

Tema 2

Introducción al lenguaje Java

Programación Orientada a Objetos

José Francisco Vélez Serrano



Universidad
Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

- Introducción a Java
- Expresiones básicas y control de flujo
- Las clases
 - Detalles sobre las propiedades
 - Detalles sobre los métodos
- Los arrays
- Los paquetes

Introducción a Java

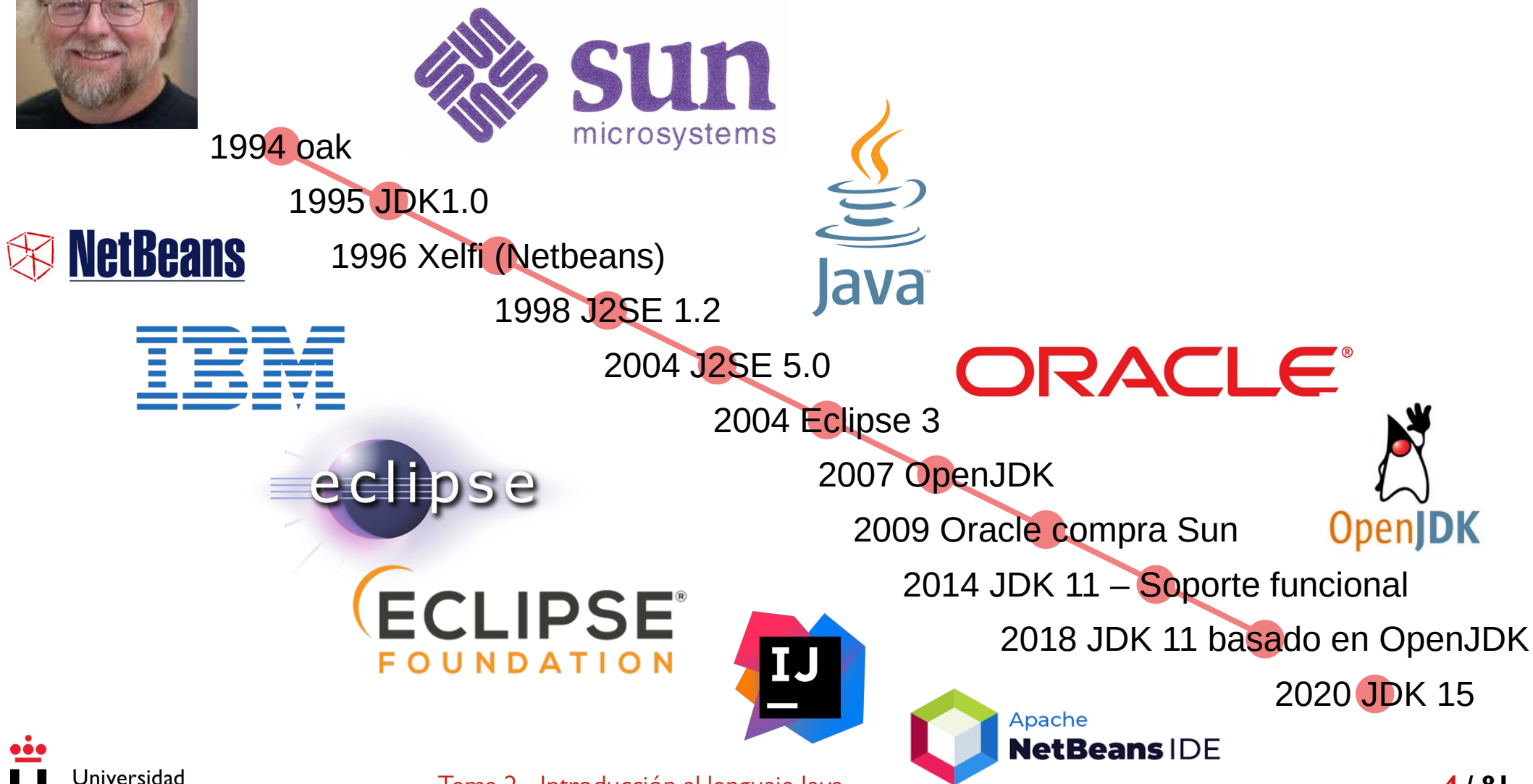
Una introducción a Java: su historia, los paquetes, la forma de compilar, la máquina virtual...

