Gato  extends Mascota {...}

class Veterinario
{
    void vacunar ( Mascota m )
    {...}
}

...
Veterinario doctor = new Veterinario();
Gato tom = new Gato();
Raton jerry = new Raton();
doctor.vacunar(tom);
doctor.vacunar(jerry);
...
```



Colecciones Heterogéneas

- Hasta ahora un *array* sólo podía contener elementos del mismo tipo (colección homogénea)
- Utilizando polimorfismo se pueden tener elementos de distinto tipo en un *array* (colección heterogénea)
- Se crean utilizando *arrays* definidos con el tipo superclase

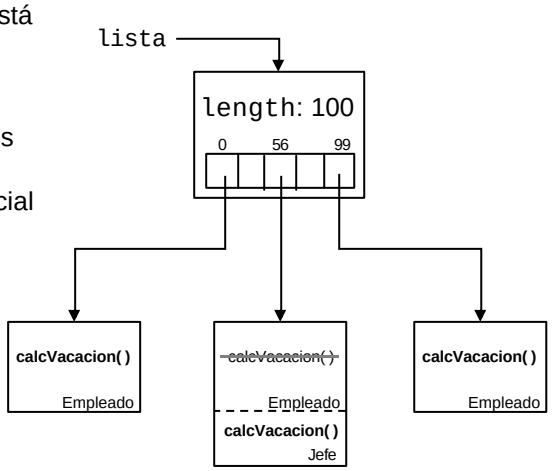
```
Mascota[] listaMascotas = new Mascota[5];
listaMascotas[0] = new Mascota();
listaMascotas[1] = new Gato();
listaMascotas[2] = new Raton();
listaMascotas[3] = new Raton();
listaMascotas[4] = new Gato();
```

Ejemplo de Colecciones Heterogéneas (I)

```
class Empleado
{
    ...
    int salario;
    int calcularVacaciones(){...}
}

class Jefe extends Empleado
{
    int numTrabajadores;
    int calcularVacaciones(){...}
}

Empleado[] lista = new Empleado[100];
lista[0] = new Empleado();
lista[1] = new Empleado();
...
lista[56] = new Jefe();
...
lista[99] = new Empleado();
for (int i = 0; i < lista.length; i++)
{
    System.out.println(lista[i].calcularVacaciones());
}
```

Ejemplo de Colecciones Heterogéneas (II)

- El método de cada elemento que se ejecuta es el que está marcado en negrita
- De esta forma se consigue tratar a todos los elementos por igual, aunque alguno tenga un tratamiento especial



Características Avanzadas del Lenguaje

Fco. Javier Alcalá Casado

Paquetes

- Un *paquete* es una agrupación de clases (librería)
- El programador puede crear sus propios paquetes con la sentencia `package` al principio del fichero fuente

```
package <nombre.paquete>;
```

  Ejemplo: `package empresa.finanzas;`

- La composición de nombres (separados por puntos) está relacionada con la jerarquía de directorios

  `CLASSPATH\empresa\finanzas\`

- Los nombres de los paquetes se suelen escribir en minúsculas

Ejemplo de Paquetes

- Fichero fuente `Empleado.java`:

```
package empresa.finanzas;

public class Empleado
{
    ...
}
```

- La clase `Empleado` realmente se llama `empresa.finanzas.Empleado`
- Si no se indica la sentencia `package`, la clase `Empleado` pertenecerá a un paquete por defecto sin nombre

Sentencia `import`

- La sentencia `import` indica al compilador dónde están ubicadas las clases que estamos utilizando
- Para importar sólo una clase de un paquete:

```
import <nombre.del.paquete>.<NombreClase>;
```

- Para importar todas las clases de un paquete:

```
import <nombre.del.paquete>.*;
```

- El compilador añade a todos los ficheros la línea

```
import java.lang.*;
```

que es el paquete que contiene las clases fundamentales para programar en Java (`System`, `String`, `Object`...)

Ejemplo de `import`

```
import empresa.finanzas.*;

public class Jefe extends Empleado
{
    String departamento;
    Empleado[] subordinados;
}
```

- Si no se pone el `import`, deberíamos referirnos a `Empleado` como `empresa.finanzas.Empleado`
- La clase `Jefe` pertenece al paquete anónimo por defecto
- `String` pertenece al paquete `java.lang`
  No necesita sentencia `import`

Control de Acceso Avanzado

- Las variables y métodos de una clase pueden estar en cualquiera de los cuatro niveles de acceso:
  - publico (`public`): acceso total desde cualquier código
  - por defecto (cuando no se pone nada): desde la misma clase y el mismo paquete
  - protegido (`protected`): acceso desde una jerarquía de clases
  - privado (`private`): máxima protección

Modificador	Misma clase	Subclases	Mismo paquete	Universal
`public`	X	X	X	X
∅	X	X	X	
`protected`	X	X		
`private`	X			

Modificador `static`

- Los miembros de una clase pertenecen a los objetos
- Para acceder a las variables y métodos de una clase es necesario crear primero un objeto
- El modificador `static` puede aplicarse a variables y a métodos para acceder a ellos *sin instanciar* ningún objeto
- Los miembros *estáticos* pertenecen a la clase, no a los objetos

```
[<modifAcceso>] [static] <tipo> <identificador>;

[<modifAcceso>] [static] <tipo> <nombre> ( <args> )
{...}
```

Variables y Métodos Estáticos (I)

- ⇨ Los miembros estáticos pertenecen a la clase y son accesibles por todos los objetos de esa clase
- ⇨ Una variable estática es una variable global dentro de la clase
- ⇨ Para acceder a un miembro estático, hay que utilizar el nombre de la clase a la que pertenece

