



JAVA 8

¿QUÉ ES JAVA?

- ▶ Un lenguaje de programación para desarrollar aplicaciones.
- ▶ Un conjunto (muy grande) de código disponible para su uso.
- ▶ Una plataforma donde ejecutar aplicaciones.

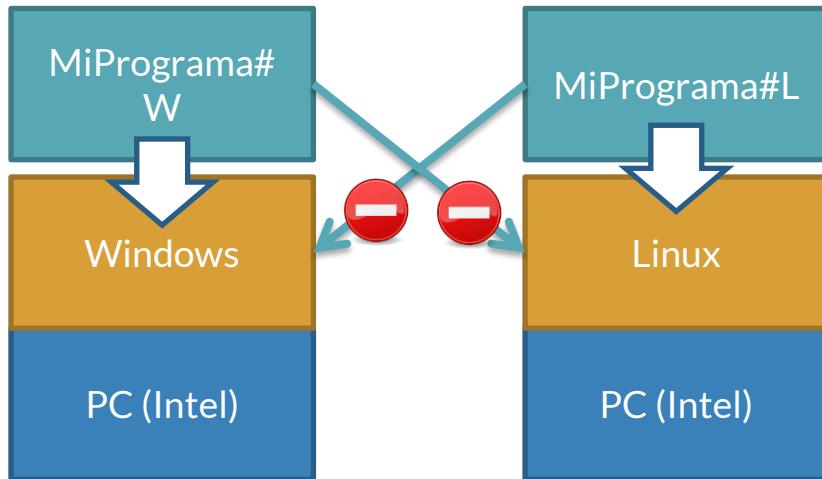
INICIOS DE JAVA

- ▶ La POO surgen en los 80 (Smalltalk) y toma auge en los 90 (C++).
- ▶ C++ no es nativamente OO, sino una extensión OO para C.
- ▶ Problemas para desarrollar software multiplataforma.

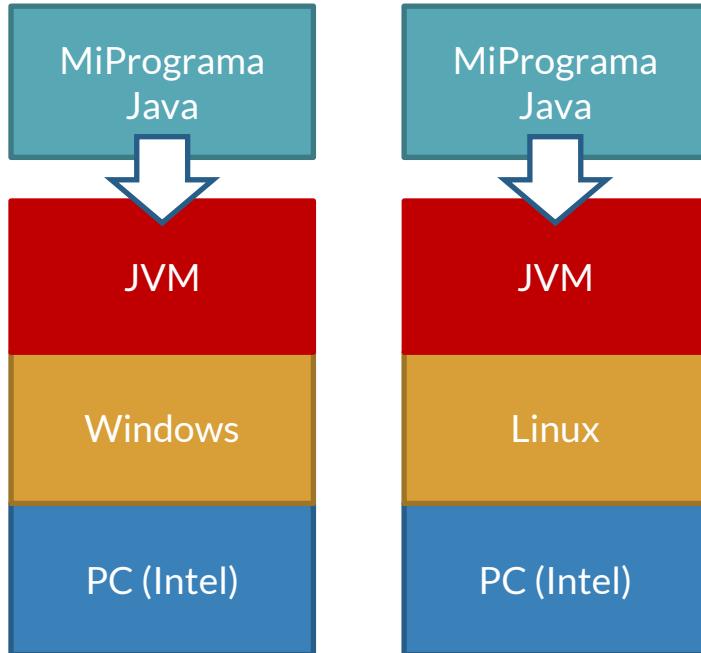
INICIOS DE JAVA

- ▶ Sun Microsystems necesita una plataforma para diferentes tipos de dispositivos.
- ▶ A mediados de los 90s surge Java, contemporáneo al auge de la WWW.
- ▶ Se plantea para que se ejecute en dispositivos muy diferentes.