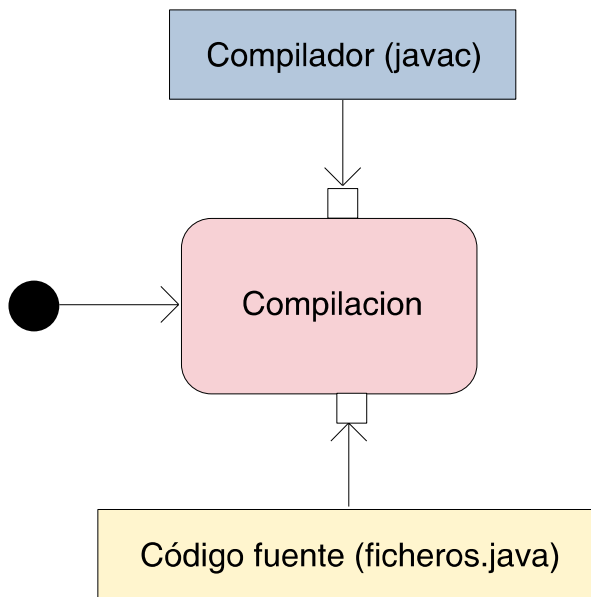
Introducción

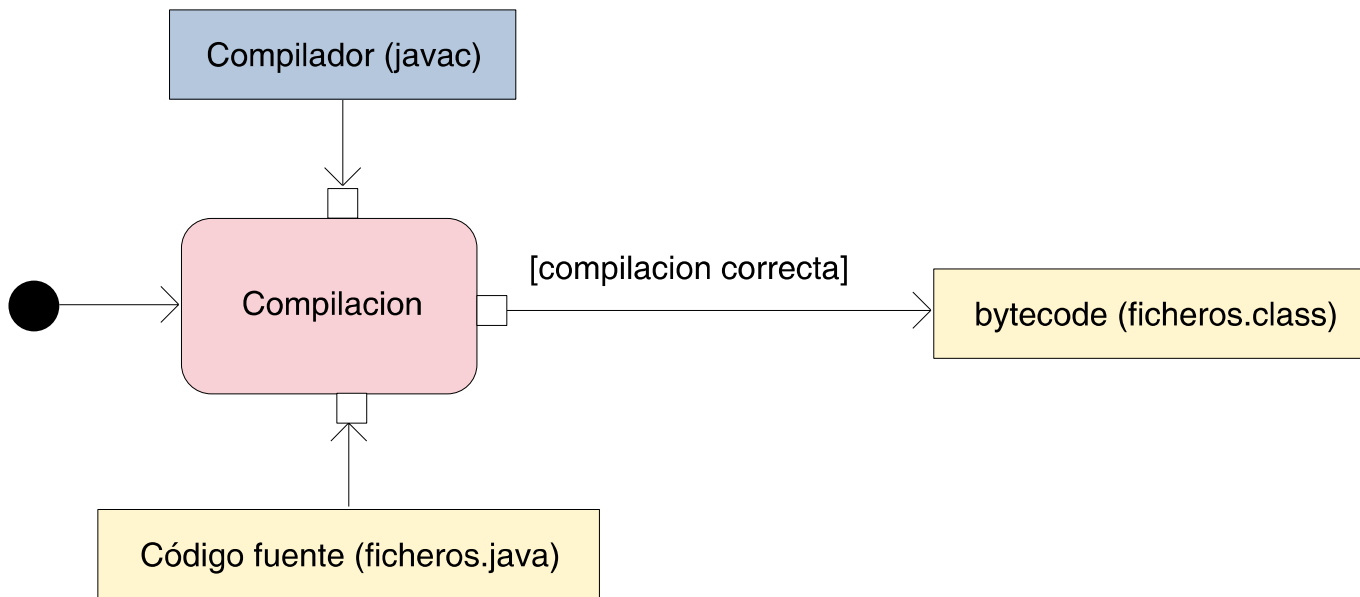
Historia



James Gosling

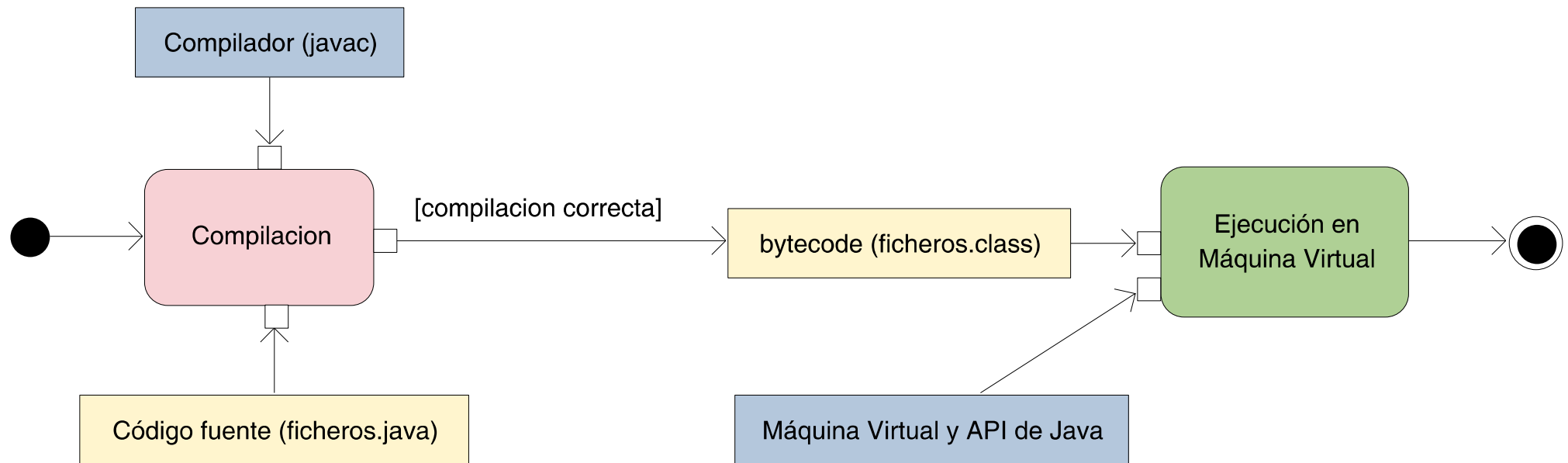






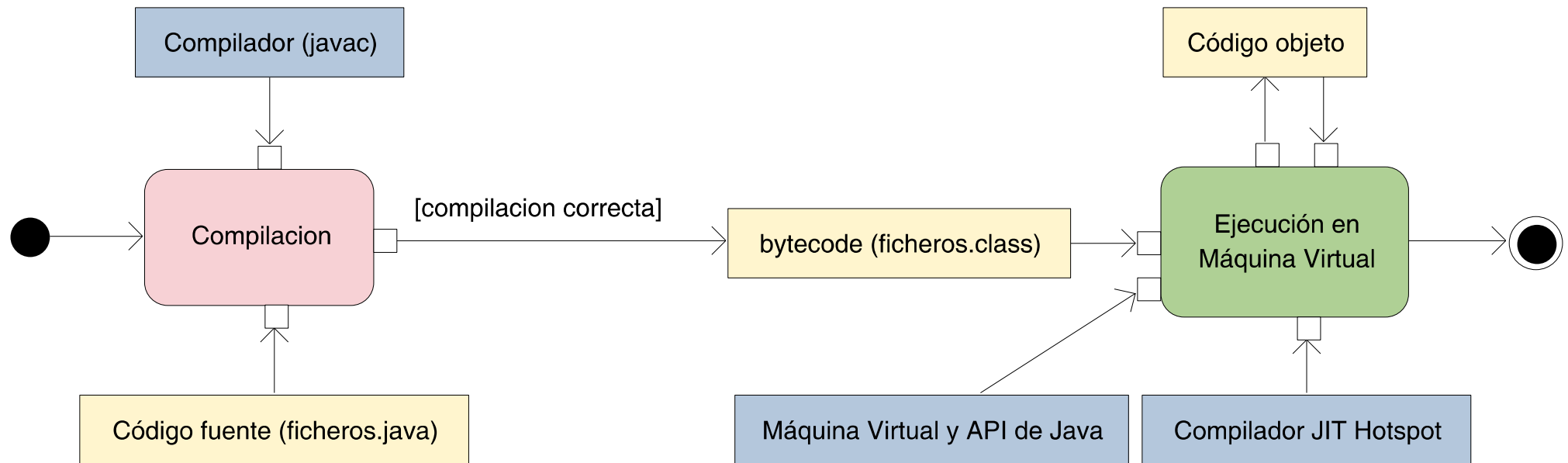
Introducción

Compilación y ejecución



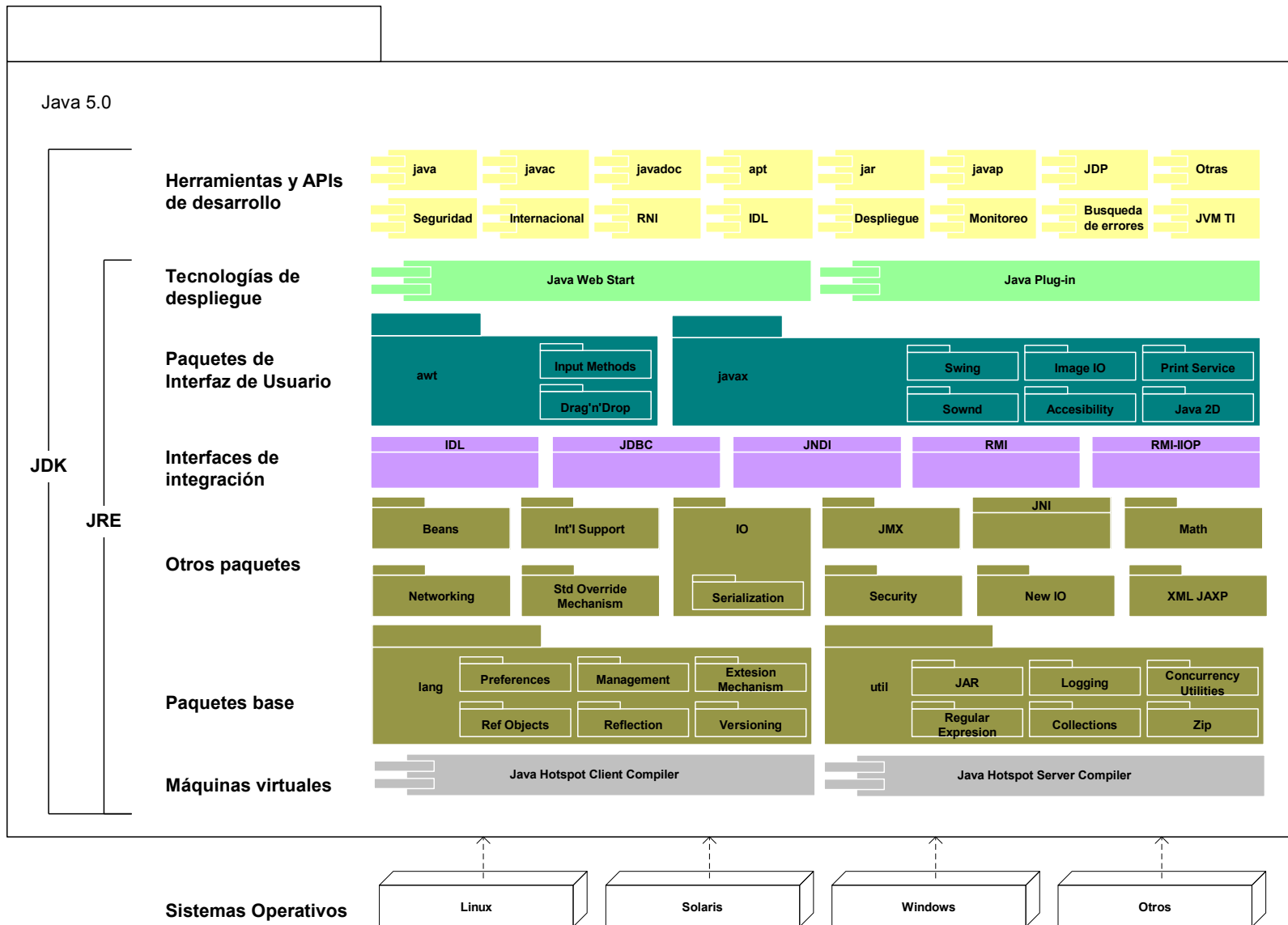
Introducción

Compilación y ejecución



Introducción

Los paquetes de Java



Introducción

El programa “Hola mundo”



Desde la línea de comandos creamos un fichero de nombre HolaMundo.java.

```
C:\> R:
R:\> mkdir Temp
R:\> cd Temp
R:\Temp> notepad HolaMundo.java
```

El fichero contiene el siguiente programa en Java:

```
/** Programa en Java que muestra un mensaje de bienvenida */
public class HolaMundo {
    // El método en el que comienza la ejecución del programa es main
    public static void main (String [ ] arg)      {
        System.out.println("Hola mundo");
    }
}
```

Compila y ejecuta el programa desde la línea de comandos ejecutando:

```
R:\Temp> PATH=C:\Program Files\Java\jdk-11.0.6\bin;%PATH%
R:\Temp> javac HolaMundo.java
R:\Temp> java HolaMundo
```

Introducción

Los comentarios



En Java existen dos formas de crear comentarios en el código.

```
/*  
    Esto es un comentario que puede ocupar  
    varias líneas.  
*/  
  
// Esto es un comentario de línea
```

Java propone una forma de escritura de comentarios que permite al programa javadoc del SDK generar de manera automática documentación del código.

```
/** Ejemplo de documentación  
    automática con javadoc  
    @author José Vélez  
    @versión 1.0  
*/
```

```
/**  
 * Descripción de la función principal  
 * @param nombre_parámetro descripción  
 * @return descripción  
 * @throws nombre_clase descripción  
*/
```

Genera la documentación del programa HolaMundo usando:

```
R:\Temp> javadoc -author -version HolaMundo.java  
R:\Temp> dir
```

