Aplicaciones Telemáticas

Tema 1.1. Introducción a Java

J. E. López Patiño, F. J. Martínez Zaldívar

DEPARTAMENTO DE COMUNICACIONES

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Índice

1. Introducción
2. Variables
3. Expresiones
4. Sentencias
5. Clases y objetos
6. Más características del lenguaje Java
7. Ficheros `.jar`
8. Javadoc
9. Estilo de programación

Índice

1 Introducción
- Contexto
- Herramientas y pruebas

Índice

1 Introducción
- Contexto
- Herramientas y pruebas

Qué es Java

- ¿Qué es?
  - Lenguaje de programación orientado a objetos y plataforma de computación
- ¿Quien?
  - Sun Microsystems: 1995
- ¿Cómo?
  - Inicialmente compilado, código objeto: *byte code*
  - Interpretado por una máquina virtual:
    - Independiencia de arquitectura o sistema operativo.
- Partes:
  - JRE: Java Runtime Environment
    - JVM: Java Virtual Machine
    - Clases, librerías, . . .
    - . . .
  - JDK: Java Development Kit:
    - Compilador
    - Herramientas de desarrollo (JavaDoc, Java Debugger,. . .)

Interés en AA. TT.

- Motivo: Android
- Estructura del lenguaje
  - Variables
  - Expresiones y sentencias
  - Clases y objetos
  - ...
- Referencias:
  - `https://docs.oracle.com/javase/tutorial/`
  - ...
- *Entrenamiento* y autoaprendizaje: sugerencia

  `http://www.sololearn.com`

  Apertura de cuenta personal, autoaprendizaje y realización de tests y pruebas.

Índice

1. Introducción
   - Contexto
   - Herramientas y pruebas

Software necesario

- Editor de texto plano:
  - `Sublime`
  - ...
- JDK: Java Develpment Kit

  `http://www.oracle.com/technetwork/es/java/javase/downloads/index.html`

  Entorno completo de compilación y ejecución para Java

Pruebas

- Marco de pruebas: edición fichero plantilla

Prueba.java

```
class Prueba {

   // IMPORTANTE: relación entre nombre de clase (Prueba) y nombre de fichero (Prueba.java)


    public static void main (String [ ] args) {

        // Pruebas y ejemplos

    }

}
```

- Compilación y ejecución: en ventana de línea de comandos (**NO** en IDE como NetBeans, Eclipse, IntelliJ IDEA, JCreator, BlueJ, . . .)

Ventana de línea de comandos

```
> javac Prueba.java %%% Aparece Prueba.class si no hay errores
> java Prueba
```

Índice

2 Variables

Tipos de datos simples o primitivos

Tipo	Tamaño (bits)	Mínimo	mín $\lvert\cdot\rvert$	Máximo	*Wrapper* (Envoltorio)
byte	8	-128	0	127	Byte
short	16	$-32\,768 = -2^{15}$	0	$32\,767 = 2^{15} - 1$	Short
int	32	$-2\,147\,483\,648 = -2^{31}$	0	$2\,147\,483\,647 = 2^{31} - 1$	Integer
long	64	$-9\,223\,372\,036\,854\,775\,808 = -2^{63}$	0	$9\,223\,372\,036\,854\,775\,807 = 2^{63} - 1$	Long
float	32	$-3,4 \cdot 10^{38}$	$1,4 \cdot 10^{-45}$	$3,4 \cdot 10^{38}$	Float
double	64	$-1,8 \cdot 10^{308}$	$4,9 \cdot 10^{-324}$	$1,8 \cdot 10^{308}$	Double
boolean	?	`false`	-	`true`	Boolean
char	16	Unicode 0 ='\u0000'	-	Unicode 65 535 ='\uffff' $= 2^{16} - 1$	Character
void	-	-	-	-	Void

Declaración

```
<tipo> <nombre> [ = <valor> ] ;
```

Ejemplo

```
char letra = 'w';
double x = 9.7;
int j;
```

Tipos short y char

- Requieren mismo almacenamiento: 16 bits
- Interpretan el contenido de manera diferente:

Ej_2_1.java: diferencias entre short y char

```
/* ' ':  código UTF-16 del carácter
         UNICODE (https://unicode-table.com/es/) entrecomillado */
char j ='ñ';
char k = 241;

short l = 241;
short m = 'ñ';

System.out.println("j: "  + j +  ", k: "  + k +  " l: "  + l +  ", m: "  + m);

/* Potenciales problemas en compilación/ejecución: véase fichero Ej_1_1.java */
```

- Codificaciones del caracter UNICODE *LATIN SMALL LETTER N WITH TILDE' (U+00F1)*: **ñ**

http://www.fileformat.info/info/unicode/char/f1/index.htm

Conversiones de tipo: contextos

- *Casting* (conversión explícita)

```
int x;
float y;
...
x = (int)y;
```

- Asignación

```
double a;
float b;
...
a = b;
```

- Invocación a método (argumentos)

Ej_2_2.java

```
class Ej_1_2 {

    static short f (int a, double b) {
        return (short) (a+b);
    }

    public static void main(String[] args) {
        long x = 1;
        byte p1 = 2; /* CÁMBIESE A TIPO float y obsérvese consecuencias...*/
        float p2 = 3.4f;
        x = f( p1, p2 );
        System.out.println( "x = " + x );
    }

}
```

- Promoción numérica:
  `byte` $\rightarrow$ `short` $\rightarrow$ `int` $\rightarrow$ `long` $\rightarrow$ `float` $\rightarrow$ `double`
- ...

Conversiones de tipo

- Algunas formas de conversión:
  - Conversión de identidad
  - Conversión primitiva expansiva *(widening)*
    - `byte` → `short, int, long, float, double`
    - `short` → `int, long, float, double`
    - `char` → `int, long, float, double`
    - `int` → `long, float, double`
    - `long` → `float, double`
    - `float` → `double`
  - Conversión primitiva compresiva *(narrowing)*
    - `short` → `byte, char`
    - `char` → `byte, short`
    - `int` → `byte, short, char`
    - `long` → `byte, short, char, int`
    - `float` → `byte, short, char, int, long`
    - `double` → `byte, short, char, int, long, float`

  En este caso, se requiere **casting**

- Algunos tipos de conversión (cont.):
  - Conversión con empaquetado *boxing* (de tipo primitivo a tipo de referencia —objeto—)
    - `boolean` → `Boolean`
    - `byte` → `Byte`
    - `short` → `Short`
    - `char` → `Character`
    - `int` → `Integer`
    - `long` → `Long`
    - `float` → `Float`
    - `double` → `Double`
  - Conversión con desempaquetado *unboxing*: inversa a *boxing*
  - Conversión a string:
    - Conversión *boxing*
    - Aplicación del método `.toString()`

Ej_2_3.java

```
...
int x = 1;
System.out.println("x = " + x);
System.out.println("x = " + new Integer(x).toString() );
```

- Info:

http://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.5

Literales enteros

- Todo literal entero es de tipo `int` (4 bytes). Ejemplos:

Literal	Valor
0	0
4_345_789	4 345 789
-3456	-3456
2147483648	¡¡¡Error en compilacion!!!
0x123	291
0xD4_f2__11_0C	-722 333 428
0b1101_0100_1111_0010__0001_0001_0000_1100	-722 333 428
0b11_010_100_111_100_100_001_000_100_001_100	-722 333 428
032474410414	-722 333 428
-042	-34

- Forzado a tipo `long` (8 bytes). Ejemplos:

Literal	Valor
0L	0
4_345_789L	4 345 789
-3456L	-3456
2147483648L	2 147 483 648
0x123L	291
0xD4_f2__11_0CL	3 572 633 868
0xFFFF_FFFF_D4_f2__11_0CL	-722 333 428
-0xFFFF_FFFF_D4_f2__11_0CL	722 333 428
032474410414L	3 572 633 868
-042L	-34

Literales enteros: tipo forzado con *casting*

- Si tipo entero del literal (`int` o `long`) no es el tipo destino:
  - Casting obligatorio si hay *pérdida* de información

Ej_2_4.java

```
class Ej_2_4 {

    static short f (int a, double b) {
        return (short) (a+b);
    }

    public static void main(String[] args) {


        byte b0 = 34; // no es necesario casting con literal que "cabe"

        //byte b1 = 128; // error, rango byte: -128..127

        byte b2 = (byte)128;
        // (byte)128 -> (byte)0x00_00_00_80 -> 0x80 -> -128 en complemento a 2 en 8 bits

        byte b3 = (byte)896; // (byte)0x380 -> 0x80 -> -128

        System.out.println("b0: " + b0 + ", b2: " + b2 + ", b3: " + b3 );

        float p1 = 2; // !!!
        float p2 = 3.4f;
        x = f( /*(int)*/ p1, p2 );  // Error en compilación
        System.out.println( "x = " + x );

    } // main

} // class Test
```

Literales reales (en coma flotante)

- Todo literal en coma flotante es de tipo `double` (8 bytes). Ejemplos:

Literal	Valor
12_345.342_2e-16	$1{,}23453422 \cdot 10^{-12}$
2.E3	2000,0
-.2E-3	$-2{,}0 \cdot 10^{-4}$
0.	0,0
2.e3D	2000,0
.4E+3d	400,0
1.e39	$1{,}0 \cdot 10^{39}$
1.e-100	$1{,}0 \cdot 10^{-100}$

- Forzado a tipo `float` (4 bytes). Ejemplos:

Literal	Valor
12_345.342_2e-16f	$1{,}2345342 \cdot 10^{-12}$
2.E3F	2000,0
-.2E-3f	$-2{,}0 \cdot 10^{-4}$
0.f	0,0
2.e3F	2000,0
.4E+3f	400,0
1.e39f	Número real de tipo float demasiado grande
1.e-100f	Número real de tipo float demasiado pequeño

- Casting requerido si hay pérdida de información:

Ej_2_5.java

```
//int x = 5.6; // Error en compilación
int x = (int)5.6;
double z = 4.5f;
//float c = 4.5; // Error en compilación
float c = (float) 4.5;
```

Literales de strings

- Un string es una cadena o secuencia de caracteres
- La clase `String` permite almacenar este tipo de información
- También podemos emplear un array de `char`, pero no dispondremos de los métodos específicos para strings
- Un literal de string se forma entrecomillando (con comillas DOBLES) la secuencia de caracteres: `"hola"`
- El operador +
  - Dos funcionalidades:
    - Sumar dos números
    - Concatenar dos strings
  - Determinación de funcionalidad:
    - Si al menos uno de los operandos es un string, concatenará el string en cuestión con el *formato* string del otro operando
    - Sólo si los dos operandos son numéricos, realiza la suma algebraica convencional

Literales de strings (cont.)

- Ejemplos:

```
String s1 = "Hola, s\u00ed se\u00f1or";
String s2 = new String( "Hola, s\u00ed se\u00f1or" );
```

- Un objeto `String` es *inmutable*
- *String pool*

Ej_2_6

```
String s1 = "abc";
String s2 = "abc";
String s3 = new String("abc");

System.out.println( "s1==s2: " + (s1==s2) );
System.out.println( "s1==s3: " + (s1==s3) ); // ?????
System.out.println("s1.equals(s2): " + s1.equals(s2) );
System.out.println("s1.equals(s3): " + s1.equals(s3) );
System.out.println("s2.equals(s3): " + s2.equals(s3) );
```

Literales caracteres y strings

- ATENCIÓN: si uno de los operandos de la operación + es de tipo `char`
  - Si el otro es de tipo numérico (`byte`, `short`, `int`, `long`, `float`, `double`), entonces + suma algebraicamente
  - Si el otro es un string, entonces + concatena el string y la representación del `char` como carácter UNICODE

Ej_2_7

```
String saludo1 = "hola" + ", ¿que tal?";
System.out.println("saludo1: "+ saludo1);

String saludo2 = saludo1 + " ¿Cómo estás?";
System.out.println("saludo2: "+ saludo2);

int n = 3;
String saludo3 = "abcd" + n;
System.out.println("saludo3: "+ saludo3);

int m = 'ñ';
String saludo4 = "abcd" + m;
System.out.println("saludo4: "+ saludo4);

char p = 'ñ';
String saludo5 = "abcd" + p;
System.out.println("saludo5: "+ saludo5);

char q = 'ñ' + 1; // cual sera el siguiente caracter a la eñe???
String saludo6 = "abcd" + q;
System.out.println("saludo6: "+ saludo6);
```

Literales de caracteres y strings (cont.)

- Caracteres: UNICODE (UTF-16)

Ej_2_8.java

```
char a = 'z';
char enye = '\u00f1';
char comilla_simple = '\'';
char comilla_doble  = '\"';
char comilla_doble2  = '"';
System.out.println(a + enye + "\'" + comilla_simple + comilla_doble + comilla_doble2 +
     "\"");

-->   363'"""" ????

System.out.println("h" + a + enye + "\'" + comilla_simple + comilla_doble + comilla_doble2 +
"\"");

-->   hzñ'"""" ???

System.out.println('h' + a + enye + "\'" + comilla_simple + comilla_doble + comilla_doble2 +
"\"");

-->   467'"""" ???
```

Otros literales

- Literales de `boolean`: `true`, `false`
- Literal de objeto: `null`;
- Literal de array: { `<valor>`, `<valor>`, ...}

```
boolean b = false;
Clase objeto = null;
int [ ] v = { 1, 2, 3, 4 };
```

Constantes

Declaración

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

Ejemplo

```
final float ANGULO = 45.f;
```

Nota: por convenio, las constantes se suelen escribir con todo mayúsculas

Tipos compuestos

- Arrays

Declaración

```
<tipo> [ ] ... [ ] <nombre>  = new <tipo> [ num ] ... [ num ];
<tipo> [ ] ... [ ] <nombre>  = { { <valor>, <valor>, ... }, { <valor>, <valor>, ...}, ... };
```

Ejemplos

```
float [ ]  vector = new float[ 4 ];
int [ ][ ] matriz = new int[ 3 ][ 4 ];
int [ ][ ] array = { { 1, 2, 3 }, { 4, 5, 6 } }; // matriz de 2x3
```

- Clases: envoltorios, `String`, colecciones, enumerados, . . .

Ejemplos

```
Integer j = new Integer( 7 );
Boolean b;
b = new Boolean( false );
b = true;
Character c = new Character( 'a' );
String s = "Hola";
String s2 = new String( "Hola" );
ArrayList<Integer> lista = new ArrayList<Integer>( );
lista.add( new Integer( 7 ) );
lista.add( new Integer( 8 ) );
for ( Integer n : lista )
   System.out.println( "Elemento: " + n.toString( ) );
enum laborables { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES };
laborables dia = laborables.MIERCOLES;
```

Ámbito de variables

Ejemplo

```
1
2  {
3     int a;
4     {
5        int b;
6        {
7           ...
8        }
9     }
10 }
11
```

- a sólo es accesible en el intervalo de líneas 3–9
- b sólo es accesible en el intervalo de líneas 5–8

Inicialización

- Variables/objetos locales (a métodos): no son inicializadas automáticamente
- Variables/objetos miembro de un objeto:
  - De clase
  - De instancia

  Éstos son inicializadas automáticamente
  - Variables numéricas/caracteres de tipos primitivos : a 0
  - Variables booleanas: `false`
  - Objetos: `null`

Ej_2_9.java

```
class Ej_2_9 {

    static class OtraClase {
        // Clase vacía: no hace nada. Es correcto
    }

    static int i;
    static float x;
    static char a;
    static boolean z;
    static OtraClase objeto;

    public static void main(String[] args) {

        System.out.println("i: " + i );
        System.out.println("x: " + x );
        System.out.println("a: " + a );
        System.out.println("(short)a: " + (short)a );
        System.out.println("z: " + z );
        System.out.println("objeto: " + objeto );

        int local_i;
        // Error: uso sin inicializar!!!
        //System.out.println("local_i: " + local_i);

    } // main

} // Ej_2_9
```

Índice

3 Expresiones

Comentarios

Ejemplo

```
// Comentario de línea (hasta fin de línea)

/* Comentario de varias líneas
   ...
   Última línea de comentario */

/** Comentario de javadoc: generación de documentación
    para Java */
```

Operadores

Tipo	Operador(es)
Asignación	`=`
Aritméticos	`+, -, *, /, %, ++, --, +=, -=, *=, /=, %=,`
Comparación	`==, !=, <, <=, >, >=, instanceof`
Lógicos	`!, &&, ||`
Manejo de bits	`&, |, ~, ^, <<, >>, >>>, &=, |=, ^=, <<=, >>=, >>>=`
Conversión *(casting)*	`(tipo)`
Ternario	`( )? :`

Precedencia

Operadores	Precedencia	Orden
Paréntesis	( )	–
Postfijos	var++, var--	⇒
Unarios	++var, --var, +expr, -expr, ~, !	⇒
Multiplicativos	*, /, %	⇒
Aditivos	+, -	⇒
Desplazamiento	<<, >>, >>>	⇒
Relacionales	<, >, <=, >=, instanceof	⇒
Igualdad	==, !=	⇒
Y bit a bit	&	⇒
O exclusiva bit a bit	^	⇒
O bit a bit	|	⇒
Y lógica	&&	⇒
O lógica	||	⇒
Ternario	( )? :	⇒
Asignación	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=	⇐

Criterios:

- Operadores en parte superior se evalúan antes que los de parte inferior
- Operadores en la misma fila: igual precedencia
  - Orden determinado por aparición en expresión
    - Todos los operadores (salvo los de asignación): de izquierda a derecha
    - Todos los operadores de asignación: de derecha a izquierda

Ej_3_1.java

```
int x;

x = 1;
x += 6 + 3 * 5 % 9 / 3 << 2; // x = 33;

System.out.println( "x = " + x );

//////////

x = 1;
x = x + ( ( 6 + ( ( (3 * 5) % 9 ) / 3 ) ) << 2 ); // x = 33;

System.out.println( "x = " + x );
```

Índice

4 Sentencias
- Selección
- Sentencias iterativas
- Saltos

Índice

4 Sentencias
- Selección
- Sentencias iterativas
- Saltos

Selección simple

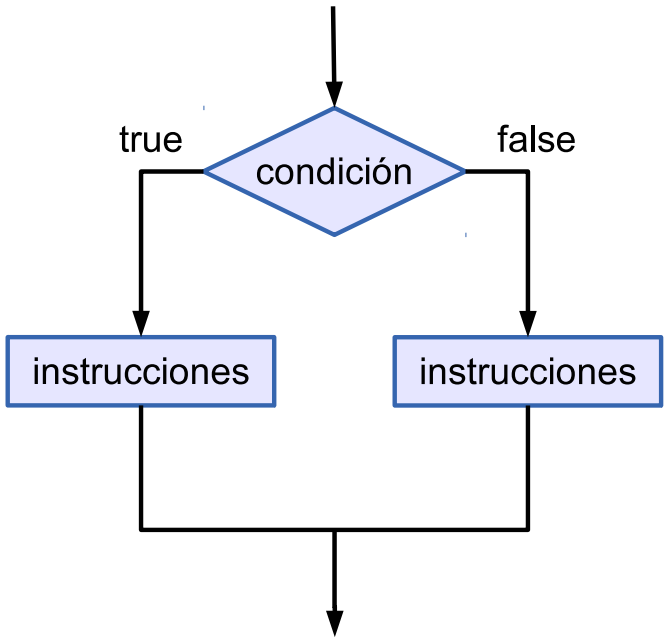
`if`:

if

```
if ( condición ) {
   instrucciones;
} else {
   instrucciones;
}
```

Ejemplo

```
if ( i > 3 ) {
   x = 6;
} else {
   x = 7;
}
```



Selección múltiple

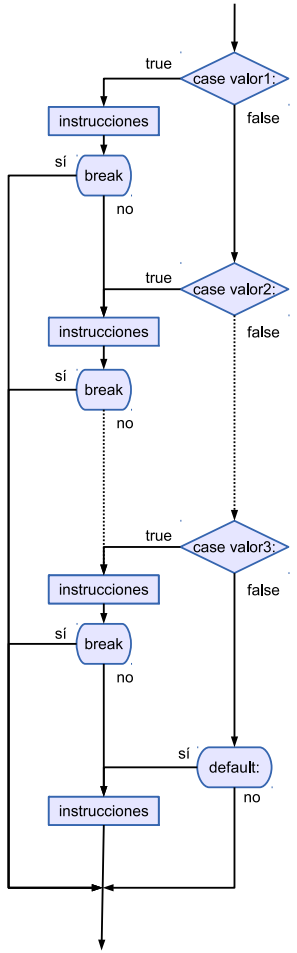
`switch`:

switch

```
switch ( variable ) {
   case valor1: [ instrucciones; ] [ break; ]
   case valor2: [ instrucciones; ] [ break; ]
   case valor3: [ instrucciones; ] [ break; ]
   ...
   [ default: [ instrucciones; ] ]
}
```