```
double tangente ( double x )
{
    return Math.sin(x) / Math.cos(x) ;
}
```

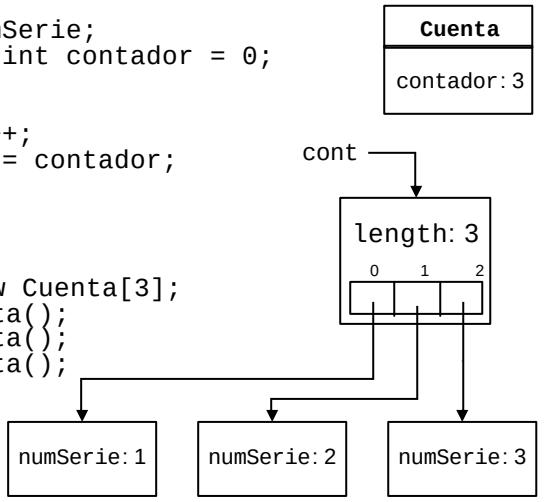
Variables y Métodos Estáticos (II)

- ⇨ Los métodos estáticos sólo pueden acceder a sus propios argumentos y a las variables estáticas y no se pueden sobreescribir

```
public class Error
{
    int x ;
    public static y ;

    public static void main (String args[])
    {
          y = 15 ;
          x = 20 ; ⇒ ERROR
    }
}
```

Ejemplo static

```
public class Cuenta
{
    private int numSerie;
    private static int contador = 0;
    public Cuenta()
    {
          contador++;
          numSerie = contador;
    }
}

...
Cuenta[] cont = new Cuenta[3];
cont[0] = new Cuenta();
cont[1] = new Cuenta();
cont[2] = new Cuenta();
...
```



Clases Abstractas

- Una clase abstracta es una clase de la que no se pueden crear objetos
- Representa un concepto que no se puede instanciar
- Se define anteponiendo el modificador `abstract` a la definición de una clase

```
abstract class Mamifero {...}
class Canino extends Mamifero {...}
class Felino extends Mamifero {...}
class Roedor extends Mamifero {...}
...
Mamifero m = new Mamifero(); ⇒ ERROR
```

Métodos Abstractos

⇨ Un método es **abstracto** si se declara (dentro de una clase abstracta), pero no se implementa

```
abstract class Mamifero
{
    ...
    public abstract void alimentar();
}
```

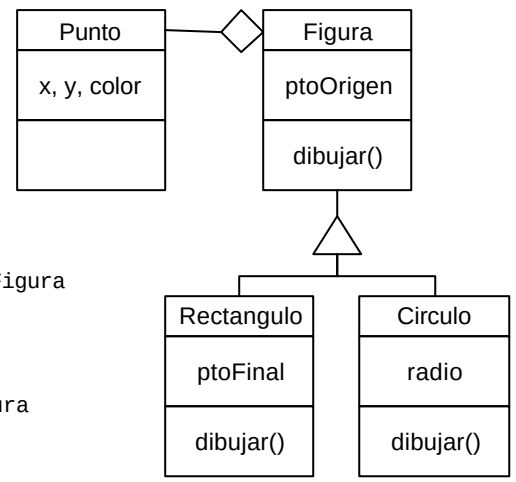
⇨ Todas las subclases de una clase abstracta deben implementar los métodos abstractos que tenga definidos

```
class Canino extends Mamifero
{
    ...
    public void alimentar() {...}
}
class Felino extends Mamifero
{
    ...
    public void alimentar() {...}
}
```

Ejemplo

⇨ Jerarquía de figuras geométricas:

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

abstract class Figura
{
  Punto ptoOrigen;
  abstract void dibujar();
}

class Rectangulo extends Figura
{
  Punto ptoFinal;
  void dibujar() {...}
}

class Circulo extends Figura
{
  int radio;
  void dibujar() {...}
}
```



Interfaces (I)

⇨ Una `interface` es un conjunto de declaraciones de métodos

⇨ Declaración:

```
interface <NombreInterfaz>
{
    <tipo> <nombreMétodo1> ( <args> );
    <tipo> <nombreMétodo2> ( <args> );
    ...
}
```

⇨ Una clase que implemente el código de la interfaz debe implementar todos sus métodos, aunque no lleven código

```
class <NombreClase> implements <NombreInterfaz>
{
    <tipo> <nombreMétodo1> ( <args> ) { <código> }
    <tipo> <nombreMétodo2> ( <args> ) { <código> }
    ...
}
```

Interfaces (II)

⇨ Las interfaces sirven para:

- Declarar métodos que serán implementados por una o más clases
- Definir la interfaz de programación de un objeto, sin mostrar el cuerpo actual de la clase

⇨ Cada interfaz debe escribirse en un fichero `*.java` con el mismo nombre de la interfaz

Equivalencia Interfaz - Clase Abstracta

```
interface Interfaz
{
    <tipo> <método1>();
    <tipo> <método2>();
    ...
    <tipo> <métodoN>();
}
```

equivale a

```
abstract class Interfaz
{
    abstract <tipo> <método1>();
    abstract <tipo> <método2>();
    ...
    abstract <tipo> <métodoN>();
}
```

Operadores de Comparación de Objetos (I)

- El método `equals()`, definido en la clase `Object`, determina si las referencias apuntan a un mismo objeto

  ```
  public boolean equals ( Object obj )
  ```

- El método `equals()` y el operador == comparan las referencias de los objetos, no sus contenidos
- El método `equals()` está sobreescrito en ciertas clases (`String`, `Date`, `File`) en las que devuelve `true` cuando el contenido y el tipo de dos objetos son iguales
- Cualquier clase definida por el usuario puede sobreescribir el método `equals()`

Operadores de Comparación de Objetos (II)

⇨
```
Fecha f1 = new Fecha(1,1,2000);
Fecha f2 = new Fecha(1,1,2000);
if (f1 == f2) ⇒ false
...
if (f1.equals(f2)) ⇒ false
...
```

⇨
```
String s1 = new String("Hola");
String s2 = new String("Hola");
if (s1 == s2) ⇒ false
...
if (s1.equals(s2)) ⇒ true
...
```

⇨ `s1.equals(s2)` equivale a `s2.equals(s1)`

Excepciones

Fco. Javier Alcalá Casado

Introducción

- Java incorpora en el lenguaje el manejo de errores en tiempo de ejecución (división por cero, índice fuera de límites, fichero que no existe...) ⇒ **Tolerancia a fallos**
- Estos errores reciben el nombre de **excepciones**
- Si no se gestiona una excepción, se termina la ejecución del programa con un mensaje de error
- Programar manejando excepciones hace que se separen el código de la tarea a realizar y el código de control de errores
  Ejemplo: abrir y leer de un fichero