Introducción

Juego de instrucciones de Java

<code>abstract</code>	<code>false</code>	<code>null</code>	<code>true</code>
<code>boolean</code>	<code>final</code>	<code>package</code>	<code>try</code>
<code>break</code>	<code>finally</code>	<code>private</code>	<code>void</code>
<code>byte</code>	<code>float</code>	<code>public</code>	<code>volatile</code>
<code>case</code>	<code>for</code>	<code>return</code>	<code>while</code>
<code>catch</code>	<code>if</code>	<code>short</code>	
<code>char</code>	<code>implements</code>	<code>static</code>	
<code>class</code>	<code>instanceof</code>	<code>super</code>	
<code>continue</code>	<code>int</code>	<code>switch</code>	
<code>default</code>	<code>interface</code>	<code>synchronized</code>	
<code>do</code>	<code>long</code>	<code>this</code>	
<code>double</code>	<code>native</code>	<code>threadsafe</code>	
<code>extend</code>	<code>new</code>	<code>transient</code>	

Introducción

Tipos primitivos

abstract

boolean

break

byte

case

catch

char

class

continue

default

do

double

extend

false

final

finally

float

for

if

implements

instanceof

int

interface

long

native

new

null

package

private

public

return

short

static

super

switch

synchronized

this

threadsafe

transient

true

try

void

volatile

while

Introducción

Control de flujo

abstract

boolean

break

byte

case

catch

char

class

continue

default

do

double

extend

false

final

finally

float

for

if

implements

instanceof

int

interface

long

native

new

null

package

private

public

return

short

static

super

switch

synchronized

this

threadsafe

transient

true

try

void

volatile

while

Introducción

Capacidades de POO

abstract

boolean

break

byte

case

catch

char

class

continue

default

do

double

extend

false

final

finally

float

for

if

implements

instanceof

int

interface

long

native

new

null

package

private

public

return

short

static

super

switch

synchronized

this

threadsafe

transient

true

try

void

volatile

while

Expresiones básicas

Declaración de expresiones y variables de los tipos más comunes en Java.

Expresiones básicas

Variables y expresiones con tipos primitivos



En Java todos los datos de un programa son objetos salvo 7 **tipos primitivos** que se añadieron por razones de **eficiencia**.

La definición de una variable a un tipo primitivo sigue la siguiente sintaxis:

```
<tipo> <identificador> [, <identificador>];  
<identificador> = <expresión de inicialización>;  
  
[final] <tipo> <identificador> = <expresión de inicialización >;
```

Ejemplos:

```
int mainCont, auxCont;    //Declaración de dos enteros  
mainCont = 2;             //Inicialización de un entero  
int i = 5;                //Declaración e inicialización de un entero  
final double  $\pi$  = 3.1416; //Declaración e inicialización de una constante real
```

Tipo carácter

Tipo	Tamaño en bytes	Descripción	Valor por defecto	Sintaxis	Ejemplo de uso
char	2	Caracteres UNICODE	'\u0000000'	'[\u[0-9]+ .]'	char c = 'ñ'; char D = '\u13';

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

[illegible]

<http://joerg.piringer.net/unicode/alldisplayablechars.txt>

Expresiones básicas

Tipos numéricos



Tipo	Tamaño en bytes	Descripción	Valor por defecto	Sintaxis	Ejemplo de uso
byte	1	Enteros desde -128 a 127	0	-{0,1}[0-9] ⁺	byte B = 100;
short	2	Enteros desde -16384 a 16383	0	-{0,1}[0-9] ⁺	short s = -8000;
int	4	Enteros desde -2 ³¹ hasta 2 ³¹ -1	0	-{0,1}[0-9] ⁺	int i = 1400000;
long	8	Enteros desde -2 ⁶³ hasta 2 ⁶³ -1	0	-{0,1}[0-9] ⁺	long l = -531L;
float	8	Reales	+0.0f	-{0,1}[0-9]*\.[0-9] ⁺ f	float x = -1.21f;
double	16	Reales largos	0.0	-{0,1}[0-9]*\.[0-9] ⁺	double pi = +3.14;

byte > short y char > int > long > float > double

```
float temperatura = 257.0f;
byte temperaturaTruncada = (byte) temperatura;
char caracter = 'A' + 10;
System.out.println(temperatura);
System.out.println(temperaturaTruncada);
System.out.println(caracter);
```

```
257.0
1
K
```

Expresiones básicas

Aritmética con tipos numéricos



Operador	Detalle	Binario	Unitario	Ejemplo
+	Suma o declaración de positivo	X	X	a + b, +a
-	Resta o declaración de negativo	X	X	a - b, -a
*	Producto	X		a * b
/	División	X		a / b
%	Resto de la división de dos números enteros	X		a % b
+=	Incremento en varias unidades de una variable		X	a += b
-=	Decremento en varias unidades de una variable		X	a -= 5
*=	Multiplicación de una variable sobre ella misma		X	a *=3
/=	División de una variable sobre ella misma		X	a /= b
%=	Resto la división de una variable sobre ella misma		X	a %=4
Solo para enteros				
++	Post o pre-incremento en una unidad de una variable		X	a++, ++a
--	Post o pre-decremento en una unidad de una variable		X	a--, --a

```
int A = 12;  
A *= 2;  
int B = A++;  
System.out.println(A);  
System.out.println(B);
```

25
24

Expresiones básicas

Operaciones relacionales en tipos numéricos



Operador	Descripción	Ejemplo
==	Igualdad	a == b
>	Mayor	a > b
<	Menor	a < b
>=	Mayor o igual	a >= b
<=	Menor o igual	a <= b
!=	Distinto	a != b

```
int a = 12;  
int b = ++a;
```

```
System.out.println(a == b);  
System.out.println(a >= b);  
System.out.println(a != b);  
System.out.println(a <= b);
```

```
true  
true  
false  
true
```

Expresiones básicas

Operaciones de bit con tipos numéricos

Operador	Detalle	Binario	Unitario	Ejemplo
&	Conjunción (and)		X	a&b
	Disyunción (or)		X	a b
^	Disyunción exclusiva (xor)		X	a^b
<<	Desplazamiento binario a la izquierda rellenando con ceros		X	a<<3
>>	Desplazamiento binario a la derecha rellenando con el bit más significativo		X	a>>3
>>>	Desplazamiento binario a la derecha rellenando con ceros		X	a>>>3
~	Negación binaria	X		~a

```
int A = 0x5E;  
byte B = -3;  
short C = 0b111;  
System.out.println(A);  
System.out.println(B);  
System.out.println(C<<1);  
System.out.println(A | (B & C));
```

```
94  
-3  
14  
95
```

Expresiones básicas

Tipo booleano y operaciones



Tipo	Tamaño en bytes	Descripción	Valor por defecto	Sintaxis	Ejemplo de uso
boolean	indefinido	cierto o falso	false	true false	boolean c = true;

Operador	Descripción	Unitario	Binario	Ejemplo
&	Conjunción (and)		X	a&b
	Disyunción (or)		X	a b
&&	Conjunción impaciente (and)		X	a&&b
	Disyunción impaciente (or)		X	a b
^	Disyunción exclusiva (xor)		X	a^b
!	Negación (not)	X		!a
&=	Asignación con conjunción		X	a&=b
=	Asignación con disyunción		X	a =b

```
boolean A = true;
boolean B = false;
boolean C = B && (A || B); //Deja de evaluarse en el &&
boolean D = A | (3 == 2+1);
System.out.println(C);
System.out.println(D);
```

false
true

Control de flujo

Presentación de las instrucciones de control del flujo del programa en Java.

Un ámbito se inicia con el carácter de **llave** abierta hacia la derecha y se termina con el carácter de llave abierta hacia al izquierda.

Los ámbitos se utilizan para:

- **Definir qué sentencias** están afectadas por una **declaración** (función, clase...) o por una instrucción de **control de flujo** (if, while...).
- **Agrupar** lógicamente sentencias.

Los ámbitos se pueden **anidar** y dentro de un ámbito siempre puede definirse otro.

Las declaraciones de **las variables son locales al ámbito** en que se declaran.