Ejemplo

```
switch ( x ) {
   case 0: y = 1; break;
   case 1: y = 2; z = 3;
   case 2:
   case 3: w = 5; break;
   default: z = -1;
}
```



Índice

4 Sentencias
- Selección
- Sentencias iterativas
- Saltos

Con condición inicial

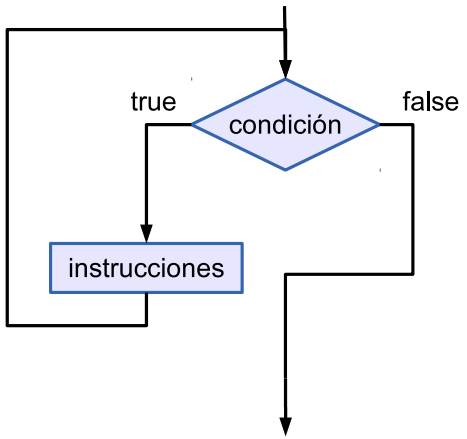
`while`:

while

```
while ( condicion ) {
   instrucciones;
}
```

Ejemplo

```
while ( x > 7 ) {
   z++;
   x -= 2;
}
```



Con condición final

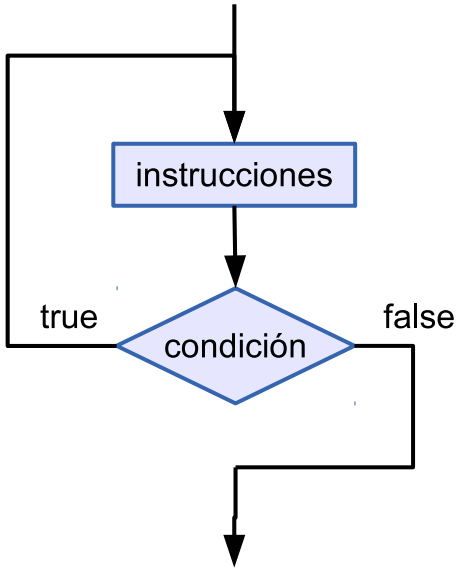
`do-while`:

do-while

```
do {
   instrucciones;
} while ( condicion );
```

Ejemplo

```
do {
   x += 23;
   y++;
} while ( x < 144 );
```



Bucles

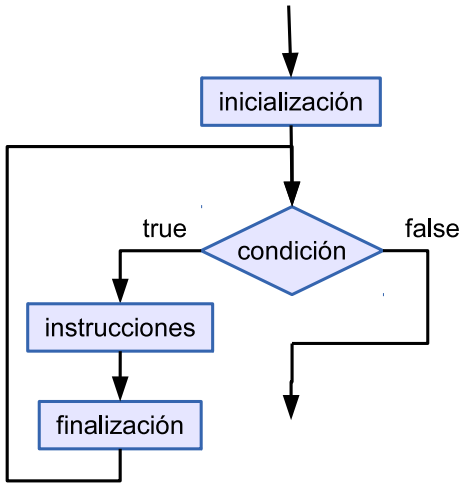
for:

for

```
for ( inicialización; condición; finalización ) {
   instrucciones;
}
```

Ejemplo

```
for ( int i = 0; i < 20; i++ ) {
   j += i;
}
```



Iteradores

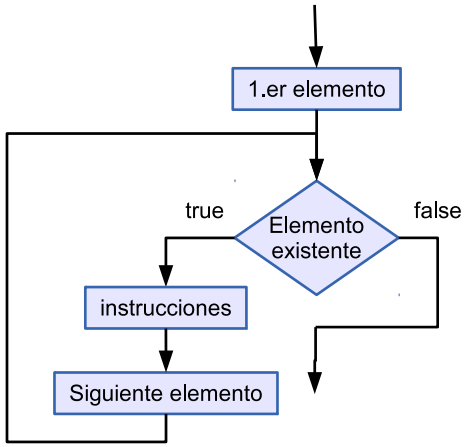
- for-iterator:

for

```
for ( <tipo> variable : coleccion ) {
   instrucciones;
}
```

Ejemplo

```
ArrayList<Integer> lista = new ArrayList<Integer>( );
lista.add( new Integer( 7 ) );
lista.add( new Integer( 8 ) );
for ( Integer n : lista ) {
   System.out.println( "Elemento: " + n.toString( ) );
}
```



- ...

Índice

4 Sentencias
- Selección
- Sentencias iterativas
- Saltos

Incondicionales

- `break;`: fuerza terminación de bucle o rama de switch.
- `break etiqueta;`: fuerza terminación de bucle encabezado con `etiqueta:`.
- `continue;`: fuerza nueva iteración de bucle
- `continue etiqueta;`: fuerza nueva iteración del bucle encabezado con `etiqueta:`.
- `return [<valor>];`: sale de la función (y retorna valor)
- `exit( codigo_retorno );`: sale del programa (y devuelve código)

Ejemplo saltos

```
...
switch( x ) {
   case 1: instrucciones...; break;
   case 2:
   case 3: mas_instrucciones...; break;
   default: instrucc.
}

int z = 0;
for ( int h = 0; h < 24; h++ ) {
   for( int i = 0; i < 1000; i++ ) {
      z = z + i*2;
      if ( z > 543 ) {
         break;
      } else if ( z > 124 ) {
         continue;
      }
   }
}
```

```
z = 0;
etiqueta1:
for ( int j = 0; j < 200; j++ ) {
   z = z + j;
   etiqueta2:
   for( int i = 0; i < 1000; i++ ) {
      z = z + i*2;
      if ( z > 543) {
         break etiqueta1;
      } else if ( z > 124 ) {
         continue etiqueta2;
      } else {
         return 4.3;
      }
   }
}
...
```

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

- Las **clases** son el mecanismo que posee Java para describir
  - Nuevos tipos o estructuras de datos. **Atributos**: campos que posee la estructura de los datos.
  - Las operaciones (*funciones* que cuando pertenecen a un objeto se denominan **métodos**) que se pueden realizar con los atributos
  - **Miembros**: atributos y métodos
- **Objeto**: memoria que contiene todos los atributos descritos en la clase, junto con sus métodos.
  - Un objeto es una **instancia** de una clase. Se crea con el operador `new` (se ejecuta un **constructor**)
    `Clase objeto = new Clase(...);`
  - Acceso a miembros: `objeto.atributo`, `objeto.metodo(...)`
- Los objetos se manipulan con referencias:
  - Variable que apunta a un objeto

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Métodos

- Toda **función miembro de un objeto** se denomina **método**
- Sintaxis:

Sintaxis de definición de método

```
[modificador(es)] [tipo_retorno] nombre_metodo_verbo ( [parametros] ) {
    ...
    Desarrollo
    ...
    [return valor]
}
```

- El tipo de la variable/objeto devuelto con `return` debe coincidir con `[tipo_retorno]` especificado.
- Si no se devuelve nada, hay que indicar como tipo de retorno `void`

Ejemplo

```
public static double calcular_hipotenusa( double cateto1, double cateto2 ) {
    double resultado =  Math.sqrt( cateto1*cateto1 + cateto2*cateto2 );
    return resultado;
}
```

Métodos y su sobrecarga

- Sobrecarga:
  - Presencia de dos o más métodos con el mismo nombre y diferenciados a partir de tipos distintos en **parámetros**.
  - Selección automática de la sobrecarga en funcion de tipo de argumentos
  - El tipo devuelto por el método no interviene en la elección de la sobrecarga
  - Se pueden sobrecargar tanto métodos como constructores

Sobrecarga

```
[modificador] <tipo> <nombre_metodo>( <parametros_1> ) {
   ...
}

[modificador] <tipo> <nombre_metodo>( <parametros_2> ) {
   ...
}
```

Ej_5_1.java

```
class Ej_5_1 {

    static double maximo( double a, double b ) {
        return ( ( a>=b )? a : b );
    }

    static double maximo( double a, double b, double c ) {
        return maximo( maximo(a,b), c );
    }

    static String maximo( String dia1, String dia2 ) {
        final String [ ] semana = { "lunes", "martes", "miercoles", "jueves",
                                    "viernes", "sabado", "domingo" };
        int dia1_idx = 0;
        int dia2_idx = 0;

        for (int i = 0; i < 7; i++ ) {
            if (dia1.equalsIgnoreCase(semana[i])) {
                dia1_idx = i;
                break;
            }
        }
...
```

Ej_5_1.java (cont.)

```
...
        for (int i = 0; i < 7; i++ ) {
            if (dia2.equalsIgnoreCase(semana[i])) {
                dia2_idx = i;
                break;
            }
        }

        return semana[ (int)maximo(dia1_idx, dia2_idx) ];

    } // static String maximo


    public static void main (String[] args) {
        System.out.println( "maximo( 3, 9.5 ) = " + maximo(3, 9.5) );
        System.out.println( "maximo( 12.1, 9.5, 3 ) = " + maximo(12.1, 9.5, 3) );
        System.out.println( "maximo( \"martes\", \"sabado\" ) = "
                            + maximo("martes", "sabado") );
    }

}
```

No existen métodos locales

- El siguiente ejemplo NO compilará:

```
class clase {
    void m() {
        void n() {
            System.out.println("Hello World");
        }
        n();
    }
}
```

- Alternativas válidas:
  - Empleo de funciones lambda
  - Usando subclases anónimas
  - Usando clases locales (éstas sí que son posibles)

Lista variable de argumentos en llamadas a métodos

- Parámetros:

  `metodo( Tipo... parametro )`

  equivalente a

  `metodo( Tipo[] parametro )`
- Llamable de dos formas:
  - Lista variable de parámetros de clase/tipo `Tipo`
  - Array de clase/tipo `Tipo`
- No se deben añadir parámetros después de una lista variable

Ej_5_2

```
class Ej_5_2 {

    // Aquí nombres es equivalente un array de Strings
    public static void metodo( String ... nombres ) {

        System.out.println( "Hay un total de " + nombres.length + " nombres" );

        for( String valor : nombres ) {
            System.out.println("nombre: " + valor);
        }

        // Alternativa convencional recorrido array
        for ( int i = 0; i < nombres.length; i++ ) {
            System.out.println( "-nombre: " + nombres[i] );
        }
    }



    public static void main (String[] args) {

        metodo( "a", "b", "c" );

        String [] x = {"d", "e", "f", "g"};
        metodo(x);

        metodo(new String[] {"h", "i"}); // argumento anónimo

    } // main

}  // Ej_5_2
```

Método constructor de la clase

- Método ejecutado automáticamente al instanciar una clase y obtener un objeto
- Mismo nombre que clase
- Puede haber **sobrecarga** de constructores
- Pueden tener cualquier **modificador de acceso** (se estudiará posteriormente)
- Un constructor puede llamar a otro constructor de su misma clase: `this(·)` (primera instrucción). Selección según **sobrecarga**
- El constructor puede llamar al constructor de la clase padre: `super(·)` con la sobrecarga oportuna. Atención: debe ser la primera instrucción del constructor.
- Si no llama a su padre con al sobrecarga oportuna, entonces se llama implicitamente a la sobrecarga del constructor `super()`. Si no existe en el padre: error en tiempo de compilación.

Ejemplo

miAlgebra\NumeroComplejo.java

```
package miAlgebra;

class NumeroComplejo {

    private double parteReal;       // parte real del número complejo
    private double parteImaginaria; // parte imaginaria del número complejo

    // Construye un número complejo
    public NumeroComplejo( double r, double i) {
        parteReal = r;
        parteImaginaria = i;
    }

    // Crea un número complejo con solo parte real;
    // parte imaginaria nula
    public NumeroComplejo( double r ) {
        this(r, 0.);
    }

    // Devuelve parte real del número complejo
    public double getParteReal( ) {
        return parteReal;
    }

    // Devuelve parte imaginaria del número complejo
    public double getParteImaginaria( ) {
        return parteImaginaria;
    }

    // Suma "este" numero complejo con el del parámetro.
    // El resultado de la suma se queda "aquí" y también
    // se devuelve o "retorna"
    public NumeroComplejo sumar( NumeroComplejo c) {
        parteReal += c.getParteReal( );
        parteImaginaria += c.getParteImaginaria( );
        return this;
    }

    // Pendiente de realizar
    public NumeroComplejo restar( NumeroComplejo c) {
        return null; // acción provisional
    }

    // Pendiente de realizar
    public String toString() {
        //return "" + parteReal + " " + parteImaginaria + "j";
        return null; // accion provisional
    }

} // class NumeroComplejo
```

miAlgebra\TestNumeroComplejo.java

```
package miAlgebra;

class TestNumeroComplejo {

    public static void main (String[] args) {

        NumeroComplejo c;
        c = new NumeroComplejo( 4.7, -11.25 );
        System.out.println("c: " + c);

    } // main

} // TestNumeroComplejo
```

Identificación de:

- **Paquete** al que pertenece la clase (se estudiará posteriormente)
- Nombre de la **clase** (mayúsculas: sustantivo)
- **Atributos** o **variables**
- **Constructor(es)**
- **Métodos** (minúsculas: verbos o acción)

La compilación y ejecución exige ciertos detalles...

Conflictos de nombres

- Mismo nombre en parámetro de método/constructor que miembro.
- Ambigüedad resuelta con prefijo `this`.

```
class Clase1 {

    int x; // miembro de Clase1

    public Clase1 (int x) { // parámetro del constructor
           // Aquí la intención es asignar a la variable miembro x,
           // el valor del parámetro x.
           //       x = x no haría nada
           // ¿Quién es quién?:
           // Aquí en el constructor, x es el parámetro, haciéndole sombra
           // al miembro x de Clase1
           // Solución:
           this.x = x;
    }
}
```

Ejemplo: creación de referencia a objeto

miAlgebra\NumeroComplejo.java

```
package miAlgebra;

class NumeroComplejo {

    private double parteReal;       // parte real del número complejo
    private double parteImaginaria; // parte imaginaria del número complejo

    // Construye un número complejo
    public NumeroComplejo( double r, double i) {
        parteReal = r;
        parteImaginaria = i;
    }

    // Crea un número complejo con solo parte real;
    // parte imaginaria nula
    public NumeroComplejo( double r ) {
        this(r, 0.);
    }

    // Devuelve parte real del número complejo
    public double getParteReal( ) {
        return parteReal;
    }

    // Devuelve parte imaginaria del número complejo
    public double getParteImaginaria( ) {
        return parteImaginaria;
    }

    // Suma "este" numero complejo con el del parámetro.
    // El resultado de la suma se queda "aquí" y también
    // se devuelve o "retorna"
    public NumeroComplejo sumar( NumeroComplejo c) {
        parteReal += c.getParteReal( );
        parteImaginaria += c.getParteImaginaria( );
        return this;
    }

    // Pendiente de realizar
    public NumeroComplejo restar( NumeroComplejo c) {
        return null; // acción provisional
    }

    // Pendiente de realizar
    public String toString() {
        //return "" + parteReal + " " + parteImaginaria + "j";
        return null; // accion provisional
    }

} // class NumeroComplejo
```

miAlgebra\TestNumeroComplejo.java

```
package miAlgebra;

class TestNumeroComplejo {

    public static void main (String[] args) {

        NumeroComplejo c;
        c = new NumeroComplejo( 4.7, -11.25 );
        System.out.println("c: " + c);

    } // main

} // TestNumeroComplejo
```

c

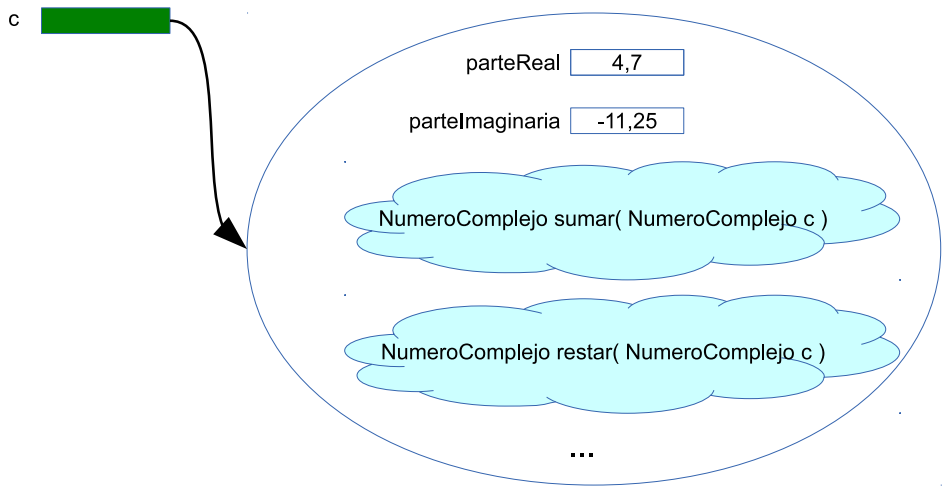
Ejemplo: creación de objeto (instancia de clase)

miAlgebra\NumeroComplejo.java

```
package miAlgebra;

class NumeroComplejo {

    private double parteReal;       // parte real del número complejo
    private double parteImaginaria; // parte imaginaria del número complejo

    // Construye un número complejo
    public NumeroComplejo( double r, double i) {
        parteReal = r;
        parteImaginaria = i;
    }

    // Crea un número complejo con solo parte real;
    // parte imaginaria nula
    public NumeroComplejo( double r ) {
        this(r, 0.);
    }

    // Devuelve parte real del número complejo
    public double getParteReal( ) {
        return parteReal;
    }

    // Devuelve parte imaginaria del número complejo
    public double getParteImaginaria( ) {
        return parteImaginaria;
    }

    // Suma "este" numero complejo con el del parámetro.
    // El resultado de la suma se queda "aquí" y también
    // se devuelve o "retorna"
    public NumeroComplejo sumar( NumeroComplejo c) {
        parteReal += c.getParteReal( );
        parteImaginaria += c.getParteImaginaria( );
        return this;
    }

    // Pendiente de realizar
    public NumeroComplejo restar( NumeroComplejo c) {
        return null; // acción provisional
    }

    // Pendiente de realizar
    public String toString() {
        //return "" + parteReal + " " + parteImaginaria + "j";
        return null; // accion provisional
    }

} // class NumeroComplejo
```

miAlgebra\TestNumeroComplejo.java

```
package miAlgebra;

class TestNumeroComplejo {

    public static void main (String[] args) {

        NumeroComplejo c;
        c = new NumeroComplejo( 4.7, -11.25 );
        System.out.println("c: " + c);

    } // main

} // TestNumeroComplejo
```