Sentencia `try - catch`

- Sintaxis:

```
try
{
    // Código que puede provocar excepciones
}
catch (<NombreExcepción1> e1 )
{
    // Código que gestiona la excepción 1
}
catch (<NombreExcepción2> e2 )
{
    // Código que gestiona la excepción 2
}
```

- Para gestionar excepciones, se coloca el código que puede causarlas dentro de la cláusula `try` y tantas cláusulas `catch` como posibles excepciones haya

Propagación de Excepciones

- Consideremos el siguiente caso:
  - `main()` llama al método `primero()`
  - `primero()` llama al método `segundo()`
  - en `segundo()` se produce una excepción

`main()` ⇒ `primero()` ⇒ `segundo()`

- Si `segundo()` no captura la excepción con un `catch`, se propaga a `primero()`; si éste tampoco la trata, se propaga a `main()`. Por último, si en `main()` tampoco se gestiona, se termina el programa con un error de ejecución

Cláusula `finally`

```
try
{
    // Código protegido que puede provocar excepciones
}
finally
{
    // Código que se ejecuta siempre al final
}
```

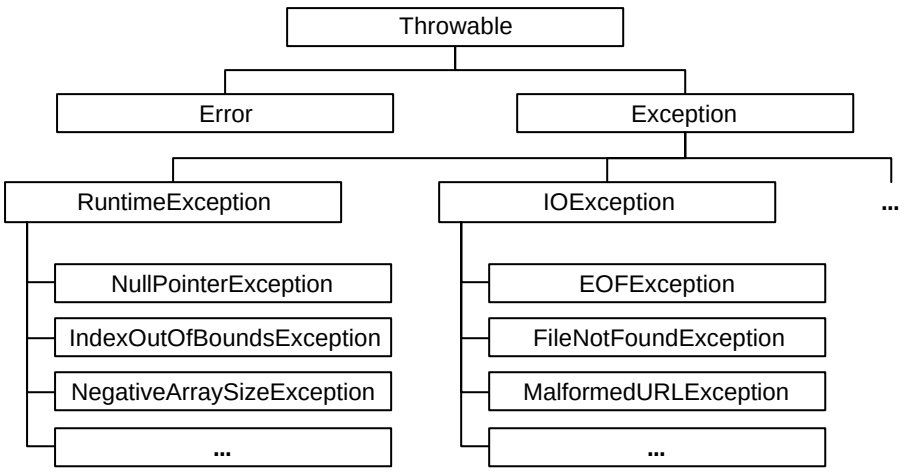
- La cláusula `finally` define un bloque que se ejecuta **siempre** independientemente de que se haya capturado, o no, la excepción
- Si dentro de `try` hay alguna sentencia `return`, se ejecuta `finally` antes de devolver el valor

Ejemplo `try - catch - finally`

⇨ Sistema de riego automático:

```
try
{
    abrirGrifo();
    regarCesped();
}
catch (MangueraRotaException e)
{
    darAlarma();
    avisarFontanero();
}
finally
{
    cerrarGrifo();
}
```

Jerarquía de Excepciones (I)



Jerarquía de Excepciones (II)

- La clase `Throwable` es la superclase de todos los errores y excepciones que se pueden producir en Java
- La clase `Error` está relacionada con errores del sistema, de la JVM o de compilación.
  Son **irrecuperables** (no se capturan)
- Los errores de la clase `Exception` se pueden capturar:
  - `RuntimeException`: son excepciones muy frecuentes relacionadas con errores de programación.
    Son excepciones ***implícitas*** que se pueden no capturar
  - Las demás (`IOException`, `AWTException`...) son excepciones ***explícitas*** que Java obliga a tenerlas en cuenta y capturarlas en algún lugar del código ⇒ **Código robusto**

Regla: "Capturar y/o Propagar"

- La gestión de excepciones explícitas se puede hacer de tres maneras:
  - **Capturando** la excepción con `try – catch`
  - **Propagando** la excepción. Para ello hay que indicarlo explícitamente en la declaración del método

```
[<modificador>] <tipo> <nombre> ( [<args>] )
   throws <Excepción1> [, <Excepción2>...]
{
 // Código del método
}
```

  - **Capturando y propagando** la excepción
- Si no se capturan ni se propagan, se producen errores de compilación

Sentencia `throw`

⇨ Sintaxis:

```
throw <variableExcepción> ;
```

⇨ Con la sentencia `throw` se genera explícitamente una excepción especificada

Ejemplo:

```
...
IOException ioe = new IOException() ;
throw ioe ;
...
```

equivale a

```
...
throw new IOException() ;
...
```

Creación de Excepciones de Usuario

⇨ Las excepciones definidas por el usuario, se crean como clases que extienden la clase `Exception`

⇨ Pueden contener variables y métodos miembro como cualquier otra clase

Ejemplo:

```
class MangueraRotaException extends Exception
{
    public MangueraRotaException ( )
    {
          ...
    }
}
```

⇨ Las excepciones de usuario se deben lanzar con `throw`

Entrada/Salida

Fco. Javier Alcalá Casado

Introducción (I)

- Java representa la E/S con un **stream** (flujo de datos)
- Un *stream* es una conexión entre el programa y la fuente o destino de los datos
- La información se traslada en serie por el *stream*
- Este concepto representa cualquier E/S:
  - lectura/escritura de archivos
  - comunicación a través de Internet
  - lectura de la información de un puerto serie
  - ...