Control de flujo

Ámbitos y sentencias

```
{  
    int a;  
    a = 25;  
    int c = 15;  
  
    //Aquí 'a' vale 25 y 'c' vale 15  
    {  
        int b;  
        b = 2;  
        // int c = 12; Java no deja definir 2 variables con mismo nombre en un ámbito  
  
        //Aquí 'a' vale 25, 'b' vale 2  
    }  
    //Aquí 'a' vale 25, 'c' vale 15 y 'b' no está declarada  
}
```

En Java, el tabulado a nivel de ámbito es una convención muy aconsejable, aunque no afecte a la compilación. El siguiente código es equivalente al anterior:

```
{int a;a = 25;int c = 15;{int b;b = 2;}}
```

La sentencia if lleva asociada una expresión booleana entre **paréntesis**. De cumplirse la expresión se ejecuta la sentencia o el ámbito siguiente al if. En otro caso se ejecuta, si existe, la rama else.

```
if (<expression booleana>
    <ámbito> | <sentencia>
[else
    <ámbito> | <sentencia>]
```

El siguiente ejemplo ilustra el uso de un if encadenado a otro.

```
if (a > b) {
    a = c;
}
else if (a < b) {
    a = d;
}
else {
    a = 0;
}
```

La palabra reservada `for` permite repetir una sentencia o un ámbito cualquier número de veces. Su estructura es la siguiente. Es buena práctica solo usarlo cuando se conoce el número de iteraciones de antemano y solo depende de una condición.

```
for (<expresión inicialización>;<expresión booleana>;<expresión incremento>)  
    <ámbito> | <sentencia>
```

El siguiente ejemplo ilustra su uso.

```
for (int cont = 0; cont < 100; cont++) {  
    a = a * B /c; //La primera vez que se ejecuta esta línea cont vale 0  
                //La última vez que se ejecuta cont vale 99  
}  
  
//Aquí la variable cont no existe, pero si existiese valdría 100
```

La palabra reservada `while` permite repetir un ámbito cualquier número de veces. Es buena práctica usarlo cuando el número de iteraciones no está definido a priori o depende de varias condiciones. Su estructura es la siguiente.

```
while (<expresión booleana>)  
    <ámbito> | <sentencia>
```

```
do  
    <ámbito> | <sentencia>  
while (<expresión booleana>;
```

El siguiente ejemplo ilustra su uso.

```
while (a > b) {  
    b = D * 25;  
    a--;  
}
```

```
do {  
    b = D * 25;  
    a--;  
} while (a > b);
```

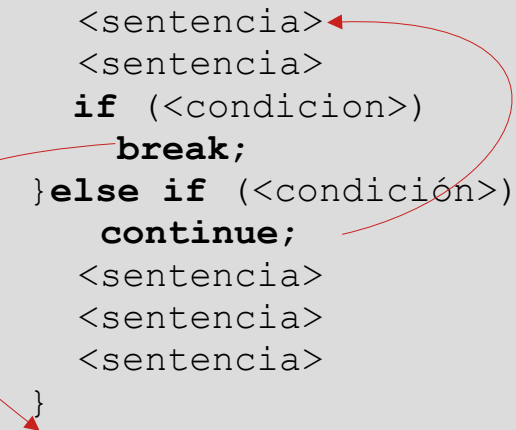
Control de flujo

break y continue

La instrucción **break** permite **interrumpir** en cualquier punto la ejecución normal de un bucle **for** o **while** y salir del mismo.

La instrucción **continue** permite interrumpir la ejecución normal de un bucle **for** o **while** y volver a la sentencia de evaluación para decidir si continuar o salir.

Tanto **break** como **continue** pueden utilizarse en combinación con **etiquetas** para salir de varios bucles simultáneamente hasta alcanzar el bucle etiquetado. Las etiquetas se definen mediante un identificador seguido del carácter de dos puntos.



```
while (<condición>) {  
    <sentencia>  
    <sentencia>  
    if (<condición>)  
        break;  
} else if (<condición>)  
    continue;  
    <sentencia>  
    <sentencia>  
    <sentencia>  
}
```

Control de flujo

Clásico switch-case-break-default

Permite evaluar una sola vez una expresión entera.

Basándose en su resultado, ejecuta las sentencias de un ámbito concreto.

Si las sentencias de un case no terminan en **break**, se ejecutan todas las sentencias siguientes, aunque pertenezcan a otros casos.

```
switch (variable)
{
    case 5 : {
        System.out.println("cinco");
        break;
    }
    case 6 : {
        System.out.println("seis");
        break;
    }
    default : {
        System.out.println("Más");
        break;
    }
}
```

```
switch (<expresión>)
{
    [case <valor constante> : [<ambito> | <sentencia>]*]*
    [default : [<ambito> | <sentencia>]*]
}
```



Hasta Java 8 solo se permitían expresiones basadas en tipos enteros.

En Java 8 se amplía el uso de expresiones a tipos enumerados y Strings.

En Java 12 se introduce: la sintaxis basada en flechas, múltiples valores por caso, y yield para devolver valores.

JDK 18 introduce la posibilidad de usar match de tipos y guardas.

```
switch (<expresión>) {  
    [case <match> [when cond][,<match> [when cond]]* -> sentencia | ámbito]*;  
    [default -> sentencia | ámbito ];  
}  
  
return switch (expresión) {  
    [case match [when cond][,match [when cond]]* -> ámbito_con_yield|valor_devuelto]*  
    [default -> ámbito_con_yield|valor_devuelto]  
}
```


Las clases

Sección que presenta la forma de crear las clases en Java.

Java permite definir clases con la interfaz e implementación de los objetos que posteriormente se podrán crear.

La definición de una clases en Java está constituida por:

- Identificación
- Miembros: Métodos o propiedades
- Clases internas
- Bloques de inicialización

```
[Modificador de clase] class <identificador> [parámetros] [herencia] [excepciones] {  
    [método|propiedad|inicializacion|clase] *  
}
```

En Java el identificador de una clase se suele escribir usando **UpperCamelCase**. Por ejemplo:

```
class Coche {  
    //Aquí iría la implementación de la clase Coche  
}
```

Java permite definir variables que sean **referencias a objetos**.

```
<nombre_clase> <identificador> [, <identificador>];  
<identificador> = <expresión de inicialización>;  
  
[final] <nombre_clase> <identificador> = <expresión de inicialización >;
```

Tipo	Tamaño en bytes	Descripción	Valor por defecto
referencia a objeto	4	Referencia a un objeto que cumple un tipo	null

La expresión de inicialización puede ser de dos tipos:

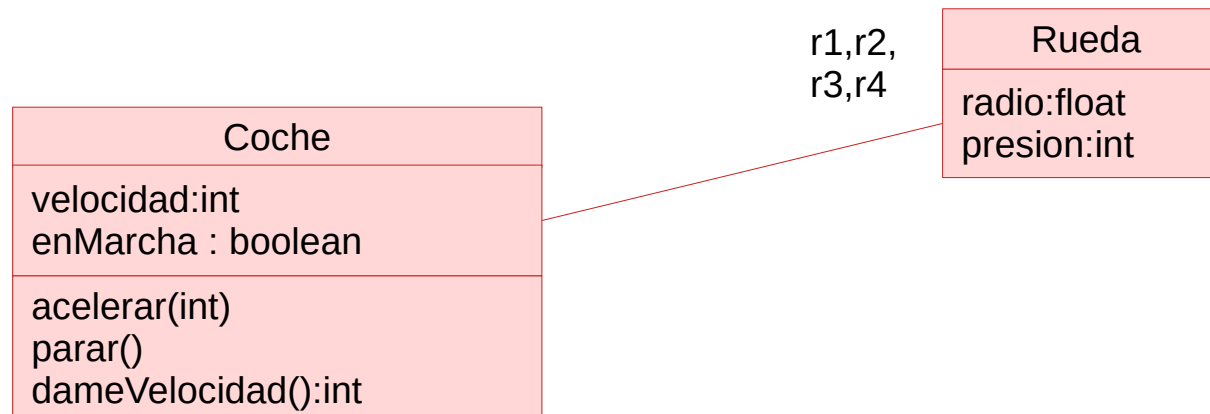
- **Creación** de un objeto, con la palabra reservada **new** seguida del nombre de la clase.
- **Referencia** a otro objeto ya creado.

```
Coche a, b, c;           //Se declaran tres variables que apuntarán a objetos.  
a = new Coche();         //Se crea un coche y la variable "a" lo apunta.  
b = new Coche();         //Se crea otro coche y la variable "b" lo apunta.  
c = a;                   //La variable "c" apunta al mismo coche que "a".
```

Los miembros de una clase pueden ser de dos tipos:

- **Propiedades:** variables que se declaran en el interior de una clase.
- **Métodos:** funciones que tiene parámetros, que devuelven elementos y que pueden contener código.

En Java, los miembros de una clase se suelen escribir usando **lowerCamelCase**.



Las propiedades, o campos, sirven para dotar de estado al objeto o a la propia clase.

Las propiedades son variables que se definen dentro de una clase y que pueden tomar valores independientes en cada objeto.