JAVA VIRTUAL MACHINE

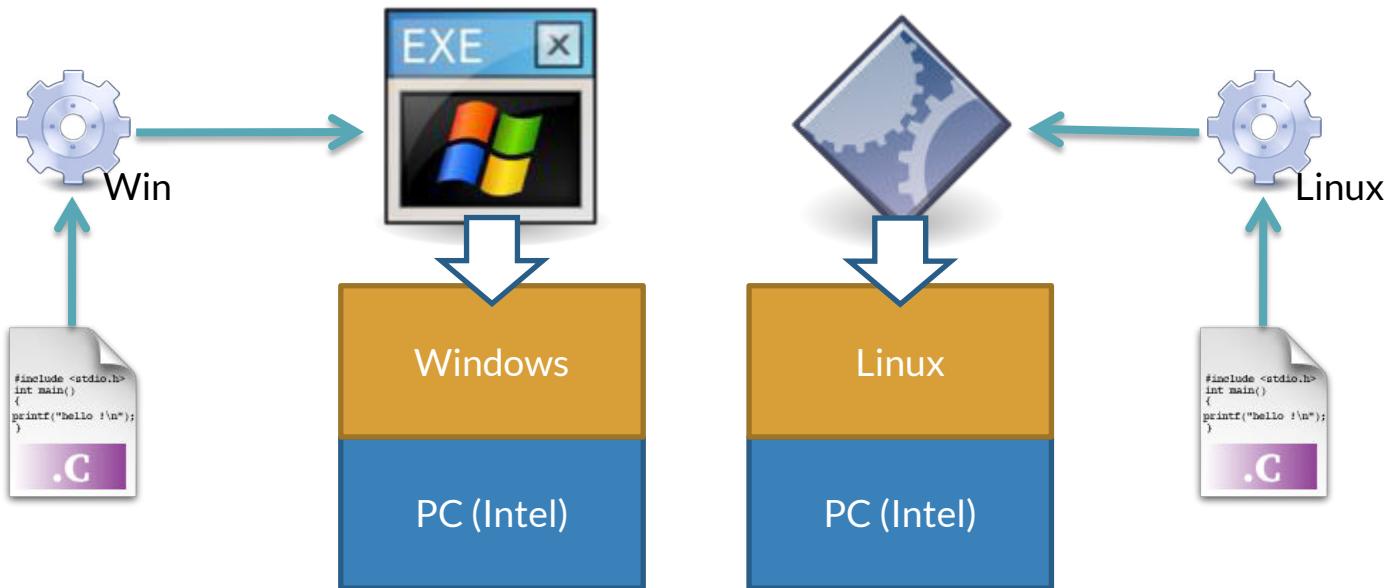


JAVA VIRTUAL MACHINE

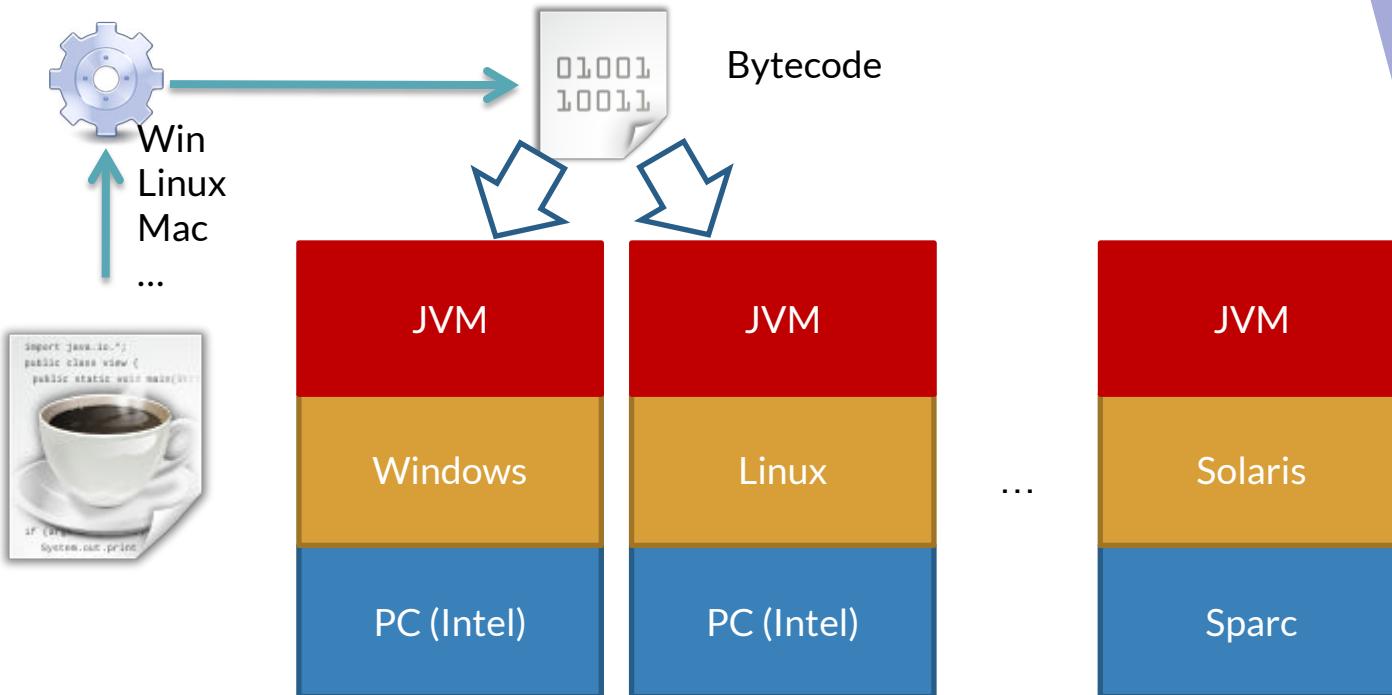


La máquina virtual de Java es un entorno virtual de ejecución de aplicaciones, que está disponible para múltiples arquitecturas, dispositivos y sistemas operativos

JAVA BYTECODE



JAVA BYTECODE



JAVA 3.000 millones de dispositivos



Sistemas de vuelo



Cajeros automáticos