Clonación de objetos por constructor de copia

Creación de un objeto por clonación mediante constructor de copia *(copy constructor)*:

miAlgebra2\NumeroComplejo2.java

```
package miAlgebra2;

class NumeroComplejo2 {

    // parte real del número complejo
    private double parteReal;

    // parte imaginaria del número complejo
    private double parteImaginaria;

    // Constructor de copia
    public NumeroComplejo2( NumeroComplejo2 x) {
        this( x.getParteReal(), x.getParteImaginaria() );
        // Alternativa equivalente
        // this( x.parteReal, x.parteImaginaria );
    }

    ...

    // Resto idem que miAlgebra\ NumeroComplejo.java

}
```

miAlgebra2\TestNumeroComplejo2.java

```
package miAlgebra2;

class TestNumeroComplejo2 {

    public static void main (String[ ] args) {
        NumeroComplejo2 c;
        c = new NumeroComplejo2( 4.7, -11.25 );
        System.out.println("c: " + c);

        NumeroComplejo2 d = new NumeroComplejo2( c );
        System.out.println( "d = " + d );

    } // main

} // TestNumeroComplejo2
```

Destrucción de un objeto: recolector de basura

- Cuando finaliza el ámbito de un objeto (no hay referencias a él)
  - El objeto es eliminado, liberando memoria
- Eliminación explícita: `.finalize()`

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Diferencia entre variables y objetos

- Variables:
  - Asignación con =
  - Comparación de valores con ==
- Objetos:
  - Nombre de un objeto: contiene referencia a memoria del objeto
  - Operador ==: saber si dos objetos tienen la misma referencia en memoria
  - Método `.equals(<objeto>)` para saber si sus atributos tiene los mismos valores:
    - Se deberá implementar para clases realizadas por el programador
    - Si no se implementa, suele devolver mismo resultado que ==
    - Relacionado con función `hasCode()` de todo objeto
  - Operador = asigna referencias (serán el mismo objeto)
  - Método `.clone()` para crear una copia idéntica (objetos distintos con idéntico contenido). Obsoleto y conflictivo: es preferible constructor de copia

Método .equals() (...y .hashcode())

Cuidado con método .equals()

Ej_5_3.java

```
class Ej_5_3 {

    static class Clase_A {

        int i;

        public Clase_A( int i ) {
            this.i = i;
        }

        public String toString() {
            return "valor (clase A) = " + i;
        }

        // metodo equals heredado de Object

    }

    static class Clase_B {

        int j;

        public Clase_B( int j ) {
            this.j = j;
        }

        public String toString() {
            return "valor (clase B) = " + j;
        }

        // metodo equals heredado de Object sobreescrito
        // por este metodo nuevo
        public boolean equals( Clase_B x ) {

            if (x.j == j) {
                return true;
            } else {
                return false;
            }

        }

        // Idealmente debería sobreescribirse adicionalmente un método
        // denominado hashcode...

    }
...
```

Ej_5_3.java (cont.)

```
...
    public static void main (String[] args) {

        Clase_A a = new Clase_A(3);
        Clase_A a2 = new Clase_A(3);

        Clase_B b = new Clase_B(5);
        Clase_B b2 = new Clase_B(5);

        System.out.println( "a: " + a + ", a2: " + a2);
        System.out.println( "b: " + b + ", b2: " + b2);

        System.out.println( "a==a2 : " + a.equals(a2) );
        System.out.println( "b==b2 : " + b.equals(b2) );
    }
}
```

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Clases enumeradas

Ej_5_4.java

```
class Ej_5_4 {

    enum laborables { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES };

    public static void main (String [] args) {

        /* VIERNES incorrecto, Laborables.VIERNES correcto */
        laborables mejorDiaLaborable = laborables.VIERNES;

        System.out.println("Mejor día laborable: " + mejorDiaLaborable);

        int i = 1;
        for (laborables d: laborables.values()) {
            System.out.println("Laborable " + (i++) + ": " + d);
        }

    } // main

} // Test
```

Ej_5_5.java

```
class Ej_5_5 {

    enum Optimismo { EXCELENTE, BIEN, REGULAR, MAL }

    enum TipoDia { LABORABLE, FESTIVO }

    enum Dias {

        /* Secuencia separada por comas. Acaba con punto y coma */
        LUNES     (TipoDia.LABORABLE, Optimismo.MAL),
        MARTES    (TipoDia.LABORABLE, Optimismo.REGULAR),
        MIERCOLES (TipoDia.LABORABLE, Optimismo.BIEN),
        JUEVES    (TipoDia.LABORABLE, Optimismo.BIEN),
        VIERNES   (TipoDia.LABORABLE, Optimismo.EXCELENTE),
        SABADO    (TipoDia.FESTIVO,   Optimismo.EXCELENTE),
        DOMINGO   (TipoDia.FESTIVO,   Optimismo.REGULAR);

        private Optimismo nivelOptimismo;
        private TipoDia tipoDia;

        /* No se pueden crear mas objetos que los definidos
        anteriormente, pero con este constructor se "autoconstruyen"*/
        Dias(TipoDia tipoDia, Optimismo nivelOptimismo) {
            this.tipoDia = tipoDia;
            this.nivelOptimismo = nivelOptimismo;
        }


...
```

Ej_5_5.java (cont.)

```
...

        public TipoDia getTipoDia() {
            return tipoDia;
        }

        public Optimismo getNivelOptimismo() {
            return nivelOptimismo;
        }

    } // Dias


    public static void main (String [] args) {

        Dias mejorDia = Dias.SABADO;

        System.out.println("Mejor día de la semana: " + mejorDia);
        switch (mejorDia) {
            /* Dias.VIERNES: ¡incorrecto!; VIERNES: ¡correcto! */
            case VIERNES:
            case SABADO:
            case DOMINGO:
                    System.out.println("Normal...");
                break;
            default:
                System.out.println("Pues no lo entiendo...");

        }

        int i = 1;
        for ( Dias d: Dias.values() ) {
            System.out.println("Laborable " + (i++)
                + ", tipo de dia: " + d.getTipoDia()
                + ", optimismo: " + d.getNivelOptimismo());
        }

    } // main

} // Test2
```

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Paquetes *(packages)*

- Agrupación de clases afines: equivalente a librería de otros lenguajes
- Una clase pertenece a un paquete y puede usar otras clases definidas en el mismo o en otros paquetes
- Los `package` también delimitan el espacio de nombres
  - El nombre de una clase debe ser único en el `package` donde se define
  - Dos clases con el mismo nombre, pero distinto *package* pueden coexistir
- Una clase se declara perteneciente a un `package` con cláusula `package nombre_package;`.
  - Debe ser la primera sentencia del fichero fuente.
  - La clase declarada en ese archivo pertenece a ese `package`

Package

```
package miPackage;
...
class MiClase {
 ...
}
```

La clase `MiClase` pertenece al paquete `miPackage`

- Si no hay cláusula `package`, la clase pertenece a un paquete por defecto sin nombre. Suele ser conveniente poner nombre de paquete.

Cláusula import

- Indicación al compilador de Java de dónde encontrar la definición de ciertas clases
- Cuando se referencia una clase, por defecto se supone que está dentro del mismo package

Ejemplo

```
package aritmetica;
...
class OperacionAditiva {
   NumeroComplejo x;
   ...
}
```

Ejemplo

```
package aritmetica;
...
class NumeroComplejo {
   NumeroComplejo(...) { ... };
   ...
}
```

Cláusula import

- Si no está en el mismo package hay que indicar su package explícitamente. Si la clase NumeroComplejo está en el paquete Números, entonces alternativas:

Ejemplo: paquete explícito con la clase

```
package aritmetica;
...
class OperacionAditiva {
   Numeros.NumeroComplejo x;
   ...
}
```

Ejemplo: importación de la clase de interés

```
package aritmetica;
import Numeros.NumeroComplejo;
...
class OperacionAditiva {
   NumeroComplejo x;
   ...
}
```

Ejemplo: importación de todas las clases del package Numeros

```
package aritmetica;
import Numeros.*;
...
class OperacionAditiva {
   NumeroComplejo x;
   ...
}
```

Nombres, subpaquetes y ubicaciones

- Los nombres puede ser compuestos separados por puntos (como una URL) y suelen ir en **minúsculas**:
  - Ejemplo: `package aritmetica.operaciones.relacionales;`
  - Denominación:
    - `aritmetica`: paquete
    - `operaciones`: subpaquete del paquete `aritmetica`
    - `relacionales`: subpaquete del paquete `aritmetica.operaciones`
- Para evitar conflictos de nombres:
  - Se sugiere que **ningún** paquete escrito por el usuario tenga como prefijo `java` ó `javax`
- La JVM buscará la clase en cuestión en el directorio `aritmetica/operaciones/relacionales`
  - Si una clase no pertenece a ningún paquete, se busca en el directorio actual de ejecución

Ejemplo sencillo

```
...> dir

06/02/2017  11:59    <DIR>          .
06/02/2017  11:59    <DIR>          ..
06/02/2017  11:49               189 Ejemplo.java
06/02/2017  11:51               731 NumeroComplejo.java

...> javac NumeroComplejo.java

...> dir

06/02/2017  12:00    <DIR>          .
06/02/2017  12:00    <DIR>          ..
06/02/2017  11:49               189 Ejemplo.java
06/02/2017  12:00             1.084 NumeroComplejo.class
06/02/2017  11:51               731 NumeroComplejo.java

...> javac Ejemplo.java
Ejemplo.java:4: error: cannot find symbol
                NumeroComplejo c;
                ^
  symbol:   class NumeroComplejo
  location: class Ejemplo
Ejemplo.java:5: error: cannot find symbol
                c = new NumeroComplejo( 4.7, -11.25 );
                        ^
  symbol:   class NumeroComplejo
  location: class Ejemplo
2 errors

...> mkdir miAlgebra

...> move NumeroComplejo.class miAlgebra

...> dir

06/02/2017  12:16    <DIR>          .
06/02/2017  12:16    <DIR>          ..
06/02/2017  11:49               189 Ejemplo.java
06/02/2017  12:16    <DIR>          miAlgebra
06/02/2017  11:51               731 NumeroComplejo.java
```

```
...> javac Ejemplo.java

...> dir

06/02/2017  12:18    <DIR>          .
06/02/2017  12:18    <DIR>          ..
06/02/2017  12:18               724 Ejemplo.class
06/02/2017  11:49               189 Ejemplo.java
06/02/2017  12:16    <DIR>          miAlgebra
06/02/2017  11:51               731 NumeroComplejo.java

...> java Ejemplo
Error: no se ha encontrado o cargado la clase principal Ejemplo

...> move Ejemplo.class miAlgebra

...> dir miAlgebra

06/02/2017  12:19    <DIR>          .
06/02/2017  12:19    <DIR>          ..
06/02/2017  12:18               724 Ejemplo.class
06/02/2017  12:00             1.084 NumeroComplejo.class

...> java miAlgebra\Ejemplo
Error: no se ha encontrado o cargado la clase principal miAlgebra\Ejemplo

...> java miAlgebra.Ejemplo
c=4.7-11.25j
```

- ¿Qué ocurriría si `Ejemplo.java` no perteneciera al paquete `miAlgebra`?
- Ubicación de los ficheros `.java` en directorio `miAlgebra`: compilación voraz...

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Tipos de modificadores

- Modificación que puede hacerse a una clase, un método o un atributo respecto a:
  - El tipo permitido de acceso a dicho recurso (clase, método o atributo): **modificadores de control de acceso**
  - Otras características de dicho recurso: **modificadores de no acceso**

Modificadores de control de acceso

- `public`: recurso accesible desde cualquier clase
- `protected`: recurso accesible desde la propia clase, desde cualquier clase extendida de ésta y desde cualquier clase perteneciente al paquete
- Sin modificador (defecto): recurso accesible desde la propia clase y desde cualquier otra clase perteneciente al mismo paquete. Denominado modificador de acceso *por paquete*
- `private`: recurso accesible sólo desde la propia clase

Recurso: dependiendo del modificador, aplicable a clases, métodos o atributos.

	Desde la misma clase	Desde otra clase del mismo paquete	Desde una subclase de otro paquete	Desde otra clase de otro paquete
`public`	X	X	X	X
`protected`	X	X	X	
	X	X		
`private`	X			

Modificadores de no acceso

- `static`
- `final`
- `abstract`
- `synchronized`
- `volatile`
- `strictfp`

Modificadores (no acceso): `static`

- `static`: para métodos, clases internas o atributos utilizables sin instanciar clase (para métodos o variables de clase)
  - Variables estáticas (variables de clase):
    - Existen independientemente de las instancias (no es necesario instanciar un objeto).
    - Existe sólo una copia de la variable estática entre todos los objetos.
  - Clases internas:
    - Existen independientemente de las instancias (no es necesario instanciar un objeto).
  - Métodos estáticos:
    - Existen independientemente de las instancias (no es necesario instanciar un objeto).

  Acceso: `Clase.variable`, `Clase.Clase_Interna` ó `Clase.metodo(·)`

  Importante: un método estático sólo puede acceder a atributos o métodos estáticos

Ejemplos de variables, clases y métodos estáticos

miAlgebra3\NumeroComplejo3.java

```
package miAlgebra3;

class NumeroComplejo3 {

    private double parteReal;       // parte real del número complejo
    private double parteImaginaria; // parte imaginaria del número complejo


    // Atributo estático
    static NumeroComplejo3 CERO = new NumeroComplejo3(0.);

    // Método estático
    static String descripcion() {
        return "Un número complejo está compuesto por...";
    }

    // Clase estática
    static class Polar {

        private double modulo;
        private double argumento;

        public Polar(NumeroComplejo3 a) {
            modulo = Math.sqrt(  a.getParteReal()*a.getParteReal()
                               + a.getParteImaginaria()*a.getParteImaginaria() );
            argumento = Math.atan(a.getParteImaginaria() / a.getParteReal() );
        }

        public double getModulo() {
            return modulo;
        }

        public double getArgumento() {
            return argumento;
        }

        public String toString() {
            return "" + modulo + "|" + argumento;
        }

    } // clase Polar

   // Constructor de copia
    public NumeroComplejo3( NumeroComplejo3 c) {
        this( c.getParteReal(), c.getParteImaginaria() );
        // equivalente a pesar de ser privados
        // this( c.parteReal, c.parteImaginaria );
    }

    // Construye un número complejo
    public NumeroComplejo3( double r, double i) {
        parteReal = r;
        parteImaginaria = i;
    }

    // Crea un número complejo con solo parte real;
    // parte imaginaria nula
    public NumeroComplejo3( double r ) {
        this(r, 0.);
    }
...
```

miAlgebra3\NumeroComplejo3.java (cont.)

```
...

    // Devuelve parte real del número complejo
    public double getParteReal( ) {
        return parteReal;
    }

    // Devuelve parte imaginaria del número complejo
    public double getParteImaginaria( ) {
        return parteImaginaria;
    }

    // Suma "este" numero complejo con el del parámetro.
    // El resultado de la suma se queda "aquí" y también
    // se devuelve o "retorna"
    public NumeroComplejo3 sumar( NumeroComplejo3 c) {
        parteReal += c.getParteReal( );
        parteImaginaria += c.getParteImaginaria( );
        return this;
    }

    // Pendiente de realizar
    public NumeroComplejo3 restar( NumeroComplejo3 c) {
        return null; // acción provisional
    }

    // Pendiente de realizar
    public String toString() {
        return "" + parteReal + " " + parteImaginaria + "j";
        //return null; // accion provisional
    }

} // class NumeroComplejo3
```

miAlgebra3\TestNumeroComplejo3.java

```
package miAlgebra3;

class TestNumeroComplejo3 {

    public static void main (String[] args) {
        NumeroComplejo3 c;
        c = new NumeroComplejo3( 4.7, -11.25 );
        System.out.println("c: " + c);

        NumeroComplejo3 d = new NumeroComplejo3( c );
        System.out.println( "d = " + d );

        System.out.println("Descripción: " + NumeroComplejo3.descripcion());

        System.out.println("CERO: " + NumeroComplejo3.CERO );

        NumeroComplejo3.Polar x = new NumeroComplejo3.Polar(d);
        System.out.println("x: " + x );

    } // main

} // TestNumeroComplejo3
```

Modificadores (no acceso): `final`

- `final`: para indicar que ya no pueden sobreescribirse (para clases, métodos o atributos)
  - En propiedades (variables, atributos o campos de una clase):
    - Ya no podrá modificarse el valor
    - Existe sólo una copia de la variable `final` entre todos los objetos.
  - Métodos `final`:
    - No podrán sobreescribirse (`@Override`)
  - Clases `final`:
    - No pueden heredarse (extenderse)

Modificadores (no acceso): `abstract`

- `abstract`: para crear métodos abstractos (sin implementar) o especificar que la clase (abstracta) contiene métodos abstractos
  - Métodos:
    - Las subclases deben proporcionar la implementación: la clase abstracta puede ya emplear estos métodos (aunque estén sin implementar).
  - Clases:
    - Denota que contiene métodos abstractos o se desea que no se pueda instanciar de forma directa (sí potencialmente através de alguna subclase).
    - No se pueden instanciar, sólo extender

Modificadores (no acceso): `synchronized`, `volatile` y `strictfp`

- `synchronized`: ejecución de método `synchronized` bloquea acceso a objeto
  - Utilizado con Threads
- `volatile`: control de acceso y actualización en variables compartidas por varios threads
- `strictfp`: cumplimiento de estándar de aritmética en coma flotante IEEE-754

Modificadores: ejemplo genérico

Clase

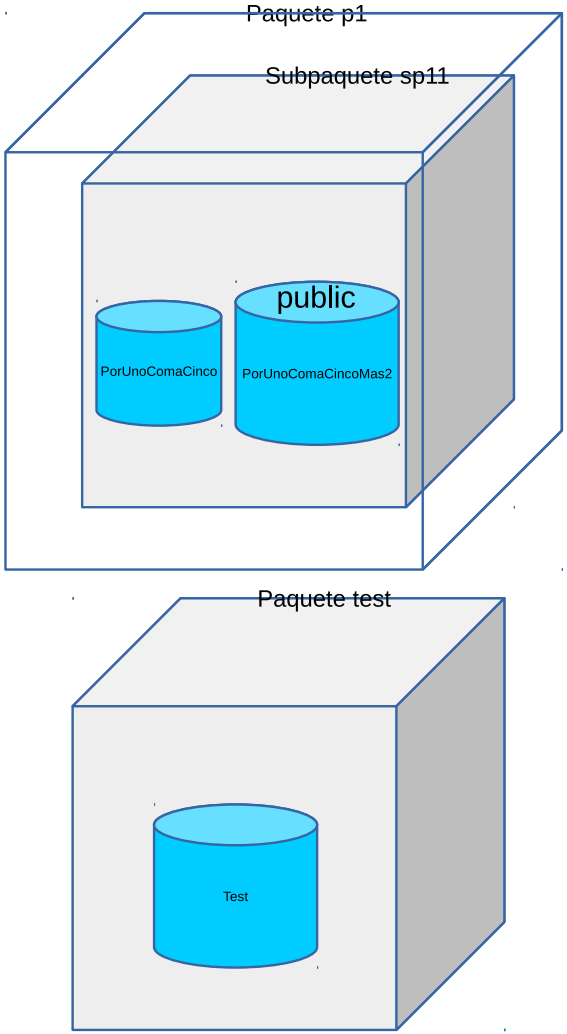
```
[ public ] [ abstract ] class <Clase> [ extends <Clase_padre> ] [ implements <Interfaz_1> [, <Interfaz_2> [...] ] ]  {

   // Declaración de atributos o propiedades
   [ acceso ] [ final ] [ static ] <tipo> <nombre_atributo> [ = valor ];

   ...

   // Declaración de constructor(es)
   [ acceso ] <Clase> ( <argumentos ) {
      <instrucciones>;
   } // Constructor

   ...

   // Declaración de clase interna
   [ acceso ] class Clase_interna [ extends <Otra_clase_padre> ] [ implements <Otra_interfaz_1> [, <Otra_interfaz_2> [...] ] ]   {
        // Declaración de la clase interna
   }

   // Declaración de métodos
   [ acceso ] [ no acceso ] <tipo> <nombre_metodo> ( <argumentos>) {
      <instrucciones>;
   } // metodo


   // Declaración de métodos abstractos -> clase debe ser abstracta
   abstract [ public | protected ] <tipo> <nombre_metodo> ( <argumentos>);

   ...

} // Clase
```