`propiedad ::= [mod control acceso] [mod uso] <tipo> <identificador> [= inicializa];`

```
class Coche {  
    boolean enMarcha;  
    int velocidad;  
    Rueda r1 = new Rueda();  
    Rueda r2 = new Rueda();  
    Rueda r3 = new Rueda();  
    Rueda r4 = new Rueda();  
}
```

```
class Rueda {  
    int presion = 2000;  
    float radio = 6.28f;  
}
```

Los métodos son **funciones** definidas dentro de la clase. Se invocan sobre los objetos de esa clase o sobre la clase misma.

método ::= [Mod acceso] [Mod uso] <tipo> <Id> ([params]) [except] [{[implementación]}]

parámetros ::= <tipo> <identificador>[, <tipo> <identificador>]*

excepciones ::= throws <tipo> [, <tipo>]*

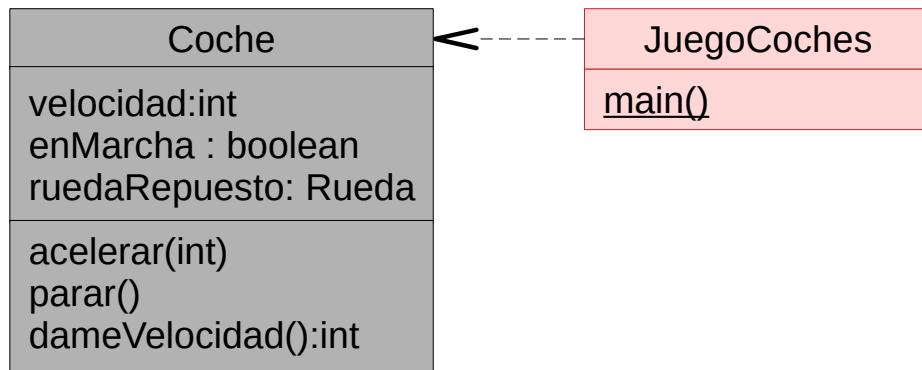
implementación ::= [sentencia]*

- El identificador del método está precedido por el **tipo** devuelto o **void** si no devuelve nada.
- Al identificador lo siguen los parámetros del método.
- Tras los parámetros puede haber un **ámbito** con la implementación.

```
class Coche {  
  
    void parar() {  
        // Código con la implementación  
    }  
  
    void acelerar(int incremento) {  
        // Código con la implementación  
    }  
  
    float reparar(float dinero, Taller t) {  
        // Código con la implementación  
    }  
}
```

Las clases

Ejemplo de definición de miembros



```
public class JuegoCoches {
    public static void main(String [] args) {
        Coche coche1 = new Coche();
        Coche coche2 = new Coche();
        Coche coche3 = c1;
        coche1.acelerar(10);
        coche2.acelerar(20);
        coche3.parar();
    }
}
```

```
class Coche {
    boolean enMarcha;
    int velocidad;
    Rueda r1 = new Rueda();
    Rueda r2 = new Rueda();
    Rueda r3 = new Rueda();
    Rueda r4 = new Rueda();

    void parar() {
        velocidad = 0;
        enMarcha = false;
    }

    void acelerar(int incremento) {
        velocidad += incremento;
        enMarcha = true;
    }

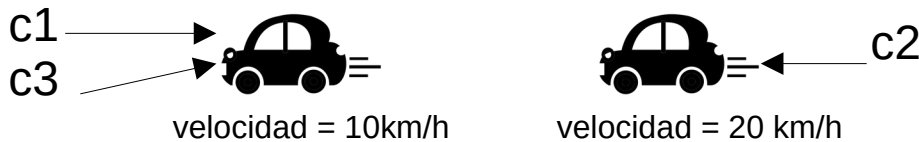
    int dameVelocidad() {
        return velocidad;
    }
}
```

Las clases

Diferencia entre objeto y referencia a objeto

En el ámbito de un método se opera con los parámetros que se le pasan y con miembros (métodos y propiedades) de la propia clase.

El método main de JuegoCoches crea 2 objetos de la clase Coche.



```
public class JuegoCoches {  
    public static void main(String [] args) {  
        Coche coche1 = new Coche();  
        Coche coche2 = new Coche();  
        Coche coche3 = c1;  
        coche1.acelerar(10);  
        coche2.acelerar(20);  
        coche3.parar();  
    }  
}
```

```
class Coche {  
    boolean enMarcha;  
    int velocidad;  
    Rueda r1 = new Rueda();  
    Rueda r2 = new Rueda();  
    Rueda r3 = new Rueda();  
    Rueda r4 = new Rueda();  
  
    void parar() {  
        velocidad = 0;  
        enMarcha = false;  
    }  
  
    void acelerar(int incremento) {  
        velocidad += incremento;  
        enMarcha = true;  
    }  
  
    int dameVelocidad() {  
        return velocidad;  
    }  
}
```


Las clases

Valores independientes de las propiedades

Cada objeto tiene valores independientes para sus propiedades.

El método main primero pone valor 10 en la propiedad velocidad de un coche, luego pone 20 en la propiedad velocidad del otro coche, finalmente pone a cero la propiedad velocidad del primero.

```
public class JuegoCoches {  
    public static void main(String [] args) {  
        Coche coche1 = new Coche();  
        Coche coche2 = new Coche();  
        Coche coche3 = c1;  
        coche1.acelerar(10);  
        coche2.acelerar(20);  
        coche3.parar();  
    }  
}
```

```
class Coche {  
    boolean enMarcha;  
    int velocidad;  
    Rueda r1 = new Rueda();  
    Rueda r2 = new Rueda();  
    Rueda r3 = new Rueda();  
    Rueda r4 = new Rueda();  
  
    void parar() {  
        velocidad = 0;  
        enMarcha = false;  
    }  
  
    void acelerar(int incremento) {  
        velocidad += incremento;  
        enMarcha = true;  
    }  
  
    int dameVelocidad() {  
        return velocidad;  
    }  
}
```

Las clases

Invocación de miembros de otros objetos



Para invocar a un miembro de otra clase se usa el identificador del objeto, un **punto** y el identificador del miembro que se desea invocar.

El método reparar de la clase Coche invoca a métodos de otro objeto (un Taller).

```
public class JuegoCoches {  
    public static void main(String [] args) {  
        Coche coche1 = new Coche();  
        Coche coche2 = new Coche();  
        Coche coche3 = c1;  
        coche1.acelerar(10);  
        coche2.acelerar(20);  
        coche3.parar();  
    }  
}
```

```
class Coche {  
    boolean enMarcha;  
    int velocidad;  
    Rueda r1 = new Rueda();  
    Rueda r2 = new Rueda();  
    Rueda r3 = new Rueda();  
    Rueda r4 = new Rueda();  
  
    void parar() {  
        velocidad = 0;  
        enMarcha = false;  
    }  
  
    void acelerar(int incremento) {  
        velocidad += incremento;  
        enMarcha = true;  
    }  
  
    int dameVelocidad() {  
        return velocidad;  
    }  
}
```

Las clases

Invocación de miembros del propio objeto



Para hacer referencia a un miembro (método o propiedad) de la propia clase se usa **simplemente el identificador** de dicho miembro.

Coche
velocidad:int enMarcha : boolean
acelerar(int) parar() dameVelocidad():int

```
public class JuegoCoches {  
    public static void main(String [] args) {  
        Coche coche1 = new Coche();  
        Coche coche2 = new Coche();  
        Coche coche3 = c1;  
        coche1.acelerar(10);  
        coche2.acelerar(20);  
        coche3.parar();  
    }  
}
```

```
class Coche {  
    boolean enMarcha;  
    int velocidad;  
    Rueda r1 = new Rueda();  
    Rueda r2 = new Rueda();  
    Rueda r3 = new Rueda();  
    Rueda r4 = new Rueda();  
  
    void parar() {  
        velocidad = 0;  
        enMarcha = false;  
    }  
  
    void acelerar(int incremento) {  
        velocidad += incremento;  
        enMarcha = true;  
    }  
  
    int dameVelocidad() {  
        return velocidad;  
    }  
}
```

Las clases

Valores devueltos



En el interior del método se usará la palabra reservada **return** para indicar el valor devuelto.

Coche
velocidad:int enMarcha : boolean
acelerar(int) parar() dameVelocidad():int

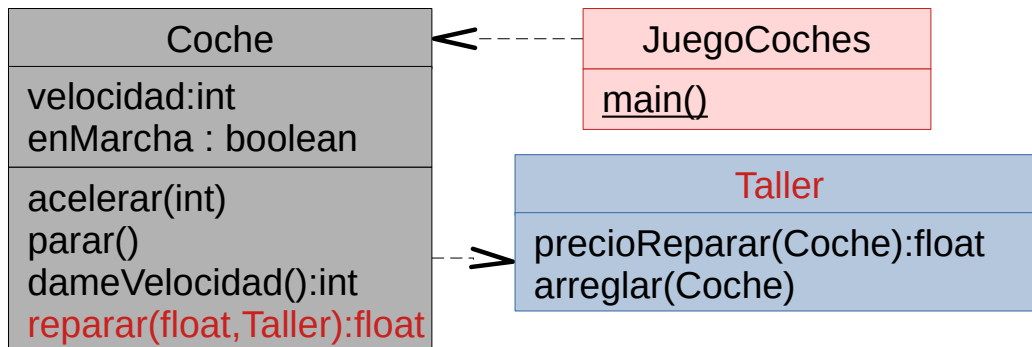
```
public class JuegoCoches {  
    public static void main(String [] args) {  
        Coche coche1 = new Coche();  
        Coche coche2 = new Coche();  
        Coche coche3 = c1;  
        coche1.acelerar(10);  
        coche2.acelerar(20);  
        coche3.parar();  
    }  
}
```

```
class Coche {  
    boolean enMarcha;  
    int velocidad;  
    Rueda r1 = new Rueda();  
    Rueda r2 = new Rueda();  
    Rueda r3 = new Rueda();  
    Rueda r4 = new Rueda();  
  
    void parar() {  
        velocidad = 0;  
        enMarcha = false;  
    }  
  
    void acelerar(int incremento) {  
        velocidad += incremento;  
        enMarcha = true;  
    }  
  
    int dameVelocidad() {  
        return velocidad;  
    }  
}
```