Reproductores de BlueRay



Libros electrónicos
Kindle



Routers



Robótica

...

JRE vs. JDK



Java Runtime Environment (JRE)

Se trata de un conjunto de ficheros y programas que nos permiten ejecutar aplicaciones Java.

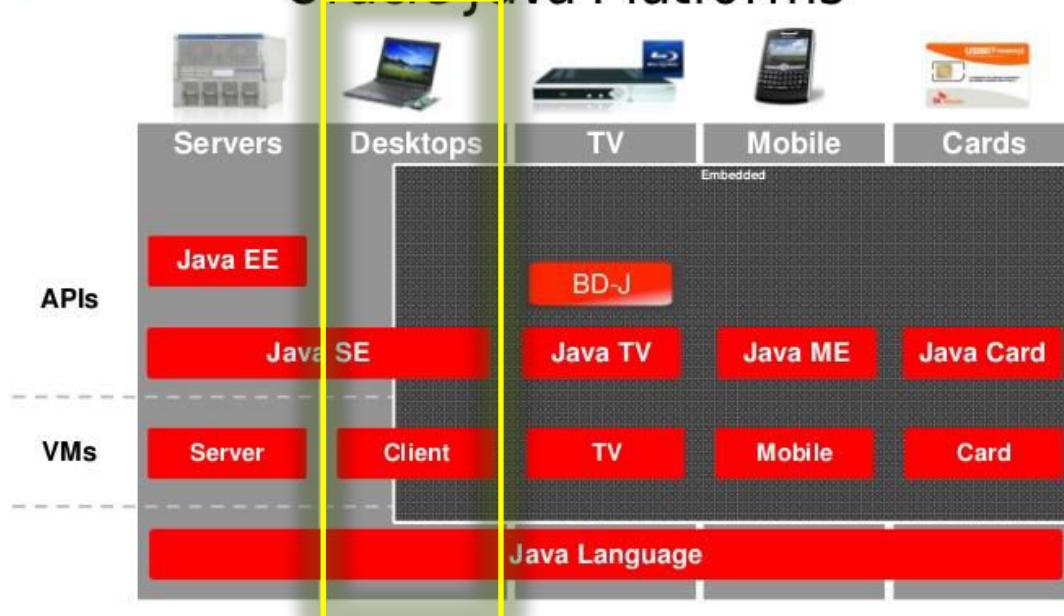


Java Development Kit (JDK)

Se trata de un conjunto de ficheros y programas que nos permiten crear, compilar y ejecutar aplicaciones Java.