Introducción (II)

- Las clases de E/S se encuentran en el paquete `java.io`
- Hay dos jerarquías diferentes según el tipo de datos:
  - para operaciones con **bytes**
  - para operaciones con **caracteres** (un carácter Unicode está formado por dos bytes)
- Para cada una de las jerarquías hay dos clases definidas según la dirección de las operaciones:
  - para manejar la **entrada**
  - para manejar la **salida**

Introducción (III)

- Operaciones con bytes:
  - para lectura (entrada): clase `InputStream`
  - para escritura (salida): clase `OutputStream`
- Operaciones con caracteres (texto):
  - para lectura (entrada): clase `Reader`
  - para escritura (salida): clase `Writer`
- Las cuatro clases son **abstractas**
- Instanciar una subclase de alguna de éstas equivale a *abrir un stream* (archivo, recurso de Internet...).
  Para *cerrarlo*, hay que utilizar el método `close()`

Jerarquías de E/S de Bytes

- InputStream
  - **FileInputStream**
  - **PipedInputStream**
  - **ByteArrayInputStream**
  - **StringBufferInputStream**
  - SequenceInputStream
  - FilterInputStream
    - DataInputStream
    - LineNumberInputStream
    - BufferedInputStream
    - PushbackInputStream
  - ObjectInputStream
- OutputStream
  - **FileOutputStream**
  - **PipedOutputStream**
  - **ByteArrayOutputStream**
  - FilterOutputStream
    - DataOutputStream
    - BufferedOutputStream
    - PushbackOutputStream
    - PrintStream
  - ObjectOutputStream

Jerarquías de E/S de Caracteres

- Reader
  - BufferedReader
    - LineNumberReader
  - **CharArrayReader**
  - InputStreamReader
    - **FileReader**
  - FilterReader
    - PushbackReader
  - **PipedReader**
  - **StringReader**
- Writer
  - BufferedWriter
  - **CharArrayWriter**
  - OutputStreamWriter
    - **FileWriter**
  - FilterWriter
  - **PipedWriter**
  - **StringWriter**
  - PrintWriter

Uso de las Clases de E/S

- Las clases en **negrita** indican que hay un dispositivo con el que se conecta el *stream* (disco, memoria, URL...)
- Las otras clases añaden características particulares a la forma de enviar los datos por el *stream*
- La intención es combinar ambos tipos de clases para obtener el comportamiento deseado, empezando por una "clase en negrita" y luego añadiendo características

  Ejemplo:

```
FileReader fr = new FileReader("autoexec.bat");
BufferedReader bf = new BufferedReader(fr);
```

Nombre de las Clases de E/S

- Se puede deducir la función de una clase según las palabras que componen su nombre:

`InputStream`, `OutputStream`	Lectura/Escritura de bytes
`Reader`, `Writer`	Lectura/Escritura de caracteres
`File`	Ficheros
`String`, `CharArray`, `ByteArray`, `StringBuffer`	Memoria (según el tipo de datos indicado)
`Piped`	Tubo de datos
`Buffered`	Buffer
`Filter`	Filtro
`Data`	Intercambio de datos de Java
`Object`	Persistencia de objetos
`Print`	Imprimir

Métodos de `InputStream`

⇨ Métodos básicos de lectura:

`int read()`	devuelve un **byte** leído o -1 si es fin de fichero (en el byte de menor peso)
`int read(byte[])`	lee un conjunto de bytes y lo pone en el *array* de bytes dado. Devuelve el número de bytes leídos
`int read(byte[],int,int)`	lee un conjunto de bytes y lo pone en el *array* de bytes dado empezando en la posición dada con la longitud dada. Devuelve el número de bytes leídos

⇨ Otros métodos:

`void close()`	cierra el *stream* abierto
`int avaible()`	devuelve el número de bytes disponibles para leer
`long skip(long)`	salta el número de bytes indicado

Métodos de `OutputStream`

⇨ Métodos básicos de escritura:

`void write(int)`	escribe el **byte** de menor peso
`void write(byte[])`	escribe el *array* de bytes dado
`void write(byte[],int,int)`	escribe el *array* de bytes dado empezando en la posición dada con la longitud dada

⇨ Otros métodos:

`void close()`	cierra el *stream* abierto
`void flush()`	fuerza a que se escriban las operaciones de escritura que haya pendientes

Métodos de `Reader`

⇨ Métodos básicos de lectura:

`int read()`	devuelve un **carácter** leído o -1 si es fin de fichero
`int read(char[])`	lee un conjunto de caracteres y lo pone en el *array* de bytes dado. Devuelve el número de bytes leídos
`int read(char[],int,int)`	lee un conjunto de caracteres y lo pone en el *array* de bytes dado empezando en la posición dada con la longitud dada. Devuelve el número de bytes leídos

⇨ Otros métodos:

`void close()`	cierra el *stream* abierto
`long skip(long)`	salta el número de caracteres indicado

Métodos de `Writer`

⇨ Métodos básicos de escritura:

`void write(int)`	escribe el **carácter** contenido en los dos bytes de menor peso
`void write(char[])`	escribe el *array* de caracteres dado
`void write(char[],int,int)`	escribe el *array* de caracteres dado empezando en la posición dada con la longitud dada
`void write(String)`	escribe una cadena de caracteres
`void write(String,int,int)`	escribe la cadena de caracteres dada empezando en la posición dada con la longitud dada

⇨ Otros métodos:

`void close()`	cierra el *stream* abierto
`void flush()`	fuerza a que se escriban las operaciones de escritura que haya pendientes

Lectura y Escritura de Archivos (I)

- Las clases `FileInputStream` y `FileOutputStream` permiten leer y escribir *bytes* en archivos *binarios*
- Las clases `FileReader` y `FileWriter` permiten leer y escribir *caracteres* en archivos *de texto*
- Cada llamada a `read()` o `write()` accede al disco para un único byte o un único carácter ⇒ poco eficiente
- Para mejorarlo, se utilizan las clases que implementan un *buffer*, de modo que se lee del disco (o se escribe) un conjunto de bytes o caracteres en cada acceso

Lectura y Escritura de Archivos (II)

- En la lectura, los constructores de `FileInputStream` y `FileReader`, si no encuentran el archivo indicado, pueden lanzar la excepción `FileNotFoundException`
- En la escritura, los constructores de `FileOutputStream` y `FileWriter` pueden lanzar la excepción `IOException`.
  Si no se encuentra el archivo dado, se crea nuevo.
  Si ya existe, por defecto escribe desde el comienzo; pero se puede indicar que añada al final (con un 2º parámetro `true`)

```
FileInputStream fis = new FileInputStream("fich.bin");
BufferedInputStream bis = new BufferedInputStream(fis);
int b = bis.read();   // lee un sólo byte, pero llena
                      // el buffer de datos para
                      // próximas lecturas
```