Las clases

Parámetros

A los métodos se les pueden pasar tipos primitivos u otros objetos. Por ejemplo, podemos añadir la clase Taller que cambia las ruedas del coche.



```
public class JuegoCoches {
    public static void main(String [] args) {
        Coche coche1 = new Coche();
        Coche coche2 = new Coche();
        Coche coche3 = c1;
        coche1.acelerar(10);
        coche2.acelerar(20);
        coche3.parar();
    }
}
```

```
class Coche {
    boolean enMarcha;
    int velocidad;
    Rueda r1 = new Rueda();
    Rueda r2 = new Rueda();
    Rueda r3 = new Rueda();
    Rueda r4 = new Rueda();

    void parar() {
        velocidad = 0;
        enMarcha = false;
    }

    void acelerar(int incremento) {
        velocidad += incremento;
        enMarcha = true;
    }

    int dameVelocidad() {
        return velocidad;
    }

    float reparar(float dinero, Taller t) {
        float precio = t.precioReparar(this);
        if (precio < dinero) {
            parar();
            t.arreglar(this);
            return dinero - precio;
        }
        return dinero;
    }
}
```

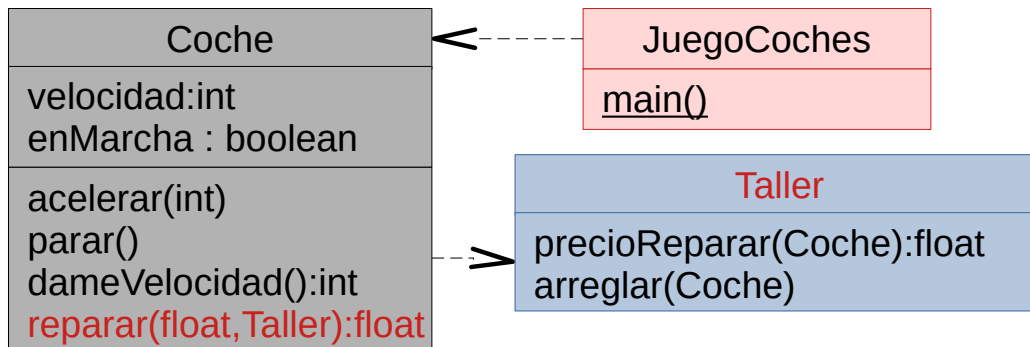
Las clases

Interior de los métodos



La palabra reservada **this** sirve para referirnos al propio objeto.

Por ejemplo, para que el coche pueda pasarse el mismo al taller.



```
public class JuegoCoches {
    public static void main(String [] args) {
        Coche coche1 = new Coche();
        Coche coche2 = new Coche();
        Coche coche3 = c1;
        coche1.acelerar(10);
        coche2.acelerar(20);
        coche3.parar();
    }
}
```

```
class Coche {
    boolean enMarcha;
    int velocidad;
    Rueda r1 = new Rueda();
    Rueda r2 = new Rueda();
    Rueda r3 = new Rueda();
    Rueda r4 = new Rueda();

    void parar() {
        velocidad = 0;
        enMarcha = false;
    }

    void acelerar(int incremento) {
        velocidad += incremento;
        enMarcha = true;
    }

    int dameVelocidad() {
        return velocidad;
    }

    float reparar(float dinero, Taller t) {
        float precio = t.precioReparar(this);
        if (precio < dinero) {
            parar();
            t.arreglar(this);
            return dinero - precio;
        }
        return dinero;
    }
}
```

Por defecto, si no se pone modificador de acceso, el miembro es **friendly**: visible desde las clases que pertenezcan al mismo paquete (directorio) e inaccesible desde fuera del paquete (~ en UML).

Los modificadores de control de acceso más habituales en Java son:

- **public**: **permite** acceder al miembro desde fuera de la clase.
- **private**: **no permite** acceder al miembro desde fuera de la clase.
 - Las propiedades suelen ser siempre privadas.
 - Los métodos privados suelen corresponder a funciones auxiliares.

Modificador	Visibilidad		
	Clase	Paquete	Todos
public	Sí	Sí	Sí
sin modificador	Sí	Sí	No
private	Sí	No	No

Coche
- velocidad : int ~ r1,r2,r3,r4 : Rueda
- incrementarExponencial() + acelerar()

```
class Coche {  
    private int velocidad;  
    Rueda r1 = new Rueda();  
    Rueda r2 = new Rueda();  
    Rueda r3 = new Rueda();  
    Rueda r4 = new Rueda();  
  
    public void acelerar() {  
        incrementarExponencial();  
    }  
  
    private void incrementarExponencial() {  
        for (int cont = 0; cont < 100; cont++)  
            velocidad += cont;  
    }  
}
```

Una **clase** con modificador **public** indica que la clase es **visible desde otros paquetes** diferentes al paquete en el que se implementa.

Por el contrario una clase no definida publica **solo es visible en el paquete** en el que se implementa.

Solo puede existir una clase pública por fichero y el nombre de tal clase debe coincidir con el nombre del fichero.

```
class Rueda {  
    private int radio = 3;  
    ...  
}  
  
public class Coche {  
    private Rueda r = new Rueda();  
    ...  
}
```

Coche.java 

Las clases

Clases internas y clases internas privadas

Una **clase interna** es una clase que se define anidada dentro de otra.

```
public class Coche {  
    class Rueda {  
        private int radio = 3;  
        ...  
    }  
  
    private Rueda r = new Rueda();  
    ...  
}
```

Una **clase interna** definida **private** solo es visible dentro de la clase en la que está definida.

```
public class Stack {  
    private class Node {  
        Node next;  
        int value;  
    }  
  
    private Node top = new Node();  
    ...  
}
```

Las clases

Todo junto en un ejemplo

```
public class Coche {
    class Rueda {
        private int presion = 2000;
        private float radio = 6.28f;

        public float dameRadio() {
            return radio;
        }
        public void subePresion() {
            presion += 10;
        }
    }

    private boolean enMarcha;
    private int velocidad;
    private Rueda ruedaRepuesto = new Rueda();

    /** Fren el coche en seco */
    public void parar() {
        velocidad = 0;
        enMarcha = false;
    }
}
```

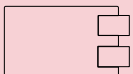
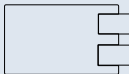
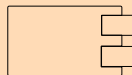
```
public class Taller {
    /** @param c Coche a ser reparado.
     * @return presupuesto de la reparación */
    public float precioReparar(Coche c) {
        return 100 * c.dameRueda().dameRadio();
    }
    /** Repara el coche.
     * @param c Coche a ser reparado.*/
    public void arreglar(Coche c) {
        c.dameRueda().subePresion();
    }
}
```

```
/** @return devuelve la rueda de repuesto */
public Rueda dameRueda() {
    return ruedaRepuesto;
}

/** Método que aumenta la velocidad
 * @param incremento Valor que se suma a velocidad */
public void acelerar(int incremento) {
    velocidad += incremento;
    enMarcha = true;
}

/** Lleva el coche al taller
 * @param dinero disponible para la reparación */
public float reparar(float dinero, Taller t) {
    float precio = t.precioReparar(this);
    if (precio < dinero) {
        parar();
        t.arreglar(this);
    }
    return dinero - precio;
}
}
```

```
public class JugandoConCoche {
    public static void main(String[] args) {
        Coche c = new Coche();
        Taller t = new Taller();
        c.reparar(1000f, t);
    }
}
```



Detalles de las propiedades

En esta sección se profundiza en el concepto de propiedad.

Detalles de las propiedades

Propiedades privadas



Normalmente las propiedades de los objetos se **mantienen privadas**.

Si se desea que una propiedad privada sea consultada o modificada desde fuera de la clase se suelen añadir métodos públicos **getters y setters**.

Los getters y setters permiten consultar o modificar variables privadas de forma controlada, **evitando romper el invariante** de una clase.

```
class Coche {  
  
    private int speed;  
  
    public int getSpeed() {  
        return speed;  
    }  
    public void setSpeed(int speed) {  
        if (speed > 120)  
            this.speed = 120;  
        else if (speed < 0)  
            this.speed = 0;  
        else  
            this.speed = speed;  
    }  
}
```

Detalles de las propiedades

Propiedades final



final es un modificador que se usa para indicar que el valor de una propiedad no va a cambiar. Así, las propiedades final son **constantes** a lo largo de la vida del objeto. Su valor puede definirse en compilación o en ejecución.

Las referencias a objetos declaradas final no pueden cambiarse, aunque los objetos en sí mismos pueden cambiar su estado.

Una variable final de tipo primitivo, por su carácter inmutable, pueden decidirse declararla pública.

```
class Coche {  
  
    private boolean enMarcha;  
    public final int numRuedas = 4;  
    private final Rueda r = new Rueda();  
  
    ...  
}
```

Coche

- numRuedas : int = 4 {readOnly}
- + enMarcha : boolean
- rueda : Rueda {readOnly}

Detalles de las propiedades

Propiedades de objeto o de clase

Al definir propiedades se pueden distinguir dos tipos de propiedades:

- **propiedades de objeto**, que se pueden consultar en los objetos que se creen de esa clase.
- **propiedades de clase** (o **estáticas**), que se pueden consultar sobre la propia clase. Las **propiedades estática son compartidas** por todos los objetos de la clase (comparten valor). Para indicar que una propiedad es estática se usa el modificador de uso **static** (en UML se subrayan).

final puede usarse combinado con static para definir propiedades constantes para todos los objetos de una clase.