JAVA es mucho JAVA

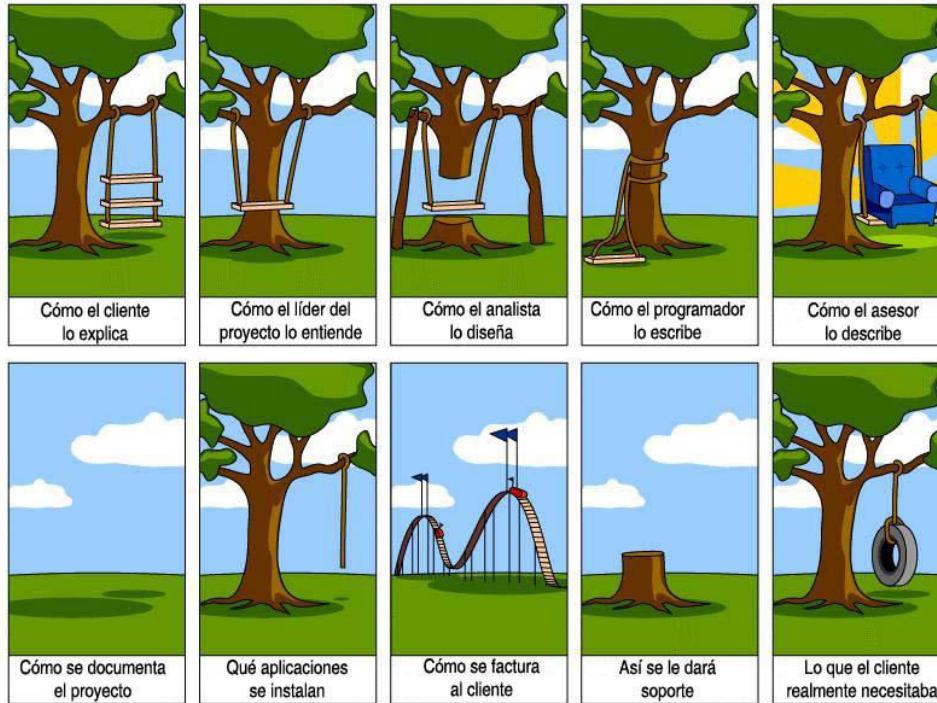
- Oracle Java Platforms



ENTORNOS DE DESARROLLO



DESARROLLO DE SOFTWARE



DESARROLLO DE SOFTWARE

- ▶ Tareas de análisis (¿qué problema vamos a resolver?)
- ▶ Tareas de diseño (¿cómo resolverlo?)
- ▶ Tareas de codificación (**implementación**, documentación, pruebas, ...)
- ▶ Tareas de mantenimiento (correctivo, evolutivo, ...)

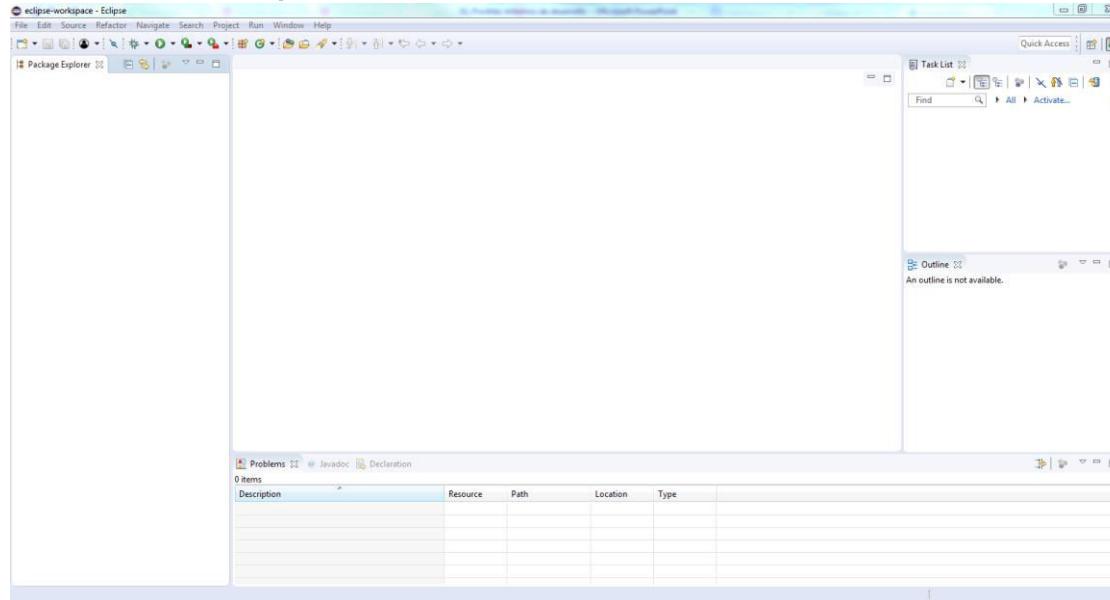
TAREAS A LA HORA DE CODIFICAR SOFTWARE

- ▶ *Picar el código*
- ▶ Depurar
- ▶ Organizar el proyecto en agrupaciones de código.
- ▶ Testear
- ▶ ...