Lectura de Archivos Binarios

```
try
{
    FileInputStream fis;
    BufferedInputStream bis;
    int dato;
    fis = new FileInputStream(“archivo.bin”);
    bis = new BufferedInputStream(fis);
    while ((dato=read()) != -1) ; // lee hasta el
                                  // fin del stream
}
catch (FileNotFoundException e1)
{
    System.err.println(“Archivo no encontrado:”+e1);
}
catch (IOException e2)
{}
finally
{
    bis.close();
}
```

Escritura de Archivos Binarios

```
try
{
    FileOutputStream fos;
    BufferedOutputStream bos;
    int dato;
    fos = new FileOutputStream(“archivo.bin”);
    bos = new BufferedOutputStream(fos);
    dato = (int)’A’; // casting para llamar a write
    bos.write(dato);
}
catch (IOException e)
{
}
finally
{
    bos.close();
}
```

Lectura de Archivos de Texto

```
String texto = new String();
try
{
    FileReader fr = new FileReader("archivo.txt");
    BufferedReader br = new BufferedReader(fr);
    String linea;
    while ((linea=br.readLine()) != null)
    {
          texto += linea;
    }
}
catch (FileNotFoundException e1)
{
    System.err.println("Archivo no encontrado:" + e1);
}
catch (IOException e2)
{}
finally
{
    br.close();
}
```

Escritura de Archivos de Texto (I)

```
try
{
    FileWriter fw = new FileWriter("archivo.txt");
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter pw = new PrintWriter(bw);
    pw.println("Esta es la primera línea");
}
catch (IOException e)
{
}
finally
{
    pw.close();
}
```

Escritura de Archivos de Texto (II)

```
// modo append (añadiendo al final del archivo)
try
{
    FileWriter fw = new FileWriter("archivo.txt",true);
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter pw = new PrintWriter(bw);
    pw.println("Esta es la segunda línea");
}
catch (IOException e)
{
}
finally
{
    pw.close();
}
```

Conversión de *Streams* de Bytes a Caracteres

- Las clases `InputStreamReader` y `OutputStreamReader` permiten convertir un *stream* de bytes en un *stream* de caracteres, para lectura y escritura respectivamente

```
FileInputStream fis = new FileInputStream("f.bin");
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);
...
String s = br.readLine();
br.close();
```

- Las clases `InputStreamReader` y `OutputStreamReader` son la única relación entre ambas jerarquías
- No existen clases que realicen la conversión inversa

DataInputStream y DataOutputStream

⇨ Las clases `DataInputStream` y `DataOutputStream` permiten leer y escribir **tipos primitivos** en modo binario

⇨ Métodos de `DataInputStream`:

```
boolean readBoolean()          char readChar()
byte readByte()                short readShort()
int readInt()                  long readLong()
float readFloat()              double readDouble()
```

⇨ Métodos de `DataOutputStream`:

```
void writeBoolean(boolean)     void writeChar(char)
void writeByte(byte)           void writeShort(short)
void writeInt(int)             void readLong(long)
void writeFloat(float)         void writeDouble(double)
```

Uso de DataInputStream y DataOutputStream

⇨ Lectura de datos primitivos:

```
FileInputStream fis = new FileInputStream("f.bin");
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
int i = dis.readInt();
float f = dis.readFloat();
boolean b = dis.readBoolean();
dis.close();
```

⇨ Escritura de datos primitivos :

```
FileOutputStream fos = new FileOutputStream("f.bin");
BufferedOutputStream bos = new BufferedOutputStream(fos);
DataOutputStream dos = new DataOutputStream(bos);
dos.writeInt(7);
dos.writeFloat(3.15F);
dos.writeBoolean(true);
dos.close();
```

La Clase `File`

- La clase `File` hace referencia a un archivo o un directorio
- Se utiliza para obtener información del archivo o del directorio (tamaño, fecha, atributos...)
- Constructores:
  - `File(String nombre)`
  - `File(String dir, String nombre)`
  - `File(File dir, String nombre)`

```
File f1 = new File("C:\\windows\\notepad.exe");
File f2 = new File("C:\\windows", "notepad.exe");
File f3 = new File("C:\\windows");
File f4 = new File(f3, "notepad.exe");
```

Métodos de la Clase `File` (I)

- Si `File` representa un archivo:

`boolean exists()`	`true` si el archivo existe
`boolean isFile()`	`true` si el objeto es una archivo
`long length()`	tamaño del archivo en bytes
`long lastModified`	fecha de la última modificación
`boolean canRead()`	`true` si se puede leer
`boolean canWrite()`	`true` si se puede escribir
`boolean delete()`	borra el archivo
`boolean renameTo(File)`	cambia el nombre

- Si `File` representa un directorio

`boolean isDirectory()`	`true` si el objeto es un directorio
`boolean mkdir()`	crea el directorio
`boolean delete()`	borra el directorio
`String[] list()`	devuelve los archivos que se encuentran en el directorio

Métodos de la Clase `File` (II)

⇨ Otros métodos de la clase `File` relacionados con el *path* del archivo:

`String getPath()`	devuelve el *path* que contiene el objeto `File`
`String getName()`	devuelve el nombre del archivo
`String getAbsolutePath()`	devuelve el *path* absoluto
`String getParent()`	devuelve el directorio padre

Entrada/Salida Estándar

⇨ En Java la entrada desde teclado y la salida a pantalla se realizan a través de la clase `System`

⇨ `System` contiene tres atributos estáticos:

- `System.in`: objeto de la clase `InputStream` que lee *datos* de la entrada estándar (teclado)
- `System.out`: objeto de la clase `PrintStream` que escribe *datos* en la salida estándar (pantalla)
- `System.err`: objeto de la clase `PrintStream` que escribe *mensajes de error* en la salida estándar (pantalla)

Entrada desde Teclado (I)

- El método `Sistem.in.read()` permite leer, en cada llamada, un único carácter y lo devuelve como **`int`**
- Puede provocar la excepción `IOException`, que debe ser tratada
- Para leer un tipo diferente de `int`, hay que hacer un *cast*

```
try
{
    char c = (char)System.in.read();
}
catch (IOException exc)
{
    // tratamiento de la excepción
}
```

Entrada desde Teclado (II)