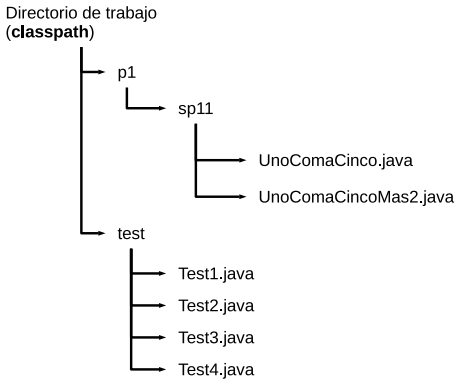
donde

- acceso: `[ ]` | `public` | `protected` | `private`
- no acceso: `[ ]` | `final` | `static` | `abstract`

[ ] (sin modificador o modificador vacío): acceso por paquete

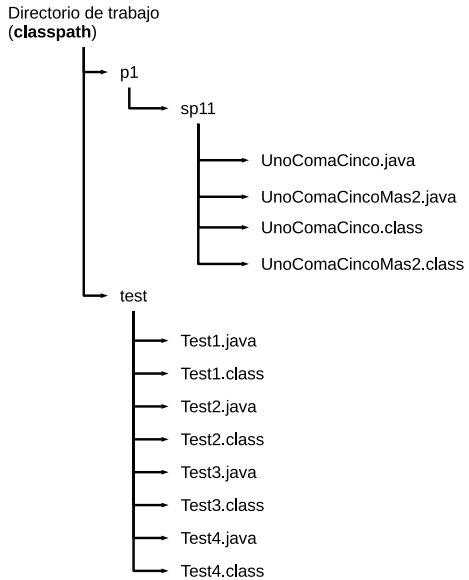
Modificadores: esquema del ejemplo





Desde directorio de trabajo: compilación voraz y selectiva (justificación nombre de fichero = nombre de clase.java)

> javac test\Test[n].java



Ejecución (desde directorio de trabajo):

> java test.Test[n]

Modificadores: ejemplo

PorUnoComaCinco.java

```
package p1.sp11;

class PorUnoComaCinco {

    private float x_por_uno_coma_cinco;
    int x;

    PorUnoComaCinco( int x ) {
        this ( (float) (1.5 * x) );
        this.x = x;
    }

    private PorUnoComaCinco( float y ) {
        x_por_uno_coma_cinco = y;
    }

    float get_x_por_uno_coma_cinco() {
        return x_por_uno_coma_cinco;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + x_por_uno_coma_cinco + ")";
    }

}
```

PorUnoComaCincoMas2.java

```
package p1.sp11;

public class PorUnoComaCincoMas2 {

    public PorUnoComaCinco a; // Se puede acceder a clase PorUnoComaCinco ???
    private int x;
    private float x_por_uno_coma_cinco_mas_2;

    public PorUnoComaCincoMas2( int x ) {
        a = new PorUnoComaCinco(x);
        // Acceso a metodo .get_x_por_uno_coma_cinco() ???
        x_por_uno_coma_cinco_mas_2 = a.get_x_por_uno_coma_cinco() + 2.f;
        this.x = a.x; // Acceso a propiedad x???
        // ¿¿¿ Podría acceder a a.x_por_uno_coma_cinco ???
    }

    public float get_x_por_uno_coma_cinco_mas_2() {
        return x_por_uno_coma_cinco_mas_2;
    }

    public int get_x() {
        return x;
    }

    public String toString() {
        return "[" + a + ", " + x_por_uno_coma_cinco_mas_2 + "]";
    }
}
```

Test1.java

```
package test;

import p1.sp11.PorUnoComaCincoMas2;

class Test1 {

    public static void main (String [] args) {

        PorUnoComaCincoMas2 z = new PorUnoComaCincoMas2(4);

        System.out.println("z: " + z); // z.toString() -public-

        // z.a ??? z.x ??? z.x_por_uno_coma_cinco_mas_2 ???

        Object t = z.a; // ¿¿¿ No puedo PorUnoComaCinco x = z.a ???

        System.out.println("t: " + t); // t.toString() ?????????
        //System.out.println("t.get_x_por_uno_coma_cinco(): "
        //     + t.get_x_por_uno_coma_cinco() ); // ???

    }
}
```

Test2.java: con error

```
package test;

import p1.sp11.PorUnoComaCincoMas2;

class Test2 {

    PorUnoComaCincoMas2 x; // Motivo del problema:

    public static void main (String [] args) {

        /* ERROR en compilación: acceso a miembro no estático desde metodo
           estático: no permitido */
        x = new PorUnoComaCincoMas2(4);

        System.out.println("x: " + x);

    }
}
```

Test3.java: una solución

```
package test;

import p1.sp11.PorUnoComaCincoMas2;

class Test3 {

    static PorUnoComaCincoMas2 x;

    public static void main (String [] args) {

        x = new PorUnoComaCincoMas2(4);

        System.out.println("x: " + x);

    }
}
```

Test4.java: otra solución

```
package test;

import p1.sp11.PorUnoComaCincoMas2;

class Test4 {

    PorUnoComaCincoMas2 x;

    public static void main (String [] args) {

        Test4 objetoTest = new Test4();

        objetoTest.x = new PorUnoComaCincoMas2(4);

        System.out.println("x: " + objetoTest.x);
```

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Herencia de clases

Herencia

```
[modificadores] class <nombre_clase_hija> extends <nombre_clase_padre> {
   ...
}
```

- La clase hija (subclase) hereda atributos y métodos de **una única** clase padre (superclase)
- Un objeto de la clase hija también lo es de la clase padre y de todas las clases antecesoras
- Un constructor puede llamar a constructor de clase padre con `super(·)`
  - Si no lo hace, se llama implícitamente a la sobrecarga sin parámetros del constructor de la clase padre (es primera instrucción —implícita—): `super()`
- La clase hija puede sobreescribir los métodos de la clase padre: anotación `@Override` recomendable

  Los modificadores de acceso **no pueden ser más restrictivos que los originales**
- Acceso a método o atributo de la clase padre (si se sobreescribe en el hijo): `super.<metodo(·)/atributo>`
  - Imposible desde un objeto si hay sobreescritura
  - No se puede acceder a ningún atributo o método de un *abuelo* o ancestro anterior ni desde un objeto ni desde la clase.
- Acceso a método o atributo de la clase actual: `[this.]<metodo(·)/atributo>`
- Toda clase hereda implícitamente la clase `Object`

Ejemplo de herencia

herencia/clases/ClasePadre.java

```
package clases;

class ClasePadre {

    public int x;
    protected int y;
    int z;
    private int w;

    ClasePadre(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
        w = x + y + z;
    }

    ClasePadre() {
        this(1,1,1);
    }

    int get_w() {
        return w;
    }

    void saludo() {
        System.out.println("Hola, soy ClasePadre");
    }

    public String toString() {
        return "x=" + x + ", y=" + y +  ", z=" + z + ", w=" + w;
    }

}
```

herencia/clases/ClaseHija.java

```
package clases;

public class ClaseHija extends ClasePadre {

    public int z;

    public ClaseHija(int x, int y, int z) {
        super( x, y, z );
        this.z = super.z;
    }

    public ClaseHija() {
        this(1000, 1000);
    }

    public ClaseHija( int x ) {
        this.x = x;
    }

    public ClaseHija( int x, int y ) {
        this.x = x;
        this.y = y;
    }
...
```

herencia/clases/ClaseHija.java (cont.)

```
...
    public void saludo() {
        System.out.println("Hola, soy ClaseHija");
    }

    public void saludo_heredado() {
        super.saludo();
    }

    public int get_w() {
        return super.get_w();
    }

    public int get_y() {
        return y;
    }

    public String toString() {
        return "[" + super.toString() + "]";
    }

}
```

herencia/test/TestHerencia.java

```
package test;

import clases.ClaseHija;

class TestHerencia {

    public static void main ( String[] args ) {

        ClaseHija a = new ClaseHija(1,2,3);
        a.saludo();
        a.saludo_heredado(); // ¿no podría a.super.saludo()?

        System.out.println("a: " + a);
        System.out.println("a.x: " + a.x);
        System.out.println("a.get_y(): " + a.get_y()); // ¿No puedo a.y ???
        System.out.println("a.z: " + a.z); // ¿Por qué puedo a.z ???
        System.out.println("a.get_w(): " + a.get_w()); // ¿Por qué no puedo a.w ???

        ClaseHija b = new ClaseHija(10, 20);
        System.out.println("b: " + b);

        ClaseHija c = new ClaseHija();
        System.out.println("c: " + c);

    } // main
} // TestHerencia
```

Posibilidades de compilación

```
Estrategia voraz:
> cd ......... herencia
> javac test\TestHerencia.java
> java test.TestHerencia

Paso a paso:
> cd ......... herencia
> javac clases\ClasePadre.java
> javac clases\ClaseHija.java
> javac test\TestHerencia.java
> java test.TestHerencia
```

Clases abstractas

- Una clase se define abstracta con el modificador `abstract`
- Puede o no tener métodos abstractos
- Un método abstracto es aquél que no está implementado (no tiene código entre llaves, sólo la declaración con los parámetros)
- Si una clase contiene algún método abstracto está obligada a declararse como abstracta
- Una clase abstracta no se puede instanciar pero sí heredar

ClaseAbstracta.java

```
public abstract class ClaseAbstracta {

    abstract void metodo(Tipo parametro); // método abstracto
}
```

Clase.java

```
public class Clase extends ClaseAbstracta {

    void metodo(Tipo parametro) {
        // Implementación (si no, esta clase también debería declararse  abstracta)
    } // método que ya ha dejado de ser abstracto al tener implementación
}
```

Ejemplo I: clases abstractas

abstracta\numeros\NumeroComplejo.java

```
package numeros;

public abstract class NumeroComplejo {


    abstract public double getParteReal();
    abstract public void   setParteReal(double parteReal);
    abstract public double getParteImaginaria();
    abstract public void   setParteImaginaria(double parteImaginaria);
    abstract public double getModulo();
    abstract public void   setModulo(double modulo);
    abstract public double getArgumento();
    abstract public void   setArgumento(double argumento);


    public NumeroComplejo sumar(NumeroComplejo a) {
        setParteReal( getParteReal() + a.getParteReal() );
        setParteImaginaria( getParteImaginaria() + a.getParteImaginaria() );
        return this;
    }

    public NumeroComplejo restar(NumeroComplejo a) {
        setParteReal( getParteReal() - a.getParteReal() );
        setParteImaginaria( getParteImaginaria() - a.getParteImaginaria() );
        return this;
    }

    public NumeroComplejo multiplicar(NumeroComplejo a) {
        //
    }

    public NumeroComplejo dividir(NumeroComplejo a) {
        //
    }
} // numeros
```

abstracta\numeros\NumeroComplejoCartesiano.java

```
package numeros;

public class NumeroComplejoCartesiano extends NumeroComplejo {

    private double parteReal;
    private double parteImaginaria;

    public NumeroComplejoCartesiano(double parteReal, double parteImaginaria) {
        this.parteReal = parteReal;
        this.parteImaginaria = parteImaginaria;
    }

    public NumeroComplejoCartesiano(NumeroComplejoCartesiano c) {
        this(c.parteReal, c.parteImaginaria);
    }

    public NumeroComplejoCartesiano(NumeroComplejoPolar p) {
        this(p.getParteReal(), p.getParteImaginaria());
    }

    public NumeroComplejoCartesiano() {
        parteReal = 0.;
        parteImaginaria = 0.;
    }

    public double getModulo() {
        return Math.sqrt(parteReal*parteReal + parteImaginaria*parteImaginaria);
    }

    public void setModulo(double modulo) {
        double argumento = getArgumento();
        parteReal      = modulo * Math.cos(argumento);
        parteImaginaria  = modulo * Math.sin(argumento);
    }

    public double getArgumento() {
        return Math.atan(parteImaginaria/parteReal);
    }

    public void setArgumento(double argumento) {
        double modulo   = getModulo();
        parteReal       = modulo * Math.cos(argumento);
        parteImaginaria = modulo * Math.cos(argumento);
    }

    public double getParteReal() {
        return parteReal;
    }

    public double getParteImaginaria() {
        return parteImaginaria;
    }

    public void setParteReal(double parteReal) {
        this.parteReal = parteReal;
    }

    public void setParteImaginaria(double parteImaginaria) {
        this.parteImaginaria = parteImaginaria;
    }

    public String toString() {
        return "(" + parteReal + " "+ ((parteImaginaria<0.)?"":"+") +
                parteImaginaria + " i)";
    }

} // Numero complejo cartesiano
```

abstracta\numeros\NumeroComplejoPolar.java

```
package numeros;

public class NumeroComplejoPolar extends NumeroComplejo {

    private double modulo;
    private double argumento;

    public NumeroComplejoPolar(double modulo, double argumento) {
        this.modulo = modulo;
        this.argumento = argumento;
    }

    public NumeroComplejoPolar(NumeroComplejoPolar p) {
        this(p.modulo, p.argumento);
    }

    public NumeroComplejoPolar(NumeroComplejoCartesiano c) {
        this(c.getModulo(), c.getArgumento());
    }

    public NumeroComplejoPolar() {
        modulo = 0.;
        argumento = 0.;
    }

    public double getModulo() {
        return modulo;
    }

    public void setModulo(double modulo) {
        this.modulo = modulo;
    }

    public double getArgumento() {
        return argumento;
    }

    public void setArgumento(double argumento) {
        this.argumento = argumento;
    }

    public double getParteReal() {
        return modulo * Math.cos(argumento);
    }

    public double getParteImaginaria() {
        return modulo * Math.sin(argumento);
    }

    public void setParteReal(double parteReal) {
        double parteImaginaria = getParteImaginaria();
        modulo = Math.sqrt(parteReal*parteReal + parteImaginaria*parteImaginaria);
        argumento = Math.atan(parteImaginaria/parteReal);
    }

    public void setParteImaginaria(double parteImaginaria) {
        double parteReal = getParteReal();
        modulo = Math.sqrt(parteReal*parteReal + parteImaginaria*parteImaginaria);
        argumento = Math.atan(parteImaginaria/parteReal);
    }

    public String toString() {
        return "("+ modulo + "|_" + argumento + ")";
    }

} // NumeroComplejoPolar
```

abstracta\test\TestNumeroComplejo.java

```
package test;

import numeros.*;

class TestNumeroComplejo {

    public static void main (String[] args) {

        NumeroComplejoPolar p   = new NumeroComplejoPolar( 10, -0.5 );
        System.out.println("p:         " + p + " = " + new NumeroComplejoCartesiano(p));

        NumeroComplejoCartesiano c  = new NumeroComplejoCartesiano(3.,4.0);
        System.out.println("c:         " + c + " = " + new NumeroComplejoPolar(c));

        /* p2 <- p */
        NumeroComplejoPolar p2  = new NumeroComplejoPolar(p);
        /* c2 <- c */
        NumeroComplejoCartesiano c2 = new NumeroComplejoCartesiano(c);

        /* Suma de numero complejo cartesiano con uno polar */
        /* p2 <- p2 (p original) +  c */
        System.out.print("p2<-p+c:   " + p2.sumar(c));
        System.out.println(" = " + new NumeroComplejoCartesiano(p2));

        /* Suma de numero complejo polar con uno cartesiano */
        /* c2 <- c2 (c original) + p */
        System.out.print("c2<-p+c:   " + c2.sumar(p));
        System.out.println(" = " + new NumeroComplejoPolar(c2));

    } // main

} // TestNumeroComplejo
```

Ejemplo II: ejercicio incompleto por resolver sobre clases abstractas

Figura.java

```
public abstract class Figura {
    public abstract double getArea(); // Devolver area figura
    public abstract double getPerimetro(); // Devolver perimetro figura
}
```

Punto.java

```
public class Punto {

    // guardar punto como array de dos elementos

    // Constructor
    public Punto (dos coordenadas -array-){
    }

    // Constructor de copia
    public Punto (Punto p) {
    }

    // Devolucion de p
    ...

    //Calcula distancia euclidea desde aqui hasta el punto q
    ... distanciaEuclidea(Punto q) {
    }

    public String toString() {
    }
}
```

Circulo.java

```
public class Circulo extends Figura {

    // Circulo definido por un centro (punto) y un radio

    public Circulo (...) {
    }
    public double getArea() {
    }
    public double getPerimetro() {
    }
    public String toString() {
    }
}
```

Rectangulo.java

```
public class Rectangulo extends Figura {

/* . esi(x0,y0)          . esd(x1,y0)

   . eii(x0,y1)          . eid(x1,y1) */

    private Punto esi; // (x0, y0)
    private Punto eid; // (x1, y1)
    private Punto esd; // (x1, y0)
    private Punto eii; // (x0, y1)
    private double ladoHorizontal; // esi - esd
    private double ladoVertical; // eii - eid

    public Rectangulo(Punto esi, Punto eid) {
    }

    public double getArea() {
        return ladoVertical * ladoHorizontal;
    }

    public double getPerimetro() {
        return 2*ladoVertical + 2*ladoHorizontal;
    }

    public String toString() {
        return "\nRectangulo:\n. "
               + esi + "     . " + esd + "\n. "
               + eii + "     . " + eid;
    }
}
```

Test.java

```
class Test {
    public static void printCaracteristicas(Figura f) {
        System.out.println("\nFigura: " + f);
        System.out.println("Area: " + f.getArea());
        System.out.println("Perimetro: " + f.getPerimetro() );
    }

    public static void main (String [] args) {
        double [] p1 = new double[]{1,2};
        Punto esi = new Punto( p1 );
        Punto eid = new Punto(new double[]{3, 4});
        Rectangulo r = new Rectangulo(esi, eid);
        printCaracteristicas(r);

        Punto centro = new Punto( new double[]{5,5} );
        Circulo c = new Circulo(centro, 4);
        printCaracteristicas(c);
    }
}
```

Interfaces

- Concepto similiar a una clase abstracta, pero con matices
- Descripción de un *contrato* de programación
- Puede contener:
  - Métodos abstractos (implícitamente `public` y `abstract`: no es necesario poner estos modificadores)
  - Métodos `default` (implícitamente `public`)
    - Método alternativo por si no llega a implementarse
  - Métodos estáticos (implícitamente `public`)
    - Métodos ya implementados y utilizables
  - Constantes (implícitamente `public static final`)
- No se pueden instanciar (crear objetos)
- Sí se pueden referenciar (polimorfismo): `Interfaz objeto = new Clase(·);` (si `Clase` implementa `Interfaz`)

Interfaces (cont.)