ENTORNOS INTEGRADOS DE DESARROLLO DE SOFTWARE

- ▶ Herramientas *visuales* que nos van a ayudar en la mayoría de las tareas.



IDEs PARA JAVA



NetBeans



University of
Kent



DEAKIN UNIVERSITY



ALGUNOS ELEMENTOS DE SINTAXIS



ESTRUCTURA DE UN PROGRAMA BÁSICO

```
Comentario    → /**
 * Nuestro primer programa en Java
 */

Paquete       → package holamundo;

Comentario    → /**
 * @author Openwebinars
 *
 */

Clase          → public class HolaMundo {

Comentario    → /**
 * @param args
 */

Método         → public static void main(String[] args) {
                  // TODO Auto-generated method stub
                  System.out.println("Hola Mundo");
                }

Sentencia o expresión → }
```

DEFINICIÓN DE UNA CLASE

- ▶ Mínimo 1 por fichero
- ▶ El nombre del fichero = nombre de la clase.
- ▶ Mínimo 1 método main.

```
public class HolaMundo {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("Hola Mundo");  
    }  
}
```

SENTENCIA o LÍNEA DE CÓDIGO

- ▶ Indican al programa que debe hacer.
- ▶ Ojo a las mayúsculas y minúsculas.
- ▶ Ojo a las tabulaciones.
- ▶ Siempre terminan en punto y coma.

```
public class HolaMundo {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("Hola Mundo");  
    }  
}
```

BLOQUE DE CÓDIGO

- ▶ Se trata de una agrupación de sentencias.
- ▶ Delimitado siempre por { }
- ▶ Clases, métodos, bloques, ...

```
public class HolaMundo {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("Hola Mundo");  
    }  
}
```

COMENTARIOS

- ▶ Documentación
- ▶ Aclaraciones
- ▶ Tareas pendientes
(Window → Show view → Task)

```
/**  
 * Nuestro primer programa en Java  
 */  
package holamundo;  
  
/**  
 * @author Openwebinars  
 *  
 */  
public class HolaMundo {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("Hola Mundo");  
    }  
}
```

VARIABLES y TIPOS DE DATOS PRIMITIVOS



DATOS

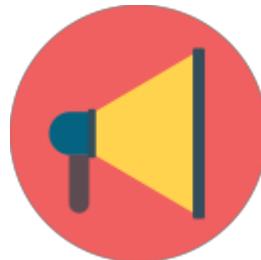
- ▶ Registro de hechos, objetos, ideas...
- ▶ Se representan mediante símbolos.



Imagen



Número



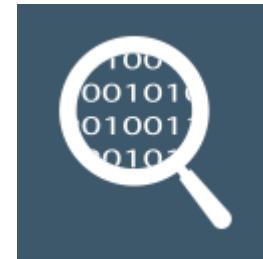
Sonido



Caracteres

VARIABLE

- ▶ Es la forma de almacenar un dato en un programa.
- ▶ Deben guardarse en la memoria del ordenador.
- ▶ A nivel físico, estos solo entienden en lenguaje de 0 y 1.