- Para leer toda una línea se debe usar un bucle, unir los caracteres y detectar el carácter `'\n'`

```
char c;
String linea = new String("");
try
{
    while ((c = System.in.read()) != '\n')
        linea += c;  //Concatena c con linea
}
catch (IOException exc)
{
}
```

Entrada desde Teclado (III)

- Un método más eficiente y "sencillo" para leer de teclado utiliza la clase `BufferedReader` que contiene el método `readLine()`
- `readLine()` lee caracteres de un *stream* hasta encontrar un delimitador de línea y los devuelve como un `String`. También puede lanzar la excepción `IOException`

```
try
{
    InputStreamReader isr;
    BufferedReader br;
    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);
    String linea = br.readLine();
}
catch (IOException exc) {}
```

Salida por Pantalla

- Se utilizan los métodos:
  - `System.out.print(<argumento>)`: imprime por pantalla el argumento dado independientemente del tipo que sea
  - `System.out.println(<argumento>)`: igual que el anterior, pero añadiendo un salto de línea
- Ambos métodos pueden imprimir:
  - valores directamente

```
System.out.println("Hola Mundo");
double numeroPI = 3.141592654;
System.out.println(numeroPI);
```

  - varios valores concatenados con el operador +

```
System.out.println("Hola Mundo" + numeroPI);
```

Lectura de un Archivo de Internet

- Java proporciona un mecanismo que permite leer archivos de Internet mediante un *stream*
- La clase `URL` del paquete `java.net` representa una dirección de Internet
- El método `InputStream openStream(URL dir)` de `URL` abre un *stream* de lectura con origen en la dirección dada

```
URL dir =
   new URL("http://www.sia.eui.upm.es/index.html");
InputStreamReader isr =
   new InputStreamReader(dir.openStream());
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
...
```

Clases Útiles

Fco. Javier Alcalá Casado

Clases para Manejar Cadenas de Caracteres

- Hay dos clases en el paquete `java.lang` que permiten la manipulación de cadenas de caracteres:
  - La clase `String` maneja cadenas constantes, es decir, que no pueden cambiar
  - La clase `StringBuffer` permite cambiar la cadena insertando, añadiendo o borrando caracteres
- La clase `String` es más eficiente, mientras que la clase `StringBuffer` ofrece más posibilidades
- El operador + entre objetos `String` utiliza internamente la clase `StringBuffer` y su método `append()`
- Se pueden utilizar lo métodos de `String` sobre literales Ejemplo: `"Hola".length()`

La Clase `String`

`String(...)`	Constructores para crear `String` a partir de varios tipos
`String(StringBuffer)`	Constructor a partir de un `StringBuffer`
`charAt(int)`	Devuelve el carácter en la pos indicada
`getChars(int,int,char[],int)`	Copia los caracteres indicados en la pos indicada de un *array* de caracteres
`indexOf(String,int)`	Devuelve la pos en la que aparece un `String` dentro de otro
`lastIndexOf(String)`	Devuelve la última vez que un `String` aparece en otro hacia el principio
`length()`	Devuelve el número de caracteres
`replace(char,char)`	Sustituye un carácter por otro
`startsWith(String)`	Indica si un `String` comienza con otro
`substring(int,int)`	Devuelve un `String` extraído de otro
`toLowerCase()`	Convierte en minúsculas
`toUpperCase()`	Convierte en mayúsculas
`trim()`	Elimina los espacios en blanco al comienzo y final de la cadena

La Clase `StringBuffer`

`StringBuffer(...)`	Constructores para crear `StringBuffer` a partir de varios tipos
`append(...)`	Añade un `String` o una variable de cualquier tipo al `StringBuffer`
`capacity()`	Devuelve el espacio libre del `StringBuffer`
`charAt(int)`	Devuelve el carácter en la pos indicada
`getChars(int,int,char[],int)`	Copia los caracteres indicados en la pos indicada de un *array* de caracteres
`insert(int,)`	Inserta un `String` o un valor en la posición indicada
`length()`	Devuelve el número de caracteres
`reverse()`	Cambia el orden de los caracteres
`setCharAt(int,char)`	Cambia el carácter en la posición indicada
`setLength(int)`	Cambia el tamaño del `StringBuffer`
`toString()`	Convierte el `StringBuffer` en `String`

La Clase `Vector` (I)

- La clase `Vector` (paquete `java.util`) representa una colección heterogénea de objetos (referencias a objetos de tipo `Object` o a cualquiera de sus subclases)
- El vector al crearse reserva cierta cantidad de memoria, aunque sus elementos sólo utilicen una parte
- El tamaño del vector se incrementa por bloques cuando se añade y se agota el espacio reservado. El tamaño de incremento se indica en el atributo `capacityIncrement`
- El vector se mantiene *compacto* en todo momento
- Cada elemento es accesible a través de un índice, pero no con los corchetes, [ ], sino con el método `elementAt(index)`

La Clase `Vector` (II)

⇨ Atributos:

- `int capacityIncrement`: incremento en la capacidad del vector. Si vale cero, duplica el tamaño actual del vector
- `int elementCount`: número de elementos válidos del vector
- `Object[] elementData`: *array* de objetos donde se guardan los elementos

⇨ Constructores:

- `Vector()`: Crea un vector vacío (capacidad 10, incremento 0)
- `Vector(int initialCapacity)`: Crea un vector vacío con la capacidad dada (incremento 0)
- `Vector(int initialCapacity, int initialIncrement)`: Crea un vector vacío con la capacidad y el incremento dados

La Clase `Vector` (III)

⇨ Métodos:

`int` **`capacity`**`()`	devuelve la capacidad que tiene el vector
`int` **`size`**`()`	devuelve el número de elementos en el vector
`boolean` **`contains`**`(Object elem)`	devuelve `true` si el vector contiene el objeto especificado
`int` **`indexOf`**`(Object elem)`	devuelve la posición de la primera vez que aparece el objeto que se le pasa
`Object` **`elementAt`**`(int index)`	devuelve el elemento situado en la posición indicada **(*)**
`void` **`setElementAt`**`(Object elem,int index)`	reemplaza el objeto que corresponde al índice por el objeto que se le pasa **(*)**
`void` **`removeElementAt`**`(int index)`	borra el objeto situado en la posición indicada **(*)**
`void` **`addElement`**`(Object elem)`	añade un objeto al final
`void` **`insertElementAt`**`(Object elem,int index)`	inserta el objeto que se le pasa en la posición indicada, desplazando el resto de elementos en el vector **(*)**