- La clase que *implemente* la interfaz, debe implementar todos los métodos abstractos de la interfaz (o en su defecto, declararse como abstracta)
- Una interfaz puede *extender* varias interfaces. (Una clase sólo puede extender una clase padre e *implementar* una o varias interfaces.)
- Se puede emplear una constante de una interfaz sin tener que implementarla, en cuyo caso debe referenciarse como `NombreInterfaz.Constante`
- Interfaz genérica:

Ejemplo genérico de interfaz

```
[ public ] interface <nombre_interface> [  extends <otra_interfaz>[, <y_otra_interfaz>, ...] ] {

   <tipo1> <nombre_metodo_1>( <argumentos_1> ); // metodo abstracto a implementar en clase

   default <tipo2> <nombre_metodo_2>( <argumentos_2> ) {  // Implementación del metodo default
      ...
   }

   static  <tipo3> <nombre_metodo_3>( <argumentos_3> ) { // Implementación del metodo estático
      ...
   };

   // Atributos: serán public static y final
   int constante1 = 1;
   ...
}
```

- La firma del contrato:

implements

```
[modificador] Clase [extends SuperClase] [  implements Interfaz1 [, Interfaz2 [, ...]] {
    ...
}
```

Ejemplo de interfaces

reproductor\Player.java

```
// no hay sentencia package: directorio actual

interface Player {
    /* atributos public static final */
    int VOL_MIN = 0;
    int VOL_MAX = 100;
    int VOL_MUTE = VOL_MIN;

    enum ESTADO STOP, PLAYING, PAUSED;

    /* public abstract */
    void play(int n);
    void pause();
    void stop();
    void next();
    void last();
    void volumen(int x);

    /* Ya implementados: default, public */
    default void mute() {
        volumen( VOL_MUTE );
    }

} // interface Player
```

reproductor\MP3.java

```
// no hay sentencia package: directorio actual

class MP3 implements Player {

    public final String sistema = "MP3";
    int n_cancion = 1;
    int vol = VOL_MIN;
    ESTADO estado = ESTADO.STOP;

    public void play(int n) {
        if (n>0) {
            n_cancion = n;
            estado = ESTADO.PLAYING;
        }
        System.out.println("En " + sistema + ". Tocando canción " + n_cancion
                           + ". Estado: " + estado );
    }

    public void stop() {
        estado = ESTADO.STOP;
        System.out.println("En " + sistema + ". Parando canción " + n_cancion
                            +". Estado: " + estado );
    }

    public void next() {
        play(n_cancion+1);
    }

    public void last() {
        play(n_cancion-1);
    }

    public void pause() {
        estado = ESTADO.PAUSED;
        System.out.println("En " + sistema + ". Pausando canción " + n_cancion
                           + ". Estado: " + estado );
    }

    public void volumen(int x) {
        if (x<VOL_MAX) {
            vol = x;
        }
        System.out.println("En " + sistema + ". Ajustando volumen a valor: "+ vol);
    }

} // class MP3
```

reproductor\CD.java

```
// no hay sentencia package: directorio actual

class CD implements Player {

    public final String sistema = "CD";
    int n_cancion = 1;
    int vol = VOL_MIN;
    ESTADO estado = ESTADO.STOP;

    public void play(int n) {
        if (n>0) {
            stop(); // Antes de tocar una cancion se pone en modo STOP
            n_cancion = n;
            estado = ESTADO.PLAYING;
        }
        System.out.println("En " + sistema + ". Tocando canción " + n_cancion
                           + ". Estado: " + estado );
    }

    public void stop() {
        estado = ESTADO.STOP;
        System.out.println("En " + sistema + ". Parando canción " + n_cancion
                            +". Estado: " + estado );
    }

    public void mute() {
        // El volumen decrece linealmente, no bruscamente hasta 0
        for (int v = vol; v >= 0; v--) {
            volumen(v);
        }
    }

    public void next() {
        play(n_cancion+1);
    }

    public void last() {
        play(n_cancion-1);
    }

    public void pause() {
        estado = ESTADO.PAUSED;
        System.out.println("En " + sistema + ". Pausando canción " + n_cancion
                           + ". Estado: " + estado );
    }

    public void volumen(int x) {
        if (x<VOL_MAX) {
            vol = x;
        }
        System.out.println("En " + sistema + ". Ajustando volumen a valor: "+ vol);
    }

} // class CD
```

reproductor\TestPlayers.java

```
// no hay sentencia package: directorio actual

class TestPlayers {

    public static void main ( String[] args ) {

        MP3 mp3 = new MP3();
        mp3.play(1);
        mp3.volumen(3);
        mp3.mute();
        mp3.play(3);
        mp3.next();
        mp3.stop();

        CD cd = new CD();
        cd.play(1);
        cd.volumen(3);
        cd.mute();
        cd.play(3);
        cd.next();
        cd.stop();
```

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Polimorfismo: concepto

- Conversión de clase de hijo a padre (y viceversa dependiendo de los casos)
- Una clase hija disfruta de todos los atributos y métodos (heredables, según modificadores) de la clase padre, por ello:
  - Se puede convertir un objeto de una clase hija a uno de la clase padre (ancestral) correspondiente
  - En el nuevo objeto sólo serán visibles los atributos y métodos de la clase padre (los propios están ocultos y de momento inaccesibles, pero están ahí...)
- Una objeto de la clase padre se puede convertir a un objeto de la clase hija si originalmente el proceso fue inverso.

Polimorfismo: concepto

polimorfismo\general\Polimorfismo

```
// X es superclase de Y y de Z.
// Y y Z son clases "hermanas" (incompatibles)

// Tres clases en un mismo fichero: ok si hay SOLO una clase pública

class X {
}

class Y extends X {
}

class Z extends X {
}

public class Polimorfismo {
    public static void main(String args[ ]) {
        X x = new X();
        Y y = new Y();
        Z z = new Z();
        X xy = new Y(); // Ok (el objeto de clase Y lo es también de la clase X)
        X xz = new Z(); // Ok (el objeto de clase Z lo es también de la clase X)
        Y yz = new Z(); // Error en compilación: incompatible type (siblings)
        Y y1 = new X(); // Error en compilación: X is not a Y
        Z z1 = new X(); // Error en compilación: X is not a Z
        X x1 = y; // Ok (y es objeto de subclase de X)
        X x2 = z; // Ok (z es objeto de subclase de X)
        Y y1 = (Y) x; // Compila OK, pero error en tiempo de ejecución
        Z z1 = (Z) x; // Compila OK, pero error en tiempo de ejecución
        Y y2 = (Y) x1; // ok (x1 es de clase Y)
        Z z2 = (Z) x2; // ok (x2 es de clase Z)
        Y y3 = (Y) z; // Compila OK, pero error en tiempo de ejecución (siblings)
        Z z3 = (Z) y; // Compila OK, pero error en tiempo de ejecución (siblings)
        Object o = z;
        Object o1 = (Y) o; // Compila OK, pero error en tiempo de ejecución
    } // main
} // Polimorfismo
```

Polimorfismo: ejemplo

espacioVectorial\GeometriaEuclidea.java

```
package espacioVectorial;

public interface GeometriaEuclidea {

    double sumaDeCuadrados();

    default double norma() {
        return Math.sqrt( sumaDeCuadrados() );
    }

}
```

espacioVectorial\Coordenada2D.java

```
package espacioVectorial;

public class Coordenada2D implements GeometriaEuclidea {

  private double x;
  private double y;

  public Coordenada2D (double x, double y) {
    this(x); /* equivalente a this.x=x */
    this.y = y;
  }

  private Coordenada2D (double x) {
    this.x = x;
    this.y = 0.;
  }

  public double getX() {
    return x;
  }

  public double getY() {
    return y;
  }

  /* Implementacion de la interfaz GeometriaEuclidea;
     de modificador de acceso por paquete pasa a protected (menos
     restrictivo): OK. Podemos reutilizar
     norma() de la implementación por defecto que hay en la interfaz
     GeometriaEuclidea, sin ninguna modificación adicional  */
  protected double sumaDeCuadrados() {
    return x*x + y*y;
  }

  public String toString() {
    return "(" + x + ", " + y + ")";
  }

}
```

espacioVectorial\Coordenada3D.java

```
package espacioVectorial;

public class Coordenada3D extends Coordenada2D implements GeometriaEuclidea {

/* ¿Es necesario poner implements GeometriaEuclidea si ya lo implementa su padre
y lo hereda ??? */

    private double z;

    public Coordenada3D (double x, double y, double z) {
        super( x, y ); // Si no se pusiera, se llamaria a super()
        this.z = z;
    }

    public double getZ() {
        return z;
    }

    /* Actualiza sumadeCudarados de Coordenada2D.
    */
    protected double sumaDeCuadrados() {
        return super.sumaDeCuadrados() + z*z;
    }

    @Override
    public String toString() {
        return "(" + getX() + ", " + getY() + ", " + z + ")";
    }

} // Coordenada3D
```

Polimorfismo: ejemplo (cont.)

test\Test2D.java

```
package test;

import espacioVectorial.*;

class Test2D {

    public static void main (String[] args) {

        Coordenada2D xy = new Coordenada2D( 3.4, -5.6 );
        System.out.println( "Coordenada 2D: " + xy );
        System.out.println( "Coordenada x: "  + xy.getX() );
        System.out.println( "Coordenada y: "  + xy.getY() );
        System.out.println( "Norma: " + xy.norma() );

    } // main

} // Test2D
```

test\Test3D.java

```
package test;

import espacioVectorial.*;

class Test3D {

    // Polimorfismo en la llamada
    static void printNorma(GeometriaEuclidea C) {
        System.out.println("Norma: " + C.norma() );
    }

    public static void main (String[] args) {

        Coordenada2D xy = new Coordenada2D( 3.4, -5.6 );
        System.out.println( "Coordenadas xy (2D): " + xy );
        System.out.println( "Coordenada x: "  + xy.getX() );
        System.out.println( "Coordenada y: "  + xy.getY() );
        System.out.println( "Norma: " + xy.norma() );
        printNorma(xy);

        Coordenada3D xyz = new Coordenada3D( 3.4, -5.6, 3.3 );
        System.out.println( "Coordenada xyz (3D): " + xyz );
        System.out.println( "Coordenada x: "  + xyz.getX() );
        System.out.println( "Coordenada y: "  + xyz.getY() );
        System.out.println( "Coordenada z: "  + xyz.getZ() );
        System.out.println( "Norma: " + xyz.norma() );
        printNorma(xyz);

        /* Acceso a un método del objeto xyz */
        String xyzStr =  xyz.toString();
        /* No se puede acceder explicitamente a un método de un padre desde
           un objeto hijo (sí desde la clase -vease sumaDeCuadrados()-):
        String xyzStr2 =  xyz.super.toString(); // Error!!! */


        // Polimorfismo
        Coordenada2D pqr = (Coordenada2D) xyz;
        System.out.println( "Coordenada pqr (2D???) : " + pqr );
        System.out.println( "Coordenada x: "  + pqr.getX() );
        System.out.println( "Coordenada y: "  + pqr.getY() );
        //Error: System.out.println( "Coordenada z: "  + pqr.getZ() );
        // ¿Qué norma? ¿La de 2D o la de 3D????
        System.out.println( "Norma?????: " + pqr.norma() );
        printNorma(pqr);

        Coordenada3D ijk = (Coordenada3D) pqr;
        System.out.println( "Coordenada ijk (3D) : " + ijk );
        System.out.println( "Coordenada x: "  + ijk.getX() );
        System.out.println( "Coordenada y: "  + ijk.getY() );
        System.out.println( "Coordenada z: "  + ijk.getZ() );
        System.out.println( "Norma: " + ijk.norma() );
        printNorma(ijk);

    } // main

} // Test3D
```

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Métodos y clases genéricos

- Planteamiento inicial: métodos genéricos
- Los parámetros **deben** ser objetos o envoltorios, y **no** tipos primitivos

genericidad\Intro\ReplicacionFunciones.java

```
class ReplicacionFunciones {

  public static void printArray_int( int [ ] vector_int ) {
    for ( int i = 0; i < vector_int.length; i++ ) {
      System.out.println( "vector_int[" + i + "]=" + vector_int[i] );
    }
  }

  public static void printArray_double( double [ ] vector_double ) {
    for ( int i = 0; i < vector_double.length; i++ ) {
      System.out.println( "vector_double[" + i + "]=" + vector_double[i] );
    }
  }

  public static void main( String [ ] args ) {
    int [ ] intArray = { 1, 2, 3 };
    double[ ] doubleArray = { 1.1, 2.2, 3.3, 4.4 } ;

    printArray_int( intArray );
    printArray_double( doubleArray );

  }

}
```

genericidad\Intro\Sobrecarga.java

```
class Sobrecarga {

  // Método printArray sobrecargado: esto NO es genericidad

  public static void printArray( int [ ] vector_int ) {
    for ( int i = 0; i < vector_int.length; i++ ) {
      System.out.println( "vector_int[" + i + "]=" + vector_int[i] );
    }
  }

  public static void printArray( double [ ] vector_double ) {
    for ( int i = 0; i < vector_double.length; i++ ) {
      System.out.println( "vector_double[" + i + "]=" + vector_double[i] );
    }
  }

  public static void main( String [ ] args ) {
    int [ ] intArray = { 1, 2, 3 };
    double[ ] doubleArray = { 1.1, 2.2, 3.3, 4.4 } ;

    printArray( intArray );
    printArray( doubleArray );

  }

}
```

genericidad\Intro\Genericidad.java

```
class Genericidad {

  public static <T> void printArray( T [ ] vector ) {
    for ( int i = 0; i < vector.length; i++ ) {
      System.out.println( "vector[" + i + "]=" + vector[i] );
    }
  }

  public static void main( String [ ] args ) {

    Integer [ ] intArray = { 1, 2, 3 };
    Double[ ] doubleArray = { 1.1, 2.2, 3.3, 4.4 };

    printArray( intArray );
    printArray( doubleArray );

  }

}
```

Restricciones

- Restricción de clases:
  - Sin acotación: `<T>`
  - Con acotación superior, ej.: `<T extends Clase>`
  - Con acotación inferior, ej.: `<T super Clase>`
  - Clase no referenciada `<? extends Clase>`
  - Dependiendo del contexto, el compilador puede *inferir* la clase o tipo paramétrico en la definición:

```
Clase<Tipo> objeto = new Clase<>(...);
```

  - Uso notacional:
    - `E`: elemento en el contexto de *colecciones*
    - `K`, `V`: pares clave-valor
    - `N`: clase Number
    - `T`: tipo (clase) genérico
    - `S`,`U`,`V`, etc.: siguientes tipos (clases) genéricas
  - ...
  - `https://docs.oracle.com/javase/tutorial/java/generics/index.html`

En `genericidad\EspaciosVectoriales[n]` hay ejemplos con mayor complejidad sobre genericidad.

Ejemplo con restricciones

Otro ejemplo de métodos genéricos: condiciones sobre las clases genéricas

genericidad\GenericidadMetodoClase\GenericidadClase.java

```
public class GenericidadClase {

   // Maximo de tres objetos "comparables"

   public static <T extends Comparable<T>> T maximo(T x, T y, T z) {

      T max = x;   // máximo provisional

      if ( y.compareTo(max) > 0 ) {
         max = y;   // y seria ahora el máximo provional
      }

      if ( z.compareTo(max) > 0) {
         max = z;   // z sería el máximno
      }

      return max;   // returna el máximo definitivamente
   }

   public static void main(String args[]) {

      System.out.printf("Máximo de %d, %d y %d es: %d\n\n",
         3, 4, 5, maximo( 3, 4, 5 ));

      System.out.printf("Máximo de %.1f, %.1f y %.1f es: %.1f\n\n",
         6.6, 8.8, 7.7, maximo( 6.6, 8.8, 7.7 ));

      System.out.printf("Máximo de %s, %s y %s es %s\n","hola",
         "adios", "hasta luego", maximo("hola", "adios", "hasta luego"));
   }

}
```

- Búsquese documentación de la interfaz `Comparable<T>`: implementación de `int compareTo(T objeto);`
- Verifíquese que `Integer`, `Double` y `String` implementan la interfaz `Comparable<T>` (`Comparable<Integer>`, `Comparable<Double>` y `Comparable<String>` respectivamente)

Métodos y clases genéricos: otro ejemplo

Ejemplo adicional: coordenadas 2D y 3D

genericidad\espaciosVectoriales\espacioVectorial\GeometriaEuclidea.java

```
package espacioVectorial;

public interface GeometriaEuclidea extends Comparable<GeometriaEuclidea> {

    double sumaDeCuadrados();

    default double norma() {
        return Math.sqrt( sumaDeCuadrados() );
    }

    default int compareTo( GeometriaEuclidea c ) {
        if ( norma()>c.norma() ) return 1;
        else if ( norma()<c.norma() ) return -1;
        else return 0;
    }
}
```

genericidad\espaciosVectoriales\espacioVectorial\test\Test3D.java

```
...

      System.out.println("Maximo de " + xy + ", " + xyz + " y "
                         + pqr + " es: " + maximo(xy, xyz, pqr ));

...
```

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

Clases anidadas

Clases anidadas: clases dentro de clases

```
class Clase {
    ...
    static class ClaseAnidadaEstatica {
        ...
    }
    class ClaseInterna { // clase anidada no estática
        ...
    }
}
```

- Objetos de clases anidadas:
  - `ClaseAnidadaEstatica`: clase anidada estática

    Ejemplo clase anidada estática

    ```
    ClaseExterna.ClaseAnidadaEstatica objeto = new ClaseExterna.ClaseAnidadaEstatica();
    ```

  - `ClaseInterna`: clase interna

    Ejemplo clase interna

    ```
    Clase objeto = new Clase(...);
    Clase.ClaseInterna objeto2 = objeto.new ClaseInterna();
    ```

- Adicionalmente:
  - Clases locales
  - Clases anónimas

Clases locales

- Definidas dentro de un *bloque* de instrucciones de un método

Clase local

```
class Clase {

   ...

   public void metodo( ) {

     int i;

      class ClaseLocal {
         ...
      } // ClaseLocal

     ClaseLocal sc = new ClaseLocal( );

     ...

   } // metodo()

} // Clase
```

Clases anónimas

- Clase anónima: la clase se define en una expresión como extensión de una clase padre y carece de nombre.
  - No se plantean nuevos miembros, si no, sobreescritura de los ya existentes en la clase padre
- El objeto se instancia al mismo tiempo que se define su clase con un claro polimorfismo
- Son clases locales, pero sin nombre
- Utilidad cuando se instancia una clase local sólo una vez
- Idea extensible a interfaces: se instancia una interfaz *implementada*
- La clase extendida (interfaz implementada) no tiene nombre explícito: anónimas

Clases anónimas: sintaxis y equivalencia

- Sintaxis:

Clase anónima

```
// Por extensión inmediata de una clase: polimorfismo implícito
Clase objeto = new Clase( param ) {
   ... extensión de la clase Clase
};

// Por implementación inmediata de una interfaz
Interfaz objeto = new Interfaz( ) {
   ... Implementación de la interfaz Interfaz
};
```