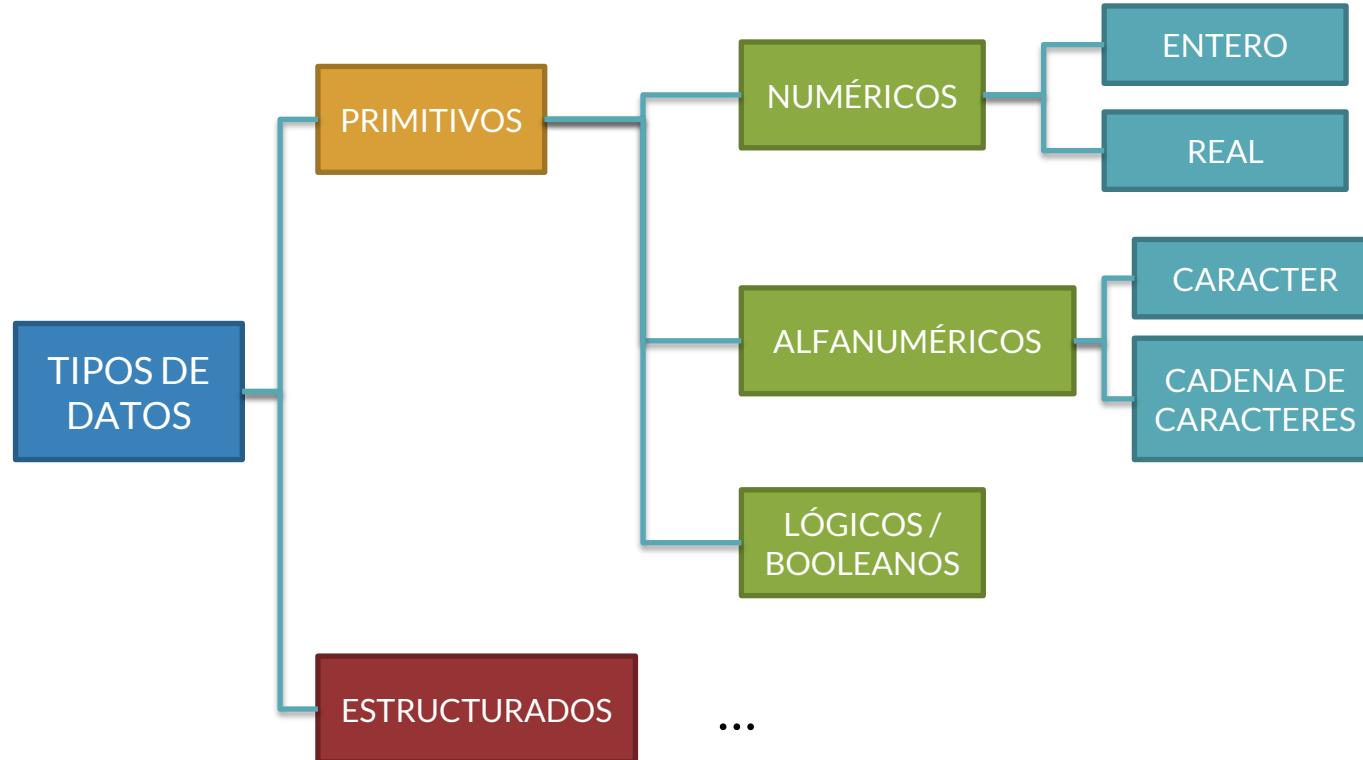
VARIABLE

- ▶ CONCLUSIÓN: Es una zona delimitada de la memoria del ordenador, a la que se le asigna un nombre y un tipo (de dato).

0	1	0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0	1	0
0	1	0	1	1	0	0	1	1	1
0	1	0	1	0	1	1	0	0	0
0	1	0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0	1	0

Tipo: número Nombre: precio Valor decimal: 45996

TIPOS DE DATOS PRIMITIVOS



DECLARACIÓN e INICIALIZACIÓN

- ▶ Como norma general, hay que declarar e inicializar una variable antes de usarla.

```
public static void main(String[] args) {  
  
    int numero = 7;  
  
    char letra = 'a';  
  
    System.out.println(numero);  
  
    System.out.println(letra);  
  
}
```

CÓMO NOMBRAR VARIABLES

- ▶ Nombre *autodescriptivo*
- ▶ *Case-sensitive*
- ▶ No espacios en blanco.
- ▶ Deben comenzar por una letra (también por \$ o _, pero no está recomendado)
- ▶ No pueden ser una palabra reservada del lenguaje (por ejemplo, *class*).
- ▶ Notación *camelCase*.
- ▶ Los valores constantes suelen ir en mayúsculas, y separan las palabras con _.

ÁMBITO DE UNA VARIABLE

- ▶ Período de tiempo que el programa almacena dicho valor en memoria.
- ▶ Como norma general, el ámbito de una variable es el bloque donde ha sido definida.

```
public static void main(String[] args) {  
  
    int numero = 7;  
  
    char letra = 'a';  
  
    System.out.println(numero);  
  
    System.out.println(letra);  
  
}
```

MANIPULACIÓN DE NÚMEROS, CARACTERES Y OTROS VALORES



TIPOS DE DATOS PRIMITIVOS EN JAVA

Lógicos	boolean	true, false
Caracteres	char	caracteres unicode
Numéricos	byte	entero de 8 bits
	short	entero de 16 bits
	int	entero de 32 bits
	long	entero de 64 bits
	float	real de 32 bits
	double	real de 64 bits