Coche
- enMarcha : boolean + numRuedas : int = 4 {ReadOnly} + <u>pi : double = 3.14 {ReadOnly}</u>

```
class Coche {  
    //Propiedades de los objetos  
    private boolean enMarcha;  
    public final int numRuedas = 4;  
  
    //Propiedad de la clase  
    public final static double pi = 3.14;  
}
```

Detalles de las propiedades

Acceso a las propiedades de clase

Para acceder a las propiedades de clase **desde fuera** de la clase se utiliza el nombre de la clase, o de una referencia a un objeto, seguido del nombre de la propiedad separados por un punto.

```
class Coche {  
  
    private boolean enMarcha;  
    public final int numRuedas = 4;  
    private Rueda r = new Rueda();  
  
    public final static double pi = 3.14;  
  
    void acelerar(int incremento) {  
        velocidad += incremento;  
        enMarcha = true;  
    }  
}  
...  
Coche coche = new Coche();  
  
System.out.println(c.numRuedas);  
System.out.println(Coche.pi);  
System.out.println(c.pi);
```

Detalles de las propiedades

Otros modificadores de uso en propiedades



Las propiedades **transient** **no persisten** aunque el objeto tenga la capacidad de persistencia.

Una propiedad **volatile** no se copia a la cache del procesador, sino que se mantiene en memoria principal. Esto evita problemas cuando la variable pueda ser utilizada desde varios threads de manera simultánea.

Detalles de los métodos

En esta sección se profundiza en el concepto de método.

Detalles de los métodos

Paso de parámetros en Java



En los métodos de Java el paso de parámetros se realiza **siempre por valor**.

El paso de parámetros por valor consiste en **copiar el contenido** de la variable que se pasa en otra **variable local al método**.

Los parámetros que se le pasan a un método son:

- Tipos primitivos, que se pasan por valor.
- Referencias a objetos, que también se pasan por valor. Obsérvese en este caso que, al ser referencias, el estado del objeto pasado puede ser modificable.

Detalles de los métodos

Métodos de objeto o de clase

En Java se distinguen dos tipos de métodos:

- métodos **de objeto** (o de instancia), que se pueden invocar sobre los objetos que se creen de esa clase.
- métodos **de clase** (también llamados **estáticos**), que se pueden invocar sobre la propia clase. En UML, los miembros estáticos se distinguen porque el nombre del método está subrayado.

Coche
- velocidad : int - <u>cochesCreados</u> : int
+ <u>sumarCoche</u> () + acelerar + parar()

Detalles de los métodos

Detalles del modificador static

En Java, para indicar que un método es de clase (o estático) se debe declarar usando la palabra reservada **static**.

Dentro de un método estático no se pueden usar las propiedades de instancia de la propia clase, porque no hay instancia, aunque sí se pueden usar propiedades estáticas.

Coche
- velocidad : int - <u>cochesCreados</u> : int
+ <u>sumarCoche()</u> + acelerar + parar()

```
class Coche {  
    private int velocidad;  
    private static int cochesCreados;  
  
    // Método de clase  
    public static void sumarCoche() {  
        cochesCreados++;  
    }  
  
    //Métodos del objeto  
    public void acelerar() {  
        velocidad++;  
    }  
  
    public void parar() {  
        velocidad = 0;  
    }  
}
```

Detalles de los métodos

Acceso a los métodos de clase

Para acceder a un método de clase **desde dentro** de la clase solo hace falta utilizar el identificador del método seguido de los paréntesis y los parámetros en su caso.

Para acceder a un método de clase **desde fuera** de la clase se utiliza el identificador de la clase seguido del identificador del método, los paréntesis y los parámetros.

```
class Coche {  
    private int velocidad;  
    private static int cochesCreados;  
  
    // Método de clase  
    public static void sumarCoche() {  
        cochesCreados++;  
    }  
  
    //Métodos del objeto  
    public void acelerar() {  
        velocidad++;  
    }  
  
    public void parar() {  
        velocidad = 0;  
    }  
}  
  
metodoExterno() {  
    Coche.sumarCoche();  
}
```

Detalles de los métodos

Detalles del modificador static

Todo programa en Java comienza su ejecución en un método static denominado **main**.

```
/** Programa en Java que muestra un mensaje de bienvenida */  
class HolaMundo {  
    // El método en el que comienza la ejecución del programa es main  
    public static void main (String [ ] arg)    {  
        System.out.println("Hola mundo");  
    }  
}
```

Detalles de los métodos

Sobrecarga estática de los métodos

En una clase de Java es posible tener **dos métodos con el mismo identificador**, aunque los parámetros o el tipo devuelto deben ser diferentes.

Esto se conoce como **sobrecargar** el método.

```
class Coche {  
    private int velocidad;  
  
    public void acelerar(int v) {  
        velocidad += v;  
    }  
  
    public void acelerar(boolean marcha) {  
        velocidad += 60;  
    }  
}
```

PrintStream

- + println()
- + println(boolean)
- + println(char)
- + println(char[])
- + println(double)
- + println(float)
- + println(int)
- + println(long)
- + println(Object)
- + println(String)

Detalles de los métodos

El constructor



La palabra reservada **new** seguida del nombre de una clase y unos paréntesis con o sin parámetros invoca al **constructor de la clase**. El constructor es una función especial que se ejecuta cuando se crea un objeto y se usa para **iniciar sus propiedades**.

Un constructor puede o no tener **parámetros**, pero **no tiene tipo de retorno**. Siempre tienen el **nombre** de la clase a la que pertenecen.

Si no se define un constructor, Java añade un **constructor por defecto** sin parámetros. Este constructor no aparece en el código y solo crea el objeto.

```
class Coche {  
  
    private boolean enMarcha;  
    public final int numRuedas = 4;  
    public final static pi = 3.14;  
    private Rueda rueda1 = new Rueda();  
    private int velocidad;  
  
    ...  
    public void acelerar(int incremento) {  
        velocidad += incremento;  
        enMarcha = true;  
    }  
}  
...  
  
metodoExterno() {  
    Coche c = new Coche();  
    System.out.println(c.numRuedas);  
    System.out.println(Coche.pi);  
}
```


Detalles de los métodos

Varios constructores

Una clase **puede tener varios constructores**, aunque no puede tener dos constructores que reciban los mismos parámetros.

En un constructor se pueden **iniciar incluso propiedades** finales.

Desde un constructor se puede llamar a otros constructores de la misma clase usando la palabra reservada **this** seguida de paréntesis con los parámetros correspondientes.

Las **llamadas entre constructores** son habituales cuando se desea añadir constructores con valores fijos para algunos parámetros.

```
class Coche {  
  
    private boolean enMarcha;  
    public final int numRuedas;  
    public final static double pi = 3.14;  
    private int velocidad;  
    ...  
    public Coche(int velInicial, boolean on) {  
        velocidad = velInicial;  
        NumRuedas = 4;  
        this(on);  
    }  
    public Coche(boolean on) {  
        enMarcha = on;  
    }  
    public Coche() {  
    }  
}  
  
metodoExterno() {  
    Coche c1 = new Coche();  
    Coche c2 = new Coche(12, true);  
    Coche c3 = new Coche(true);  
}
```

Detalles de los métodos

Destrucción de objetos y finalizadores

La máquina virtual de Java revisa de periódicamente los bloques de memoria reservados buscando los que no están referenciados por ninguna variable para liberarlos.

La tarea que realiza esta operación se llama **recolector de basura** (**garbage collector**) y se ejecuta en segundo plano aprovechando los tiempos de baja intensidad de proceso.

Los **finalizadores** son métodos que se ejecutan **antes de la liberación** de la memoria ocupada por los objetos. Permiten tareas como cerrar ficheros, avisar a otros objetos... El finalizador será el método llamado **void finalize()**. Desde Java 9 **se desaconseja su uso** (**deprecated**) por la no determinidad de su momento de ejecución.

```
class Coche {  
    private Parking parking;  
  
    public Coche(Parking p) {  
        parking = p;  
    }  
  
    public void finalize() {  
        parking.removeCar(this);  
    }  
}  
...  
metodoExterno() {  
    Parking p = new Parking();  
    Coche c = new Coche();  
    p.add(c);  
}
```

Detalles de los métodos

Otros modificadores de uso en los métodos



Los métodos **native** se implementan en un **lenguaje nativo** de la máquina (C, C++...) y se asocian a Java usando bibliotecas de enlace dinámico mediante la Java Native Interface (JNI).

Los métodos **synchronized** solo pueden ser ejecutados **por un hilo** en cada momento para cada objeto.

Un método **strictfp** fuerza a Java a utilizar una **aritmética flotante independiente del procesador** para asegurar compatibilidad multiplataforma.

Los arrays

Presentación del sistema de arrays de Java.

Los arrays

Declaración de arrays



Java proporciona una **clase array** como contenedor básico de objetos.

Para la declaración de una referencia a un objeto array se utiliza el tipo de objetos o tipo primitivo que contendrá el array seguido de una pareja de corchetes vacía.

```
<tipo> '[]' <identificador>;
```

```
int [] vector;
```

```
Coche [] parking;
```

Los arrays

Inicialización de arrays



Como siempre, si una referencia a un array **no se inicializa** su valor es **null**.