- Formulación equivalente

```
//////// Situación convencional:

class Clase_B extends Clase_A {
    Clase_B(param) {
        super(param);
    }
    @Override
    public metodo() {
        ...
    }
    int k = 27;
}

// Polimorfismo
Clase_A objeto = new Clase_B(param);


// Utilización como argumento
cierto_metodo( objeto );
// Alternativa: el objeto del argumento es anónimo
cierto_metodo( new Clase_B(param) );

////////// Alternativa con clases anónimas

// En este nuevo contexto,
// Clase_B sería la extensión, pero
// no aparece con dicho nombre:

Clase_A objeto = new Clase_A (param) {
    metodo() {
        ...
    }
    int k = 27;
}
// La clase anónima es la ampliación de la Clase_A


// Utilización como argumento
cierto_metodo( objeto );
// Alternativa: objeto anónimo de clase anónima
cierto_metodo( new Clase_A (param) { metodo(){... } int k = 27;}  );
```

Ejemplo de clases anónimas

genericidad\EspaciosVectoriales\testClasesAnonimas\TestAnonimato.java

```
package testClasesAnonimas;

import espacioVectorial.*;

class TestAnonimato {

    public static void main (String[] args) {

        // Referencia: extensión convencional de una clase
        // NuevaCoordenada3D es una clase local
        class NuevaCoordenada3D extends Coordenada3D {
            NuevaCoordenada3D(double x, double y, double z) {
                super(x, y, z);
            }

            @Override public String toString() {
                return "[" + super.toString() + "]";
            }
        };

        NuevaCoordenada3D n = new NuevaCoordenada3D(1., 2., 3.);
        System.out.println("n: " + n);


        // Clase anónima: no reutilizable posteriormente
        Coordenada3D m = new Coordenada3D(10., 20., 30.) {
            // Si no hay sobreescritura se mantienen constructores
            // de superclase
            @Override public String toString() {
                return "" + super.toString() + "";
            }
        };

        System.out.println("m:" + m);

        // Sin generar objeto intermedio:
        System.out.println("Coordenada: " +
            new Coordenada3D(100, 200, 300) {
                @Override public String toString() {
                    return "¨" + super.toString() + "¨";
                } // toString
            } // new Coordenada3D
        );  // println
...
```

...\TestAnonimato.java (cont.)

```
...
        // Con Polimorfismo
        GeometriaEuclidea g = new Coordenada3D(3.4, 5.6, 3.3) {
            @Override public String toString() {
                return "#" + super.toString() + "#";
            }
        };

        System.out.println("norma de g: " + g.norma() + " .toString: "+ g);

        // Clase anónima implementando un interfaz
        GeometriaEuclidea g2 = new GeometriaEuclidea() {
            // public ya que los métodos abstractos del interfaz son
            // siempre publicos y en la implementación no se puede
            // aminorar el acceso
            public double sumaDeCuadrados() { // ...pequeña licencia...
                return 81.;
            }

            @Override public String toString() {
                return "*" + super.toString() + "*";
            }
        };

        // Explicación
        System.out.println("norma de g2: " + g2.norma() + " .toString: "+ g2);

    } // main
} // TestAnonimato
```

Índice

5 Clases y objetos
- Conceptos básicos
- Métodos, sobrecargas y constructores
- Comparación y asignación de objetos
- Clases enumeradas
- Paquetes
- Modificadores
- Extensión de clases (herencia) e implementación de interfaces
- Polimorfismo
- Genericidad en Java
- Clases anidadas
- Expresiones Lambda

- Permiten simplificar algunos bloques de código que hacía uso de clases anónimas
- Una expresión lambda consta de:
  - Conjunto de parámetros
  - Expresión que opera con los parámetros
- Bajo ciertas condiciones, puede sustituirse el uso de una clase anónima por una expresión lambda

Test.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;


class Persona {

    private String nombre;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public Persona(String nombre) {
        super();
        this.nombre = nombre;
    }

} // Persona



public class Test {

    public static void main(String[] args) {

        ArrayList<Persona> milista= new ArrayList<Persona>();

        milista.add( new Persona("Miguel") );
        milista.add( new Persona("Alicia") );

        Collections.sort( milista,
            new Comparator<Persona>() {
                public int compare( Persona p1, Persona p2 ) {
                    return p1.getNombre().compareTo(p2.getNombre());
                }
            }
        ); // Collections.sort

        for (Persona p: milista) {
            System.out.println(p.getNombre());
        } // for

    } // main

} // class Test
```

TestLambda.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;


class Persona {

    private String nombre;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public Persona(String nombre) {
        super();
        this.nombre = nombre;
    }

} // Persona



public class TestLambda {

    public static void main(String[] args) {

        ArrayList<Persona> milista= new ArrayList<Persona>();

        milista.add( new Persona("Miguel") );
        milista.add( new Persona("Alicia") );

        Collections.sort( milista,
            ( Persona p1,Persona p2 ) -> p1.getNombre().compareTo( p2.getNombre() )
        ); // Collections.sort

        for (Persona p: milista) {
            System.out.println(p.getNombre());
        } // for

    } //main

} // class TestLambda
```

Índice

6 Más características del lenguaje Java
- Excepciones
- Programación multihilo
- Compilación y ejecución
- Depuración en Java

Índice

6 Más características del lenguaje Java
- Excepciones
- Programación multihilo
- Compilación y ejecución
- Depuración en Java

Bloques try, catch y finally

Excepciones

```
try {

      <instrucciones donde se pueden producir excepciones>

} catch ( TipoExcepcion1 excepcion1 ) {

      <instrucciones donde se trata la excepción de clase TipoExcepcion1>;

} catch ( TipoExcepcion2 excepcion2 ) {

      <instrucciones donde se trata la excepción de clase TipoExcepcion2>;

} finally {

      <instrucciones a ejecutar tanto si ocurren excepciones en el bloque try o si
      ocurre alguna nueva dentro de cualquier bloque catch,  como si no>;

      <si ocurre alguna excepción nueva dentro de algún bloque catch, entonces adicionalmente
       tendrá que ser  "tratada" en un bloque catch de nivel "superior", no ejecutándose
       "...siguientes instrucciones">

      <si no ocurre ninguna excepción nueva dentro de ningún bloque catch, se continúa
      en "...siguientes instrucciones">

}

... siguientes instrucciones java
```

Sin tratamiento interno de excepciones: `throws`

Si el método no trata las excepciones, debe *lanzarlas* para que las trate el llamante:

Ejemplo excepciones: throws

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
   ...
}
```

Ejemplo de excepciones: con un string fijo

excepciones\Excepciones_0.java

```
class Excepciones_0 {

    public static void main( String [] args ) {

        String numeroStr = "3.2454aadf56";

        double x=0;

        boolean valido = true;

        try {

            x = Double.parseDouble( numeroStr );
            System.out.println("Numero: " + x );

        } catch ( NumberFormatException e ) {

            System.out.println( "Formato de número erroneo. Excepción: " + e);
            valido = false;

            // int j = 3/0; // Pruébese...

        } finally {

            System.out.println("El string \"" + numeroStr + "\" tenia un formato " + ((valido)? "" : "in") + "correcto");

        }

        System.out.println("En cualquier caso, ... si no hay excepciones en catch");

    }

}
```

Ejemplo de excepciones: lectura de una línea por teclado

excepciones\Lectura.java

```
/*
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
*/

class Lectura {

    public static void main( String [] args ) {

        BufferedReader br = new BufferedReader( new InputStreamReader( System.in ) );

        String linea;

        try {

            System.out.print("Escribe una línea (fin con <CRLF/LF>): ");
            linea = br.readLine();
            System.out.println("Línea leída: " + linea );

        } catch ( IOException e ) {

            System.out.println( "Error en lectura. Excepción: " +  e);

        }

    } //main

} // Excepciones
```

Ejemplo de excepciones: lectura de un double por teclado

excepciones\Excepciones_1.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
// import ????.NumberformatException ????

class Excepciones_1 {

    public static void main( String [] args ) {

        String numeroStr;

        boolean valido;

        double x;

        BufferedReader br = new BufferedReader( new InputStreamReader( System.in ) );

        do {

            valido = true;
            System.out.print("Introduce un número: ");

            try {

                numeroStr = br.readLine();

                try {

                    x = Double.parseDouble( numeroStr );
                    System.out.println(" x = " + x );

                } catch ( NumberFormatException e ) {

                    System.out.println( "Formato de número erroneo. Excepción: " + e);
                    valido = false;

                } finally {

                    System.out.println("El string \"" + numeroStr + "\" tenia un formato " + ((valido)? "" : "in") + "correcto");

                }

            } catch ( IOException e ) {

                System.out.println( "Error en Entrada/Salida. Excepción: " + e );
                valido = false;

            }

        } while (!valido);

    }

}
```

Ejemplo de excepciones: empleo de `throws` (excepción desatendida)

excepciones\Excepciones_2.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

class Excepciones_2 {


    static double leerNumero() throws NumberFormatException {

        BufferedReader br = new BufferedReader( new InputStreamReader( System.in ) );

        System.out.print("Introduce un número: ");

        try {

            String numeroStr = br.readLine();
            double x = Double.parseDouble( numeroStr );
            return x;

        } catch ( IOException e ) {

            System.out.println( "Error en Entrada/Salida. Excepción: " + e );

        } finally {
            System.out.println("Todo OK, o algún tipo de Excepción (E/S o formato de doubles)...");
        }

        System.out.println("No deberia estar aquí");
        // nunca debe ocurrir esto
        return 0;

    }


    public static void main( String [] args ) {

        double z = 0;
        boolean valido;
        do {

            try {

                z = leerNumero();
                valido = true;

            } catch ( NumberFormatException e ) {

                System.out.println( "Formato de número erroneo. Excepción: " + e);
                valido = false;

            }
        } while (!valido);

        System.out.println(" z = " + z );

    }


}
```

Ejemplo de excepciones: múltiples excepciones

excepciones\Excepciones_3.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

class Excepciones_3 {

    public static void main( String [] args ) {

        String numeroStr = null;

        boolean valido;

        double x = 0;

        BufferedReader br = new BufferedReader( new InputStreamReader( System.in ) );

        do {

            valido = true;
            System.out.print("Introduce un número: ");

            try {

                numeroStr = br.readLine();
                x = Double.parseDouble( numeroStr );
                System.out.println(" x = " + x );

            } catch ( NumberFormatException | IOException e ) {

                System.out.println( "Formato de número erroneo o Error de E/S. Excepción: " + e);
                valido = false;

            } finally {

                System.out.println("El string ¨ + numeroStr + "¨ tenia un formato " + ((valido)? "" : "in") + "correcto");

            }

        } while (!valido);

    }

}
```

Ejemplo de excepciones: desatención total de excepciones

excepciones\Excepciones_4.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

class Excepciones_4 {

    public static void main( String [] args ) throws Exception {

        BufferedReader br = new BufferedReader( new InputStreamReader( System.in ) );

        System.out.print("Introduce un número: ");

        String numeroStr = br.readLine();
        double x = Double.parseDouble( numeroStr );
        System.out.println(" x = " + x );

    }

}
```

Ejemplo de excepciones: creacion de excepciones propias

excepciones\Excepciones_5.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

class Excepciones_5 {


    static class ExcepcionPorComa extends Exception {
        String mensaje;
        ExcepcionPorComa(String s) {
            super(s);
            mensaje = s;
        }
        ExcepcionPorComa(){
            super();
            mensaje="Motivo desconocido";
        }
        public String toString() {
            return mensaje;
        }


    }


    public static void main( String [] args ) throws Exception{

        BufferedReader br = new BufferedReader( new InputStreamReader( System.in ) );

        System.out.print("Introduce un número: ");

        try {
            String numeroStr = br.readLine();
            if ( numeroStr.indexOf(",")>=0 ) throw new ExcepcionPorComa(numeroStr+" lleva coma!!!");
            double x = Double.parseDouble( numeroStr );
            System.out.println(" x = " + x );
        } catch (ExcepcionPorComa e ) {
            System.out.println("Excepcion por coma. Excepcion: " + e );
        }

    }

}
```

Índice

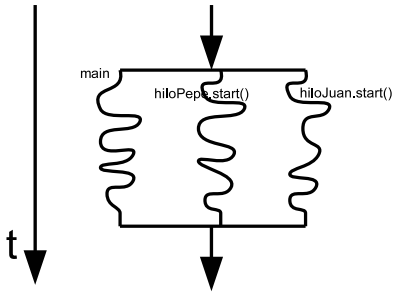
6 Más características del lenguaje Java
- Excepciones
- Programación multihilo
- Compilación y ejecución
- Depuración en Java

Extensión de la clase Thread

ThreadEjemplo.java

```
public class ThreadEjemplo {

    static class Hilo extends Thread {

        // Constructor
        public Hilo(String str) {
            super(str);
        }

        // Accion a realizar por el hilo al ejecutar .start()
        public void run() {
            for (int i = 0; i < 10 ; i++)
                System.out.println(i + " " + getName());
            System.out.println("Termina objeto Hilo " + getName());
        }

    } // Hilo


    public static void main (String [] args) {

        System.out.println("Arranca (thread) main");


        Hilo hiloPepe = new Hilo("Pepe");
        Hilo hiloJuan = new Hilo("Juan");

        hiloPepe.start();
        hiloJuan.start();

        /* Alternativamente de forma anónima:
        new Hilo("Pepe").start();
        new Hilo("Juan").start();
        */

        System.out.println("Termina (thread) main");

    } // main

} // class
```

```
Arranca (thread) main
Termina (thread) main
0 Pepe
1 Pepe
2 Pepe
3 Pepe
0 Juan
4 Pepe
1 Juan
5 Pepe
2 Juan
6 Pepe
3 Juan
7 Pepe
4 Juan
8 Pepe
5 Juan
9 Pepe
6 Juan
Termina objeto Hilo Pepe
7 Juan
8 Juan
9 Juan
Termina objeto Hilo Juan
```



Implementación de la interfaz Runnable

ThreadEjemplo2.java

```
public class ThreadEjemplo2  {


    static class OtroHilo implements Runnable {

        // Único método a implementar de la interfaz Runnable
        public void run() {

            for (int i = 0; i < 5 ; i++) {
                System.out.println( i + " " + Thread.currentThread().getName() );
            }

            System.out.println( "Termina thread " + Thread.currentThread().getName() );

        } // run

    } // OtroHilo


    public static void main (String [] args) {

        Thread hiloPepe = new Thread ( new OtroHilo() , "Pepe");
        Thread hiloJuan = new Thread ( new OtroHilo() , "Juan");

        hiloPepe.start();
        hiloJuan.start();

        /* Alternativa anónima
            new Thread ( new OtroHilo() , "Pepe").start();
            new Thread ( new OtroHilo() , "Juan").start();
        */


    } // main

} // class
```

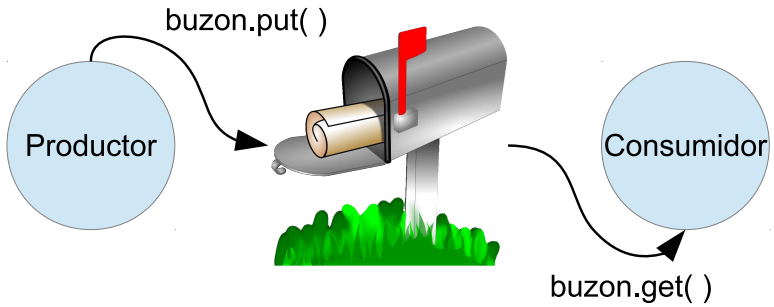
Sincronización de hilos: ejemplo

hilos\ProductorConsumidor\ProductorConsumidorTest.java

```
public class ProductorConsumidorTest {

    public static void main(String[] args) {

        Buzon c = new Buzon ();
        Productor produce = new Productor (c);
        Consumidor consume = new Consumidor (c);
        consume.start();
        produce.start();

    } // main

} // class
```

hilos\ProductorConsumidor\Productor.java

```
public class Productor extends Thread {

    private Buzon buzon;

    public Productor (Buzon c) {
        buzon = c;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {

            // produce i
            buzon.put(i);

            try {

                // sleep cada x segundos; x uniformemente
                //distribuido entre 0 y 5 s
                sleep((int)(Math.random() * 5000));

            } catch (InterruptedException e) {
                // Es obligatorio hacer un catch de sleep
            } // catch

        } // for
    } // run

} // Productor
```



hilos\ProductorConsumidor\Consumidor.java

```
public class Consumidor extends Thread {

    private Buzon buzon;

    public Consumidor (Buzon c) {
        buzon= c;
    }

    public void run() {

        int value;
        for (int i = 0; i < 10; i++) {
            value = buzon.get();
            // Consumo de value
        }

    } // run

} // Consumidor
```

Sincronización de hilos: solución (synchronized)

- Sincronización de los métodos de acceso al objeto: modificador `synchronized`, implica bloqueo del acceso a dicho objeto hasta que finalice la ejecución del métodos `synchronized` en cuestión

hilos\ProductorConsumidor\Buzon.java

```
public class Buzon {

    private int dato;
    private boolean hayDato = false;

    public synchronized void put(int valor) {

        if (hayDato) {

            try {
                // espera a que se consuma el dato
                wait();
            } catch (InterruptedException e) {
                // Obligatorio catch para wait
            }
        }

        dato = valor;
        hayDato = true;

        System.out.println("Productor. put: " + valor );
        // notificar que ya hay dato.
        notify();

    } // put

...
```

hilos\ProductorConsumidor\Buzon.java (cont.)

```
...

  public synchronized int get() {

        if (!hayDato) {
            try {

                // espera a que el productor coloque un valor
                wait();

            } catch (InterruptedException e) {
                // Obligatorio catch para wait
            }

        } // if

        hayDato = false;

        System.out.println("Consumidor. get: " + dato);
        // notificar que el valor ha sido consumido
        notify();

        return dato;

    } // get

} // Class Buzon
```

En `hilos\ProductorConsumidorIncorrecto` aparece un ejemplo "incorrecto".