Los métodos con **(*)** pueden lanzar la excepción `ArrayIndexOutOfBoundsException`

La Clase `Math`

⇨ La clase `Math` del paquete `java.lang` tiene todos sus miembros estáticos

`E`	Constante del número e (2.718...)
`PI`	Constante con el valor π (3.1415...)
`double` **`abs`**`(double x)`	Valor absoluto de x
`long` **`round`**`(double x)`	Entero más cercano a x
`double` **`ceil`**`(double x)`	Entero más cercano hacia +infinito
`double` **`floor`**`(double x)`	Entero más cercano hacia -infinito
`double` **`cos`**`(double x)`	Coseno de x
`double` **`sin`**`(double x)`	Seno de x
`double` **`tan`**`(double x)`	Tangente de x
`double` **`acos`**`(double x)`	Arco coseno de x
`double` **`asin`**`(double x)`	Arco seno de x
`double` **`atan`**`(double x)`	Arco tangente de x entre $-\pi/2$ y $\pi/2$
`double` **`atan2`**`(double,double)`	Arco tangente entre $-\pi$ y π
`double` **`exp`**`(double)`	Exponencial de x
`double` **`log`**`(double)`	Logaritmo neperiano de x
`double` **`pow`**`(double x,double y)`	x elevado a y
`double` **`sqrt`**`(double)`	Raíz cuadrada de x
`double` **`max`**`(double a,double b)`	Máximo entre a y b
`double` **`min`**`(double a,double b)`	Mínimo entre a y b
`double` **`random`**`()`	Número aleatorio $\in$ [0.0,1.0)

Clases Envolventes

⇨ Java no trata a los tipos primitivos como objetos. Los trata de forma diferente por razones de eficiencia

⇨ Las **clases envolventes** son un complemento de los tipos primitivos

⇨ Cada tipo primitivo tiene su correspondiente clase envolvente en el paquete `java.lang`:

`byte`	⇒ **`Byte`**	`short`	⇒ **`Short`**
`int`	⇒ **`Integer`**	`long`	⇒ **`Long`**
`float`	⇒ **`Float`**	`double`	⇒ **`Double`**
`boolean`	⇒ **`Boolean`**	`char`	⇒ **`Character`**

La Clase `Integer`

`Integer``(int)`	Constructor desde el tipo primitivo
`Integer``(String)`	Constructor desde una cadena de caracteres
`byte` **`byteValue`**`()`	Convierte al tipo primitivo `byte`
`short` **`shortValue`**`()`	Convierte al tipo primitivo `short`
`int` **`intValue`**`()`	Convierte al tipo primitivo `int`
`long` **`longValue`**`()`	Convierte al tipo primitivo `long`
`float` **`floatValue`**`()`	Convierte al tipo primitivo `float`
`double` **`doubleValue`**`()`	Convierte al tipo primitivo `double`
`Integer` **`decode`**`(String)`	Convierte un `String` a `Integer`
`int` **`parseInt`**`(String)`	Convierte un `String` en entero con signo
`String` **`toString`**`()`	Convierte el objeto a `String`
`Integer` **`valueOf`**`(String)`	Devuelve un nuevo objeto `Integer` con el valor del `String`
`String` **`toBinaryString`**`(int)`	Crea un `String` en base 2
`String` **`toOctalString`**`(int)`	Crea un `String` en base 8
`String` **`toHexString`**`(int)`	Crea un `String` en base 16
`MAX_VALUE`	Constante con el valor máximo posible
`MIN_VALUE`	Constante con el valor mínimo posible
`TYPE`	Constante que representa al tipo primitivo

La Clase `Double`

`Double``(double)`	Constructor desde el tipo primitivo
`Double``(String)`	Constructor desde una cadena de caracteres
`byte` **`byteValue`**`()`	Convierte al tipo primitivo `byte`
`short` **`shortValue`**`()`	Convierte al tipo primitivo `short`
`int` **`intValue`**`()`	Convierte al tipo primitivo `int`
`long` **`longValue`**`()`	Convierte al tipo primitivo `long`
`float` **`floatValue`**`()`	Convierte al tipo primitivo `float`
`double` **`doubleValue`**`()`	Convierte al tipo primitivo `double`
`String` **`toString`**`()`	Convierte el objeto a `String`
`Double` **`valueOf`**`(String)`	Devuelve un nuevo objeto `Double` con el valor del `String`
`boolean` **`isInfinite`**`()`	Devuelve `true` si es infinito
`boolean` **`isNaN`**`()`	Devuelve `true` si no es un número (Not-a-Number)
`MAX_VALUE`	Constante con el valor máximo posible
`MIN_VALUE`	Constante con el valor mínimo posible
`POSITIVE_INFINITY`	Constante con el valor +∞
`NEGATIVE_INFINITY`	Constante con el valor -∞
`NaN`	Constante con el valor NaN
`TYPE`	Constante que representa al tipo primitivo

Threads

Fco. Javier Alcalá Casado

Introducción (I)

- Un ordenador con un solo procesador es *monotarea*
- Algunos sistemas operativos simulan **multitarea**, dividiendo el tiempo del procesador entre varios procesos
- Un **proceso** es cada uno de los programas o aplicaciones que se ejecutan de forma independiente
  Tiene asociado:
  - Tiempo de CPU
  - Memoria para CÓDIGO
  - Memoria para DATOS
- Un ***thread*** (también llamado *hilo* o *hebra de ejecución o proceso ligero*) es un flujo de ejecución simple dentro de un proceso

Introducción (II)

- Un único proceso puede tener varios hilos ⇒ **multihilo**
- Multitarea vs. multihilo
- La JVM es un proceso, desde el punto de vista del SO, con varios *threads* ejecutándose a la vez:
  - *Garbage Collector*
  - *AWT* (*Abstract Window Toolkit*)
  - Método `main()`
- Un hilo tiene asociado tiempo de CPU, memoria para código y memoria para datos, pero se puede comunicar, coordinar y sincronizar con otros hilos.