Para crear un array, se utiliza la palabra reservada **new** seguida de una pareja de corchetes con las dimensiones del array.

```
<tipo> '[' <identificador> = new <tipo> [<dimensión>];
```

Inicialmente, cada posición del array se rellena con el valor por defecto del tipo de datos asignado al array.

Una vez dimensionado un array **no es posible redimensionarlo**.

```
int [] vector = new int[25]; //Crea un array de tamaño 25 inicialmente con valores 0
```

```
int N = math.rand();  
Coche [] parking = new Coche[N]; //Crea un array de tamaño N inicialmente con valores null
```

Los arrays

Declaración implícita

Java también permite declarar **implícitamente** la dimensión un array inicializándolo con los elementos que contiene y sin especificar su dimensión.

```
<tipo> '[]' <identificador> = {[objetos|valores primitivos|cadenas de caracteres]*};
```

```
int [] fib = {1,1,2,3,5,8,13};
```

```
char [] cadena = {'j','u','a','n'};    //Array de char
```

Los arrays

Acceso a los elementos de un array



El acceso a un elemento de un array se realiza utilizando la **variable y entre corchetes** el índice del elemento al que se desea acceder.

Los índices **comienzan por cero** y alcanzan como máximo un valor igual a la dimensión del array menos uno.

```
int [] vector = new int[25];

for (int x = 0; x < 25; x++)
    vector[x] = 3;

int resultado = 0;
for (int x = 0; x < 25; x++) {
    resultado += vector[x];
}
```

Java no permite sobrecargar símbolos para ejecutar métodos, pero en el caso de los array se ha hecho una excepción con los corchetes.

Los arrays

Arrays de arrays de...

Java también permite definir un **array compuesto de otros arrays** de tamaños variables (y así sucesivamente).

Esta posibilidad se potencia con el uso de la propiedad **length** que tiene la clase array y que devuelve la dimensión del mismo.

```
<tipo> '[' <identificador> = new <tipo> [<dimensión>];  
<tipo> '[][]' <identificador> = new <tipo> [<dimensión1>] [<dimensión2>];  
<tipo> '[' <identificador> = new <tipo> [<dimensión1>] ... [<dimensiónN>];
```

```
Coche [][] tablaCoches = new Coche[25][30];
```

```
for (int x = 0; x < 25; x++)  
for (int y = 0; y < 30; y++)  
    tablaCoches[x][y] = new Coche();
```

```
int [][] vectores = new int[2][];  
vectores[0] = new int[5];  
vectores[1] = new int[8];  
  
for (int x = 0; x < vectores.length; x++)  
for (int y = 0; y < vectores[x].length; y++)  
    vectores[x][y] = -1;
```

Los arrays

Métodos para el tratamiento de arrays

La clase `java.util.Arrays` incluye varios métodos para realizar diferentes operaciones sobre arrays.

```
java.util.Arrays
+ int binarySearch(char[], int, int, char)
+ int binarySearch(double[], double)
+ char[] copyOf(char[] original, int)
+ double[] copyOf(double[] original, int)
+ char[] copyOfRange(char[] original, int, int)
+ double[] copyOfRange(double[] original, int, int)
+ boolean equals(boolean[], boolean[])
+ boolean equals(char[], char[])
+ boolean equals(double[], double[])
+ void fill(char[], char val)
+ void fill(double[], double val)
+ int hashCode(char[])
+ int hashCode(double[])
+ void sort(char[])
+ void sort(double[])
+ String toString(char[])
+ String toString(double[])
...
```

Los paquetes

Presentación de la forma de crear y acceder a los paquetes en Java.

Las clases en Java se organizan en ficheros y los ficheros en Paquetes. Java define una **correspondencia** entre los **directorios**, que contienen físicamente a los ficheros, y los **paquetes**, que los contienen lógicamente.

Java permite **organizar en forma de árbol los paquetes** gracias al uso de los directorios de ficheros.



Los paquetes

Declaración de paquetes



La pertenencia de un fichero a un paquete se declara en la primera línea de código del fichero usando la palabra reservada **package** seguida del nombre del paquete.

Los nombres de los paquetes se suelen escribir en **minúsculas**, los nombres de las clases usando **UpperCamelCase** y lo de los miembros usando **lowerCase**.

El anidamiento de paquetes en Java se representa usando el punto como separador.

Abajo, la clase Parking pertenece al paquete `urjc.util.automovilismo`.

```
package urjc.util.automovilismo;

public class Parking {
    Coche c = new Coche();
    Camion c1 = new Camion();
    vehiculos.agricolas.Tractor = new vehiculos.agricolas.Tractor();
}
```



Un paquete debe residir en un directorio con su mismo nombre localizable a partir de alguno de los directorios declarados en la variable de entorno **CLASSPATH** o en el parámetro `classpath` en la invocación a la máquina virtual Java.

En el ejemplo, el fichero `Parking.java` estará en el directorio `automovilismo`, que estará dentro del directorio `util`, que estará dentro del directorio `urjc`, que será accesible desde el **CLASSPATH**.

```
package urjc.util.automovilismo;

public class Parking {
    Coche c = new Coche();
    Camion c1 = new Camion();
    vehiculos.agricolas.Tractor = new vehiculos.agricolas.Tractor();
}
```

Los paquetes

Acceso a un paquete desde otro



Para acceder a un tipo definido en un fichero se puede usar su **nombre completamente calificado**, que consiste en separar con puntos los nombres ordenados de toda la rama de paquetes hasta el tipo.

Se puede utilizar la palabra reservada **import** al principio de un fichero para evitar usar nombre completamente calificados para acceder a un tipo.

```
package urjc.util.automovilismo;

import vehiculos.utilitarios.Coche;
import vehiculos.camiones.*;

public class Parking {
    Coche c = new Coche();
    Camion c1 = new Camion();
    vehiculos.agricolas.Tractor t = new vehiculos.agricolas.Tractor();
}
```

Los paquetes

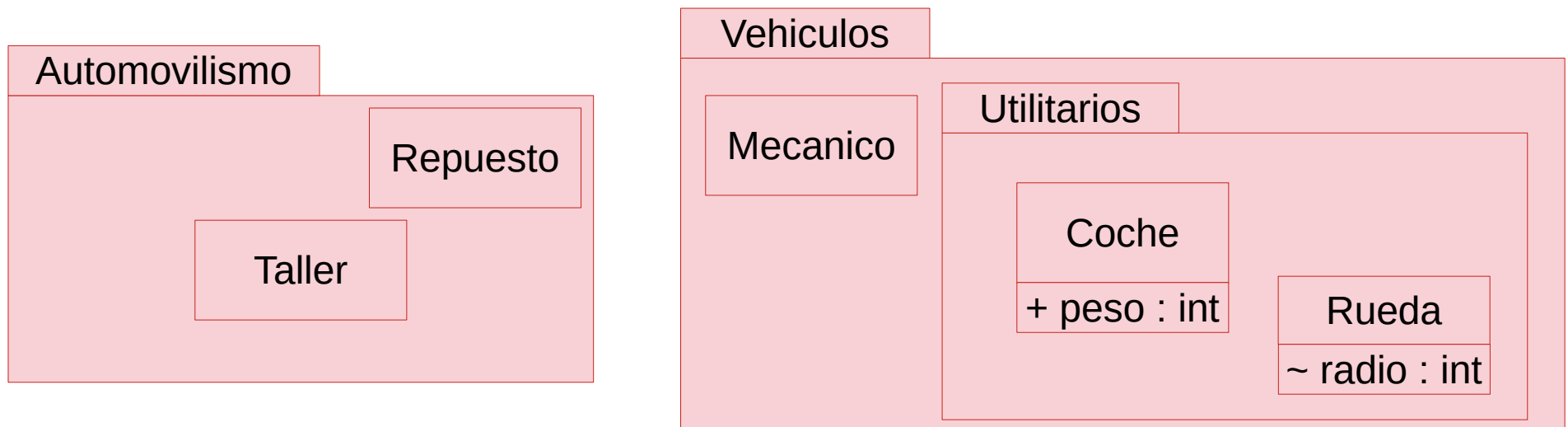
Visibilidad de los elementos de un paquete

Dentro de un paquete se comparten los elementos **friendly**.

No se puede acceder a los elementos friendly de otro paquete aunque exista relación de anidamiento.

En el ejemplo Coche ve la propiedad friendly radio de la rueda, pero Taller no la ve porque no está en el mismo paquete.

El anidamiento de paquetes no tiene consecuencias en la visibilidad.



Referencias

- El lenguaje de programación Java, 3ª Edición, Arnold Gosling, Addison Wesley, 2001.
- Piensa en Java, Eckel, 2ª Edición, Addison Wesley, 2002.
- Introducción a la programación orientada a objetos con Java, C. Thomas Wu, Mc Graw Hill, 2001.
- Deep Dive Into the New Java JIT Compiler – Graal, Baeldung ([web](#) consultada en octubre de 2020).
- Diseñar y programar todo es empezar, Vélez y otros, Dykinson 2008 ([web](#)).