Otro ejemplo: condición de carrera *(race condition)*

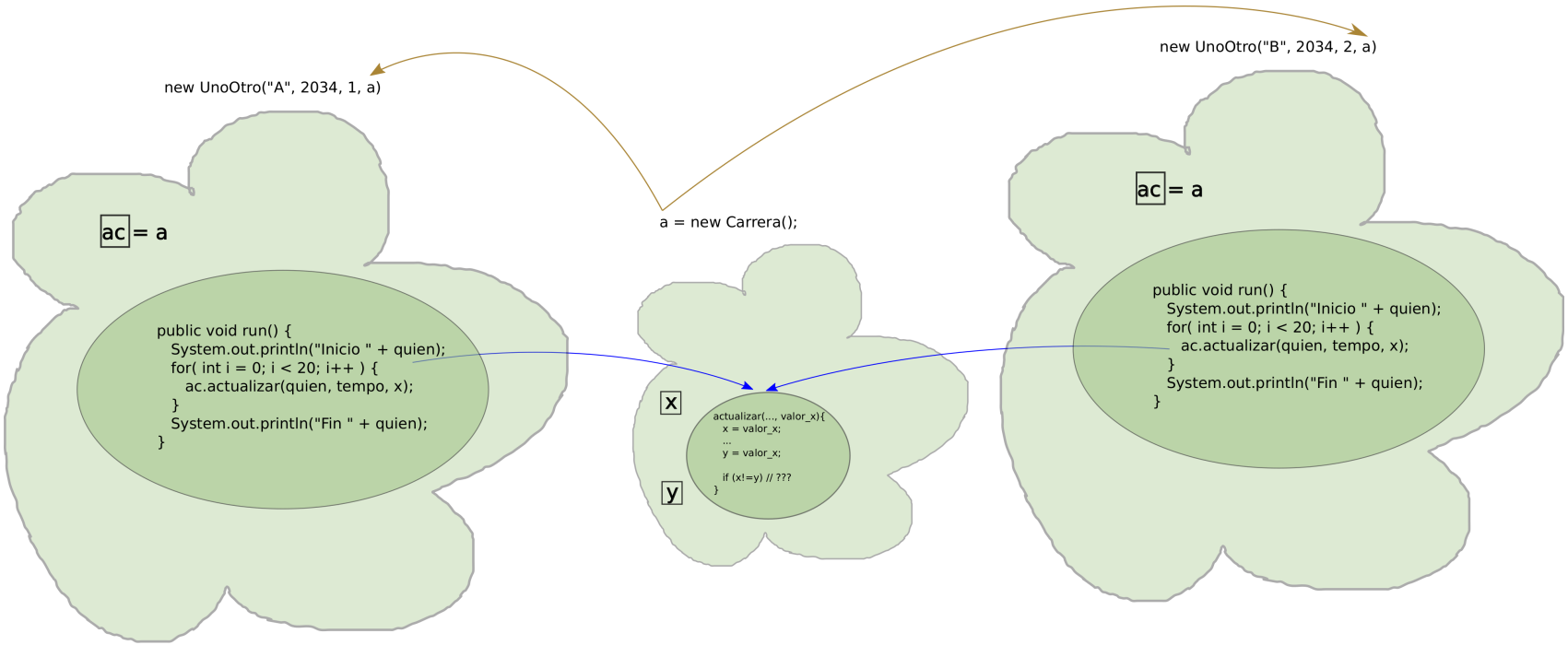
hilos\carrera\UnoOtro.java

```
class Carrera extends {

    int x;
    int y;

    //synchronized  // obsérvese diferencia
    void actualizar(String quien, int tempo, int valor_x) {
        x = valor_x;

        try {
            // Espera aleatoria dentro del intervalo [0..tempo]
            Thread.sleep( (long)Math.ceil(Math.random()*tempo) );
        } catch (InterruptedException e ) {
        }

        y = valor_x;

        if ( x != y )
            System.out.println( quien + " ??????" );
        else
            System.out.println( quien + " OK" );

    } // actualizar

}  // Carrera
...
```

hilos\carrera\UnoOtro.java (cont.)

```
...
class UnoOtro extends Thread {

    String quien;
    int tempo;
    int x;
    Carrera ac;

    UnoOtro(String quien, int tempo, int x, Carrera ac) {
        this.quien = quien;
        this.tempo = tempo;
        this.x     = x;
        this.ac    = ac;
    }

    public void run() {
        System.out.println("Inicio " + quien);
        for( int i = 0; i < 20; i++ ) {
            ac.actualizar(quien, tempo, x);
        }
        System.out.println("Fin " + quien);
    }

    public static void main(String[] args) {
        Carrera a = new Carrera();
        new UnoOtro("A", 2034, 1, a).start();
        new UnoOtro("B", 1178, 2, a).start();
    }
} // UnoOtro
```

Otro ejemplo: condición de carrera *(race condition)* (cont.)



Índice

6 Más características del lenguaje Java
- Excepciones
- Programación multihilo
- Compilación y ejecución
- Depuración en Java

Clases y ficheros

- En términos generales: una clase por fichero y nombre del fichero: `nombre_de_la_clase.java`
- Un fichero puede contener más de una clase si
  - Hay a lo sumo una única clase pública, en cuyo caso
    - Nombre del fichero: `nombre_de_la_clase_publica.java`

Punto de entrada: el método `main`

- A la JVM se le debe proporcionar una clase que contenga un método con el nombre `main` de la siguiente manera (firma del método —nombre y parámetros— modificadores y tipo de vuelta):

Fichero Principal.java: clase con metodo main

```
class Principal {
   ...
   public static void main ( String[ ] args ) {
      ...
   }
}
```

- Por tanto, una vez compilado,

Compilación desde línea de comandos

```
> javac Principal.java
```

la llamada será:

Ejecución desde línea de comandos

```
> java Principal
```

- Si se indica la pertencia a cierto paquete

main

```
package paquete.subpaquete.prin;

class Principal {
   ...
   public static void main ( String[ ] args ) {
      ...
   }
}
```

la clase resultante de la compilación **debería** ubicarse en un directorio resultante de una secuencia de directorios coincidente con las secuencia de paquetes. La llamada deberá ser:

Llamada desde línea de comandos

```
> java paquete.subpaquete.prin.Principal
```

Compilación y destino de clases

- Se puede realizar simultáneamente la compilación y la reubicación de clases en sus directorios oportunos a partir de una referencia (opción `-d`):

Compilación desde línea de comandos y reubicación de clases desde este directorio (.)

```
> javac -d . Principal.java
```

`Principal.class` se encontrará en `paquete\subpaquete\prin`. La secuencia de subdirectorios la crea el propio compilador. Su invocación será:

Invocación a clase con main

```
> java paquete.subpaquete.prin.Principal
```

- Si se ubica en otro sitio disinto, por ejemplo:

Compilación desde línea de comandos y reubicación de clases en directorio\clases

```
> javac -d directorio\clases Principal.java
```

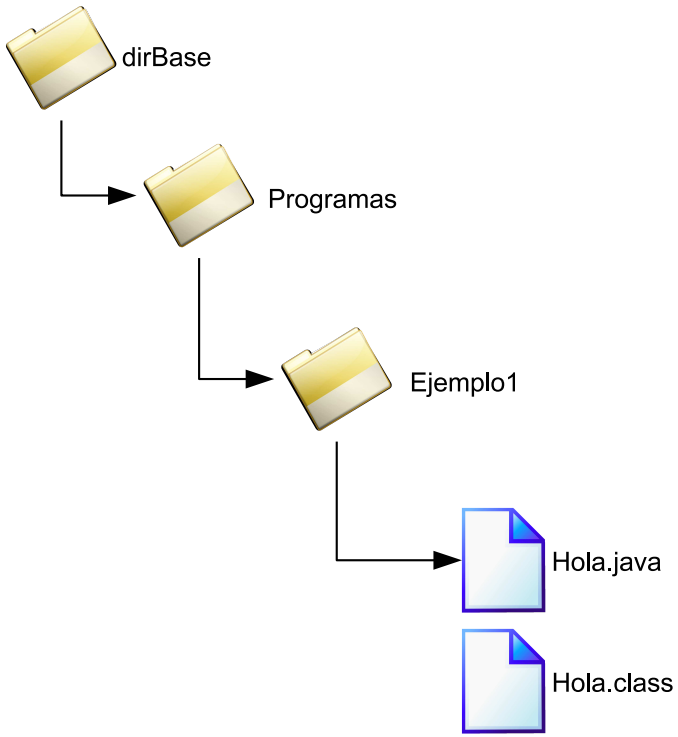
entonces se requiere opción `-cp` o `-classpath`:

Compilación desde línea de comandos y ubicación de clases desde este directorio

```
> java -cp directorio\clases paquete.subpaquete.prin.Principal
```

- Si los fuentes se ubican respetando secuencia de subdirectorios - secuencia de paquetes, se puede realizar compilación *voraz*.

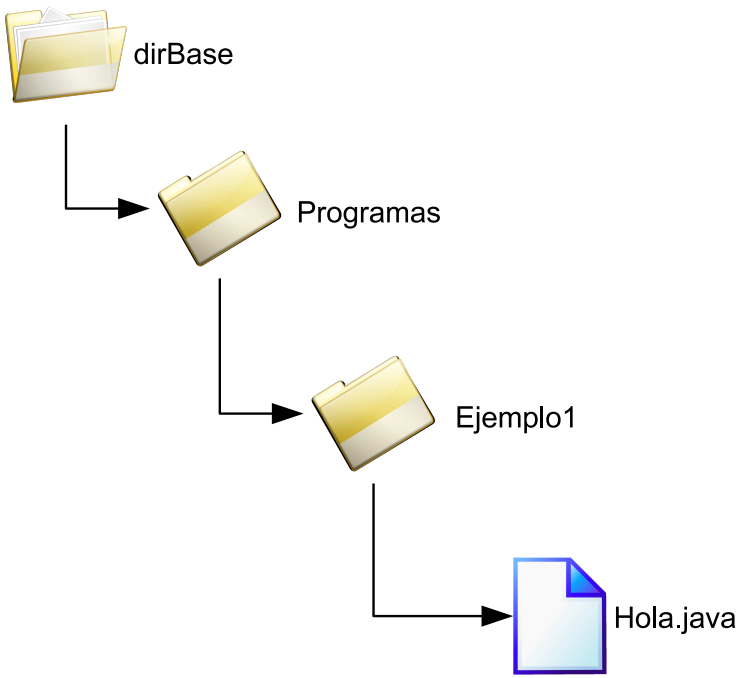
Ejemplo gráfico



Hola.java

```
package Programas.Ejemplo1;

class Hola {
   public static void main( String[ ] args ) {
      System.out.println( "Hola, ¿qué tal?" );
   }
}
```

Ejemplo gráfico



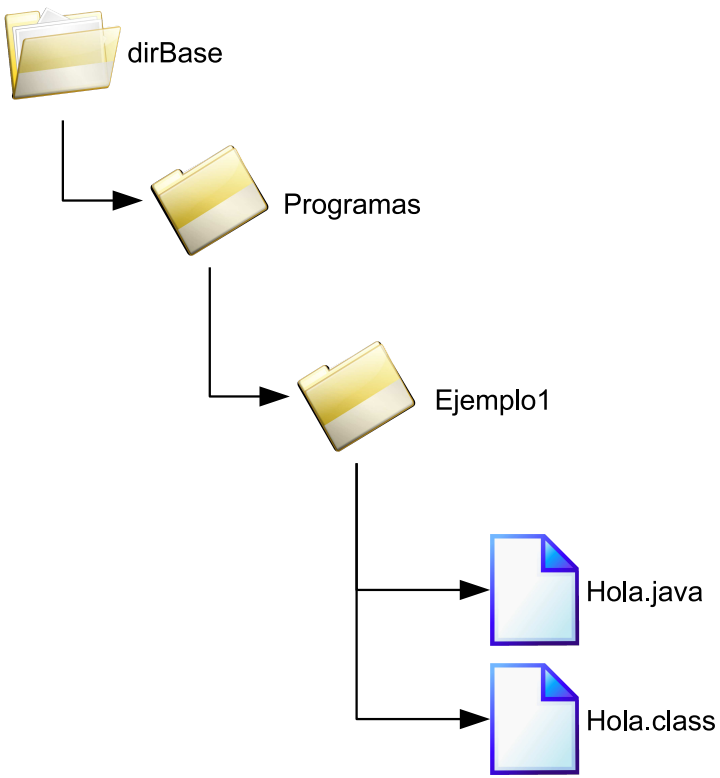
Hola.java

```
package Programas.Ejemplo1;

class Hola {
   public static void main( String[ ] args ) {
      System.out.println( "Hola, ¿qué tal?" );
   }
}
```

Directorio de trabajo

```
> cd path_hasta_dirBase
```

Ejemplo gráfico



Hola.java

```
package Programas.Ejemplo1;

class Hola {
   public static void main( String[ ] args ) {
      System.out.println( "Hola, ¿qué tal?" );
   }
}
```

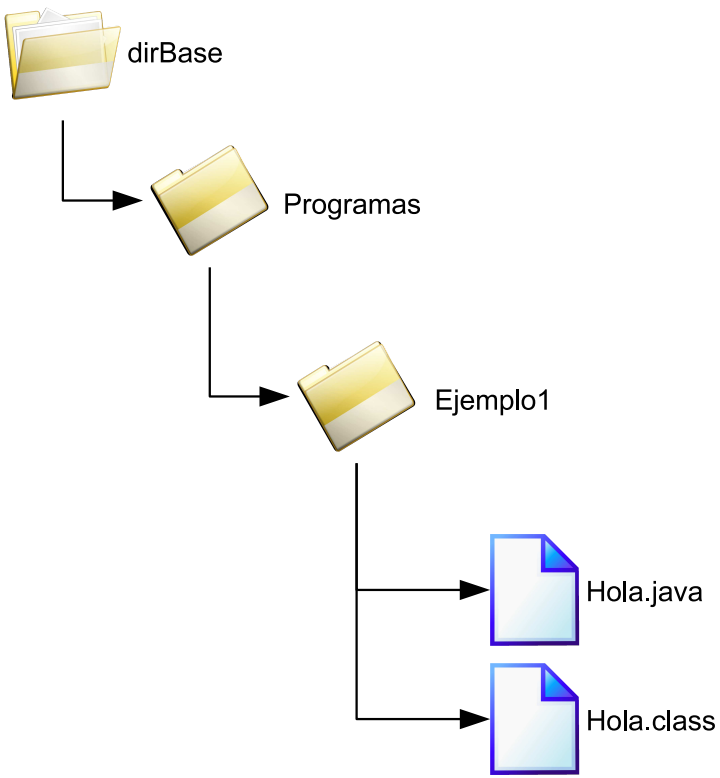
Directorio de trabajo

```
> cd path_hasta_dirBase
```

Compilación

```
> javac Programas\Ejemplo1\Hola.java
```

Ejemplo gráfico



Hola.java

```
package Programas.Ejemplo1;

class Hola {
   public static void main( String[ ] args ) {
      System.out.println( "Hola, ¿qué tal?" );
   }
}
```

Directorio de trabajo

```
> cd path_hasta_dirBase
```

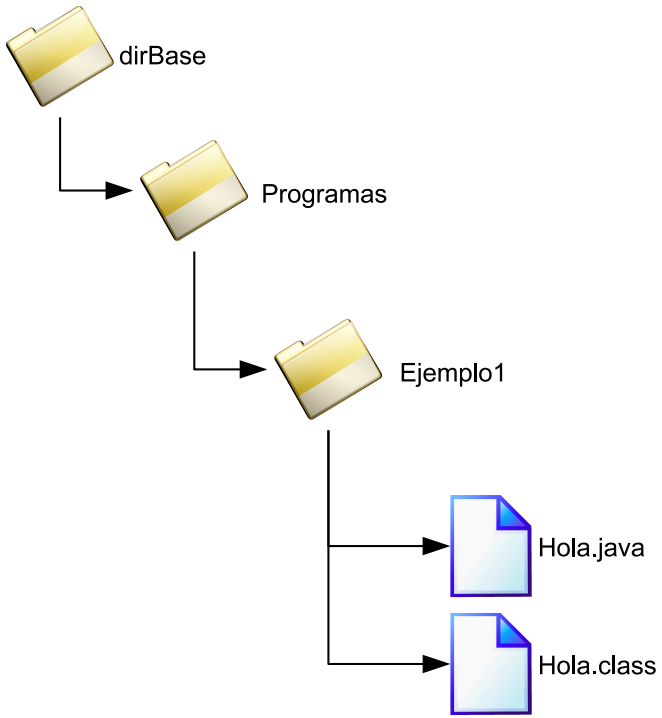
Compilación

```
> javac Programas\Ejemplo1\Hola.java
```

Ejecución

```
> java Programas.Ejemplo1.Hola
Hola, ¿qué tal?
>
```

Ejemplo gráfico



Hola.java

```
package Programas.Ejemplo1;

class Hola {
   public static void main( String[ ] args ) {
      System.out.println( "Hola, ¿qué tal?" );
   }
}
```

Directorio de trabajo

```
> cd path_hasta_dirBase
```

Compilación

```
> javac Programas\Ejemplo1\Hola.java
```

Ejecución

```
> java Programas.Ejemplo1.Hola
Hola, ¿qué tal?
>
```

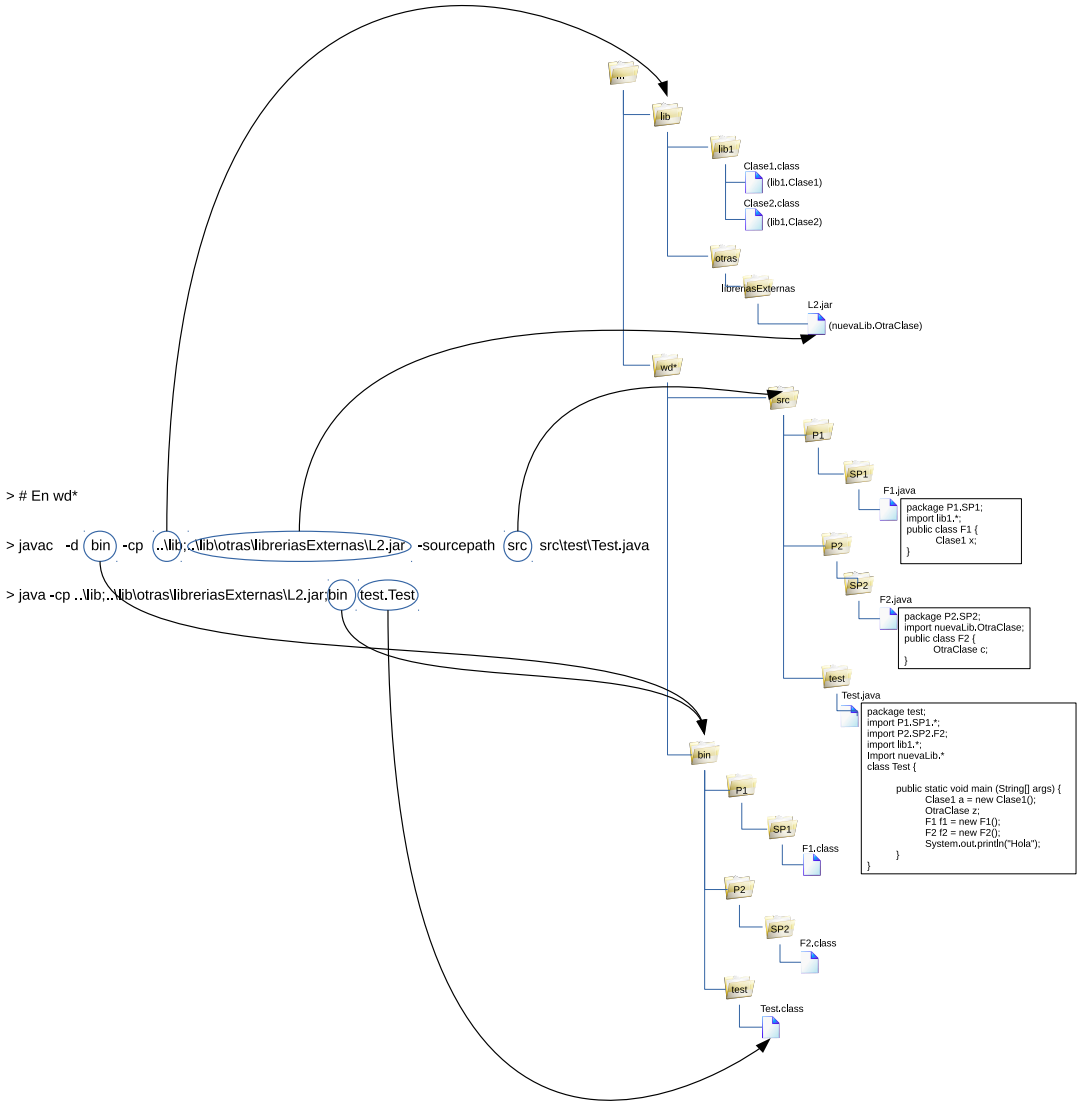
Alternativa

```
> cd path_hasta_OtroDirectorio
> java -classpath path_hasta_dirBase Programas.Ejemplo1.Hola
Hola, ¿qué tal?
>
```