Creación de un *Thread*

- Hay dos formas de crear un *thread*
  - Declarar una clase que implemente la interfaz `Runnable` y crear un objeto de tipo `Thread` a partir de ella
  - Crear una clase que herede de la clase `Thread`
- La clase `Thread` encapsula todo el **control** necesario sobre los hilos de ejecución
- Hay que distinguir claramente entre un objeto `Thread` (parte estática) y un hilo de ejecución o *thread* (parte dinámica)
  La única forma de controlar el comportamiento de los hilos es a través de la clase `Thread`

Creación de un *Thread* implementando Runnable

```
class HiloRunnable implements Runnable
{
    public void run ( )
    {
          System.out.println(“Hola Mundo”);
    }
}
class HolaMundo
{
    public static void main (String args[])
    {
      HiloRunnable hr = new HiloRunnable ( );
      Thread t = new Thread(hr); // Crea el hilo
      t.start ( ) ;              // Lo ejecuta
    }
}
```

Ejemplo: *Hola* Multihilo implementando Runnable

```
class Hilo implements Runnable
{
  String nombre ;
  Hilo ( String n )
  {
    nombre = n ;
  }
  public void run ( )
  {
    try
    {
      Thread.currentThread().sleep((int)(Math.random()*3000));
    }
    catch ( InterruptedException e ) { }
    System.out.println("Hola, soy " + nombre ) ;
  }
}
class MultiHola
{
  public static void main (String args[])
  {
    for (int i = 0 ; i < 10 ; i++ )
         new Thread ( new Hilo("Hilo "+i) ).start();
  }
}
```

Creación de un *Thread* extendiendo Thread

```
class HiloThread extends Thread
{
    // Sobreescribe el método run()
    public void run ( )
    {
      System.out.println(“Hola Mundo”);
    }

    public static void main (String args[])
    {
      Thread t = new HiloThread(); // Crea el hilo
      t.start ( ) ;                // Lo ejecuta
    }
}
```

Ejemplo: *Hola* Multihilo extendiendo Thread

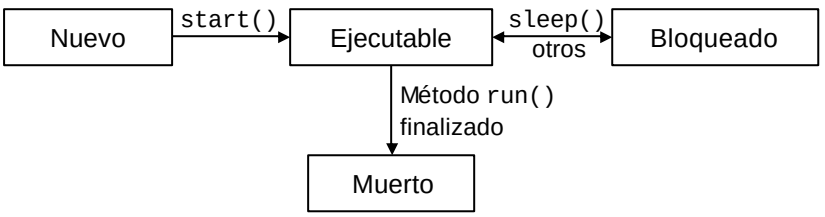
```
class MultiHola extends Thread
{
  String nombre ;
  Hilo ( String n )
  {
    nombre = n ;
  }
  public void run ( )
  {
    try
    {
      sleep( (int)(Math.random()*3000) );
    }
    catch( InterruptedException e ) { }
    System.out.println("Hola, soy " + nombre ) ;
  }
  public static void main (String args[])
  {
    for (int i = 0 ; i < 10 ; i++ )
        new MultiHola("Hilo "+i).start();
  }
}
```

Características de Ambos Métodos de Creación

- Implementar la interfaz `Runnable`
  - Mejor diseño orientado a objetos
  - Limitado por la herencia simple ⇒ Applets
- Extender la clase `Thread`
  - Código más simple
  - Complica futuras modificaciones del código

Estados de un *Thread*

- Un *thread* puede estar en cuatro estados:
  - *Nuevo*: el *thread* ha sido creado, pero no inicializado con el método `start()`
  - *Ejecutable*: el *thread* puede estar ejecutándose o preparado para ejecutarse. Se inicia con `start()`
  - *Bloqueado*: el *thread* podría estar ejecutándose, pero está esperando que termine alguna otra tarea, p.e., E/S de disco
  - *Muerto*: terminación del *thread* al finalizar el método `run()`

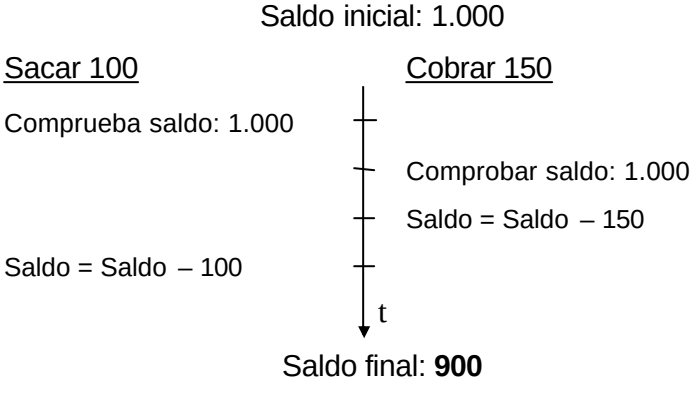


Sincronización de *Threads* (I)

- Problema de la cuenta bancaria:
  - Saldo de la cuenta: 1.000 €
  - Operaciones que se van a realizar en un instante dado:
    - Sacar desde un cajero 100 €
    - Cobro de un recibo de 150 €
  - Ambas operaciones tienen la misma estructura:
    - Comprobar si se puede realizar la operación bancaria
    - Realizar la operación
  - El banco dispone de un servidor que atiende este tipo de peticiones lanzando un hilo por cada una de ellas
  - Tras la ejecución de ambos hilos, la cuenta se debería quedar con 750 €

Sincronización de *Threads* (II)

- Si la ejecución de ambas operaciones se realizan simultáneamente, puede suceder lo siguiente:



Sincronización de *Threads* (III)

- Una **región crítica** es un recurso (fichero, impresora, objeto...) accesible por más de un *thread* a la vez
- La **sincronización** es un mecanismo que evita que se haga más de un acceso simultáneo a una región crítica
- El modificador `synchronized` permite la ejecución exclusiva de un método sin ser interrumpido

```
public synchronized void sacarCajero (int cantidad)
{
    int saldo = getSaldo();
    if (saldo > cantidad)
          saldo = saldo - cantidad ;
}
```