Algunas opciones de compilación y ejecución

- Compilación `javac`:
  - `-sourcepath directorio`: directorio *base* a partir del cual puede buscar los fuentes para llevar a cabo la compilación *voraz* (teniendo en cuenta la relación paquetes.subpaquetes con directorios\subdirectorios)
  - `-d directorio`: directorio será el directorio *base* de reubicación para todos los ficheros `.class` generados (respetando la estructura de paquetes.subpaquetes con directorios\subdirectorios)
  - `-classpath directorio` o bien `-cp directorio`: **secuencia** de directorios (separados por ; en Windows, separados por : en Linux/OS-X **¡sin espacios!**) donde buscar los ficheros `.class` necesarios para la verificación de la compilación
- Ejecución `java`:
  - `-classpath directorio`: **secuencia** de directorios (separados por ; en Windows, separados por : en Linux/OS-X **¡sin espacios!**) donde buscar los ficheros `.class` necesarios para la ejecución

Ejemplo gráfico: en carpeta `compilacion`



Parámetros del método main

Argumentos.java

```
class Argumentos {

    public static void main ( String argumentos[ ] ) {

        for ( int i = 0; i < argumentos.length; i++ ) {
            System.out.println( "Argumento " + i + ": " + argumentos[ i ] );
        }

    }

}
```

```
> javac Argumentos.java
> java Argumentos hola 1 -3.4 fichero.txt
```

Índice

6 Más características del lenguaje Java
- Excepciones
- Programación multihilo
- Compilación y ejecución
- Depuración en Java

Depuración *debugging*

- Etimología *debug*: desinsectar (historia...)
- Utilidad: seguimiento en la ejecución de un programa Java
- Numerosas herramientas: *Java Platform Debugger Architecture*
- `jdb`: depuración en línea de comandos
- Variable de entorno: `JAVA_HOME`
- Variable de entorno: `PATH=$PATH:$JAVA_HOME/bin`
- Ejecución (numerosas posibilidades): `jdb [opciones] [clase] [argumentos]`
- Ayuda: `jdb -help`
- Comandos dentro de jdb:
  - `...> jdb`
  - `Initializing jdb ...`
  - `> help`

Comandos

- Puntos de ruptura *(breakPoints)*: `stop at/in`
  - `stop at <nombre de clase>:<numero de linea>`
  - `stop in <nombre de clase>.<nombre de metodo>`. Sobrecarga: especificación de tipo/clase
  - `stop in <nombre de clase>.<init>`: constructor
  - `stop in <nombre de clase>.<clinit>`: inicialización estática de la clase
  - `clear <nombre de clase>:/.<numero de linea>/nombre de metodo>` limpia punto de ruptura
- Arranque:
  - `run`: arranca la ejecución (hasta final o hasta que encuentre *breakpoint*)
- Ejecución:
  - `step`: Ejecución de sólo una instrucción; si hay llamada a método, salta a método
  - `next`: Ejecución de sólo una instrucción
- Listado:
  - `list`
- Impresión de variables:
  - `print`: impresión de variables primitivas y objetos (toString())
  - `dump`: idem que `print` para variables primitivas, impresion de atributos para objetos
  - `locals`: impresión de parametros y variables locales (requiere compilación con `javac -g`)

Inspector de ficheros `.class`: `javap`

- El comando
  - `> javap Fichero.class`

  permite observar algunas propiedades de los ficheros `.class` resultantes de la compilación
- Véase `> javap -help` para una explicación de las posibles opciones

Índice

7 Ficheros .jar
- Conceptos básicos
- Firma de ficheros jar

Índice

7 Ficheros .jar
- Conceptos básicos
- Firma de ficheros jar

Ficheros jar

- `jar`: **j**ava **ar**chive
- Permite *empaquetar* ficheros relativos a la ejecución en el entorno Java
- Aspectos relativos a los ficheros `jar`:
  - Creación
  - Visualización de contenidos
  - Extracción de contenidos
  - Actualización
  - Ejecución

Creación de ficheros jar

Creación de ficheros jar

```
> jar cf[v][0][M][m][e] fichero.jar [-C dir] ficheros_o_directorios
```

con

- v: modo verboso
- 0: sin compresión
- M: sin fichero de manifiesto *(manifest file)*
- m: adición de fichero de manifiesto al generado
- e: punto de entrada *(entry point)*
- -C: cambio de directorio

Ejemplo

```
# Incluye el fichero Clase.class y los directorios (y sus contenidos) dir1 y dir2
> jar cf paquete.jar Clase.class dir1 dir2

# Incluye en el fichero paquete2.jar todos los ficheros que cuelguen de dir1\dir2.
# La opción -C <dir> <fichero/directorio> sólo permite UN fichero o UN directorio tras -C <dir>
# (si se indica . se incluyen todos los directorios que cuelgan de <dir>
# En el siguiente ejemplo se incluye todo lo que cuelgue de "dir1\dir2"
> jar cf paquete2.jar -C dir1\dir2 .
```

Visualización y actualización

Visualización

```
> jar tvf paquete.tar
META-INF/MANIFEST.MF
Clase.class
dir1/
dir1/Clase1.class
dir1/Clase1.java
dir2/
dir2/Clase2.class
dir2/Clase2.java
```

Actualización de dir1/Clase1.class

```
> jar uf paquete.jar dir1/Clase1.class
```

Ejemplo: carpeta compilacion

- Creación de fichero .jar

```
> jar cf app.jar ..\lib\otras\libreriasExternas\L2.jar -C bin . -C ..\lib lib1
```

- Verificación:

```
> jar tvf app.jar
     0 Thu Feb 14 16:38:58 CET 2019 META-INF/
    69 Thu Feb 14 16:38:58 CET 2019 META-INF/MANIFEST.MF
   743 Thu Feb 14 14:45:26 CET 2019 otras/libreriasExternas/L2.jar
     0 Thu Feb 14 14:52:16 CET 2019 P1/
     0 Thu Feb 14 14:52:16 CET 2019 P1/SP1/
   213 Thu Feb 14 16:33:16 CET 2019 P1/SP1/F1.class
     0 Thu Feb 14 14:52:16 CET 2019 P2/
     0 Thu Feb 14 14:52:16 CET 2019 P2/SP2/
   220 Thu Feb 14 16:33:16 CET 2019 P2/SP2/F2.class
     0 Thu Feb 14 14:52:16 CET 2019 test/
   510 Thu Feb 14 16:33:16 CET 2019 test/Test.class
     0 Thu Feb 14 14:48:02 CET 2019 lib1/
   191 Thu Feb 14 14:47:10 CET 2019 lib1/Clase1.class
```

- Utilización:

```
> java -cp app.jar test.Test
```

- Otro ejemplo: sin incluir todas

```
> jar cf app_incompleta.jar ..\lib\otras\libreriasExternas\L2.jar -C bin .
> java -cp app_incompleta.jar;..\lib test.Test
```

Extracción

Extracción

```
> jar xf fichero.jar [ archivo1 ] [ archivo2 ] ...
```

Sólo se extraen `archivo1`, `archivo2` si son explicitados; en caso contrario se extraen todos:

Extracción

```
> jar xvf paquete.tar
META-INF/MANIFEST.MF
Clase.class
dir1/
dir1/Clase1.class
dir1/Clase1.java
dir2/
dir2/Clase2.class
dir2/Clase2.java
```

Ejecución: java -jar ...

Ejecución: fichero MANIFEST.MF

```
> java -jar paquete.jar
```

- Sin indicacación explícita del punto de entrada *(entry point)*, es decir clase con método `public static void main( String [ ] args )`.
- Esta información es necesaria y puede indicarse en el fichero `META-INF\MANIFEST.MF`

META-INF\MANIFEST.MF

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
```

- Posibilidad cuando se crea el fichero `.jar` (ejemplo `compilacion`:

```
> jar cef test.Test app.jar ..\lib\otras\libreriasExternas\L2.jar -C bin . -C ..\lib lib1

// Importante el orden de las opciones: cef o cfe: mismo orden que parámetros

> jar cfe app.jar test.Test ..\lib\otras\libreriasExternas\L2.jar -C bin . -C ..\lib lib1

// Entonces:

> java -jar app.jar
Hola
```

Comprobación:

```
> jar xf app.jar META-INF\MANIFEST.MF
> type META-INF\MANIFEST.MF
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
Main-Class: test.Test
```

Fichero MANIFEST.MF (cont.)

- Alternativas:
  - Actualizar filas oportunas del fichero MANIFEST.MF

Contenido del fichero linea_manifest.txt

```
Main-Class: test.Test

```

¡¡¡Importante la línea en blanco tras la última línea del fichero anterior MANIFEST.MF!!!

Actualización del fichero META-INF\MANIFEST.MF

```
> jar ufm app.jar linea_manifest.txt

¡¡¡Cuidado con orden de parámetros. Equivalentemente:

> jar umf linea_manifest.txt app.jar

// Entonces

> java -jar app.jar
Hola
```

  - Añadir el fichero explícitamente en el momento de la creación

Fichero de manifiesto: mi_manifiesto.txt

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
Main-Class: test.Test

```

¡¡¡Importante la línea en blanco tras la última línea del fichero anterior!!!

```
> jar cmf mi_manifiesto.txt app.jar ..\lib\otras\libreriasExternas\L2.jar -C bin . -C ..\lib lib1

// Importante el orden de las opciones: cmf o cfm: mismo orden que parámetros

> jar cfm app.jar mi_manifiesto.txt ..\lib\otras\libreriasExternas\L2.jar -C bin . -C ..\lib lib1

// Entonces

> java -jar app.jar
Hola
```

Más campos en MANFIEST.MF

- Recordemos que:

```
> jar cf app.jar ..\lib\otras\libreriasExternas\L2.jar -C bin . -C ..\lib lib1
> jar cf app_incompleto.jar ..\lib\otras\libreriasExternas\L2.jar -C bin .
> java -cp app_incompleto.jar;..\lib test.Test
```

Entonces:

lineas_manifest.txt

```
Main-Class: test.Test
Class-Path: ..\lib\
```

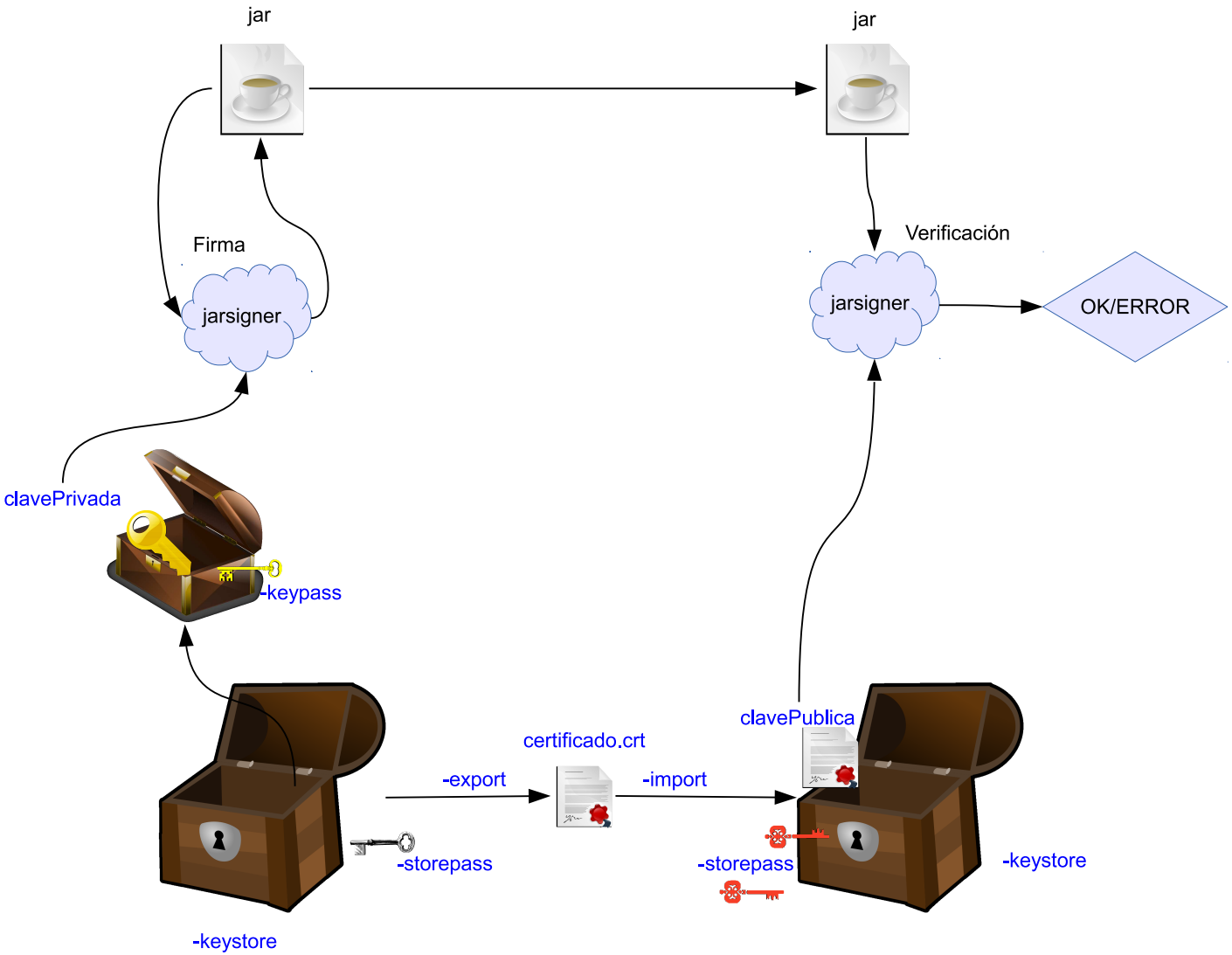
En `Class-Path` se pueden especificar ficheros `.jar` y directorios con clases, separados por espacios
¡¡¡Importante finalizar el directorio con un *backslash* (\)!!!
Por lo que

```
> java umf lineas_manifest.txt app_incompleto.jar
> java -jar app_incompleto.jar
Hola
```

Índice

7 Ficheros .jar
- Conceptos básicos
- Firma de ficheros jar

Idea conceptual



Creación de certificado digital

Creación de certificado digital

```
> keytool -genkey -alias pacorsa -validity 365 -v    ### alias mejor en minúscula
>
> ### Llavero y ubicación no por defecto
> keytool -keystore almacen -genkey -alias pacorsa -keyalg RSA -keysize 1024
-sigalg SHA1withRSA -validity 365 -v

...

> ### Visualización
> keytool -keystore almacen -list -alias pacorsa -v
...
> ### Exportación del certificado
> keytool -keystore almacen -export -alias pacorsa -v -file paco.crt
...
> ### Importación del certificado
> keytool -keystore almacenOtro -import -alias pacorsaotro -v -file paco.crt
```

Firma y verificación

Firma

```
> ### Estado actual de hola.jar
> jar tf hola.jar
META-INF/MANIFEST.MF
...


> ### Firma
> jarsigner -keystore almacen hola.jar pacoRSA [ -tsa http://timestamp.digicert.com ]

> ### Estado actual de hola.jar
> jar tf hola.jar
META-INF/MANIFEST.MF
META-INF/PACORSA.SF
META-INF/PACORSA.RSA
...

> ### Verificación sin certificado digital
> jarsigner -verify -verbose hola.jar
> ### Verificación con certificado digital
> jarsigner -verify -keystore almacenOtro -verbose -certs hola.jar pacoRSA
...

> ### Alteración de un fichero:
> jar uf hola.jar fichero.java
> jarsigner -verify -verbose hola.jar
jarsigner: java.lang.SecurityException: SHA-256 digest error for fichero.java
```

Índice

8 Javadoc

¿Qué es Javadoc?

- Herramienta para generar documentación HTML a partir de comentarios en el fichero fuente Java
- Discriminación de comentarios:
  - Comentarios de documentación (javadoc): `/**...*/`
  - Comentarios de implementación: `/*...*/`
- Dentro de comentarios de documentación:
  - Descripciones de clases, constructores, campos, interfaces, métodos, ...

Ejemplo genérico de documentación

```
/** Class Description of MyClass */
public class MyClass {

    /** Field Description of myIntField */
    public int myIntField;

    /** Constructor Description of MyClass() */
    public MyClass() {
        // Do something ...
    }
}
```

Etiquetas javadoc

- Palabras clave reconocidas por `javadoc` de definen la información que sigue.
- Aparecen tras una descripción (con una separación de una línea en blanco)
- Algunas etiquetas con ejemplos:
  - `@author Pepe`
  - `@version v1.0`
  - `@param x valor de entrada necesario para...`
  - `@return el resultado de la solución...`
  - `@exception IOException provocado por inexistencia de fichero...`
  - `@throws IOException no recogida tras la lectura...`
- Ejemplo de ejecución de `javadoc` (véase `> javadoc -help`)
  - directorio destino de toda la documentación HTML: `-d doc`
  - `-private`: se documenta todo (privado, publico, paquete, protegido, ...)
  - charset a especificar en HTML: `-charset utf8`
  - Especificación de fuentes a documentar:
    - Indicando `-sourcepath` vectores
    - A partir de ahí, especificando *paquetes*/directorios/ficheros

```
$ cd javadocVectores
$ javadoc -charset utf8 -d doc -private -sourcepath vectores test espacioVectorial
```

Ejemplo javadoc

javadocVectores\vectores\test\Test2D.java

```
package test;

import espacioVectorial.*;

/**
 * <h1> Test para coordenadas en 2 dimensiones </h1>
 * En este test se lleva a cabo la verificación de que
 * se pueden generar y escribir coordenadas en 2D
 *
 * @author Paco Martínez
 * @version 1.0
 * @since febrero 2019
 *
 * @see test.Test3D
 *
 */
class Test2D {



    /**
      * Pruebas para {@link espacioVectorial.Coordenada2D}.
      *
      * Este método verifica el correcto funcionamiento de la clase
      * {@link espacioVectorial.Coordenada2D}, escribiendo coordenadas
      * en 2D, obteniendo sus componentes y su norma Euclídea.
      */
    public static void main (String[] args) {
        Coordenada2D xy = new Coordenada2D( 3.4, -5.6 );
        System.out.println( "Coordenada 2D: " + xy );
        System.out.println( "Coordenada x: " + xy.getX() );
        System.out.println( "Coordenada y: " + xy.getY() );
        System.out.println( "Norma: " + xy.norma() );

    } // main

} // Test2D
```

javadocVectores\vectores\espacioVectorial\GeometriaEuclidea.java

```
package espacioVectorial;

/**
  * Interfaz cuya implementación implica el desarrollo del método sumaDeCuadrad
  *
  * Incluye como método <code>default</code> el  método <code>double norma()</co
  */
public interface GeometriaEuclidea {


    /**
     * Debe devoler la suma de cada coordenada al cuadrado
     *
     * @return la suma de los cuadrados de las coordenadas
     */
    double sumaDeCuadrados();


    /**
      * Sea cual sea la dimensionalidad de la coordenada, la norma Euclídea
      * será siempre la raíz cuadrada de la suma de los cuadrados de todas
      * las coordenadas. Por ello se incluye como método <code>default</code>
      * @return la norma euclídea de la coordenada
      */
    default double norma() {
        return Math.sqrt( sumaDeCuadrados() );
    }

}
```

Índice

9 Estilo de programación

Estilo

- Fuente:
  `https://www.oracle.com/technetwork/java/codeconventions-150003.`
- 80 % coste software: mantenmiento
- El software no suele ser mantenido por el programador original
- Las convenciones de estilo del código ayudan a entender más rápidamente el software