Adicionalmente, tenemos la cadena de caracteres `java.lang.String`

VALORES LITERALES

- ▶ Representación de un valor fijo.
- ▶ Se escriben directamente en el código.

```
//Valor booleano VERDADERO  
boolean resultado = true;  
//Letra C  
char letraMayuscula = 'C';  
//Número 100  
byte by = 100;  
//Número 1000  
short sh = 1000;  
//Número 1000000  
int in = 1000000;
```

```
//Valor 26, en decimal  
int decVal = 26;  
//Valor 26, en hexadecimal  
int hexVal = 0x1a;  
//Valor 26, en binario  
int binVal = 0b11010;|
```

VALORES LITERALES

- ▶ Desde Java SE 7, se puede usar el guión bajo (_) para delimitar parte de un literal. Java no lo procesará como parte de ese literal.

```
long creditCardNumber = 1234_5678_9012_3456L;
```

OPERADOR DE ASIGNACIÓN

- ▶ =
- ▶ Nos permite asignar un valor a una variable.
- ▶ El valor asignado debe ser del tipo sobre el que se ha definido la variable.

```
//Valor booleano VERDADERO  
boolean resultado = true;  
//Letra C  
char letraMayuscula = 'C';  
//Número 100  
byte by = 100;  
//Número 1000  
short sh = 1000;  
//Número 1000000  
int in = 1000000;
```

1.

TIPOS DE DATOS NUMÉRICOS

NÚMEROS ENTEROS

- ▶ Números sin decimales, positivos y negativos.

Tipo de dato	Tamaño	Rango	Valor por defecto
byte	8 bits (1 byte)	De -128 a 127	0
short	16 bits	De -32,768 a 32,767	0
int	32 bits	De -2^{31} a $2^{31}-1$	0
long	64 bits	De -2^{63} a $2^{63}-1$	0L

NÚMEROS REALES

- ▶ Números con decimales, positivos y negativos.

Tipo de dato	Tamaño	Valor por defecto
float	32 bits (IEEE 754)	0.0f
double	64 bits (IEEE 754)	0.0d

IEEE 754

Núm = **mantisa** * **base**^{exponente}

$$345 = 0.345 * 10^3$$

OPERADORES NUMÉRICOS

Tipo	Operador	Precedencia	Operación realizada
Prefix, postfix	-- ++	expr++ expr-- ++expr --expr	Incremento/Decremento en una unidad.
Unarios	+ -	+ -	Cambio de signo
Multiplicativos	* / %	* / %	Multiplicación, división y resto
Aditivos	+ -	+ -	Suma, resta
De movimiento	<< >> >>>	<< >> >>>	Desplazamiento a nivel de bits.

OPERADORES NUMÉRICOS DE ASIGNACIÓN ABREVIADA

Operador	Uso	Equivalente a
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>

OPERADORES NUMÉRICOS A NIVEL DE BITS

Operador	Descripción
&	Realiza la operación lógica AND a nivel de bits
^	Realiza la operación lógica OR exclusivo a nivel de bits
	Realizar la operación lógica OR inclusivo a nivel de bits
~	Cambia cada 0 por un 1, y cada 1 por un cero
<<	Desplaza un número de bits hacia la izquierda. Rellena los huecos con ceros.
>>	(Signed). Desplaza un número de bits hacia la derecha. Rellena los huecos con el bit más significativo (el que indica el signo).
>>>	(Unsigned). Desplaza un número de bits hacia la derecha. Rellena con ceros a la izquierda

2.

TIPOS DE DATOS LÓGICOS

VALORES BOOLEANOS

- ▶ Susceptibles de ser VERDADEROS (true) o FALSOS (false).
- ▶ Muy útiles en comparaciones y otras operaciones.

OPERADORES LÓGICOS CONDICIONALES

Operador	Descripción
!	Realiza la negación del operando
&&	Realiza la operación lógica condicional AND
	Realiza la operación lógica condicional OR
?:	(Ternario) Si el primer operando es verdadero, devuelve el valor del segundo; en otro caso, devuelve el tercero

OPERADORES LÓGICOS RELACIONALES

Operador	Descripción
<code>==</code>	Devuelve verdadero si ambos valores son verdaderos
<code>!=</code>	Devuelve el valor inverso a <code>==</code>
<code>></code>	Devuelve verdadero si el valor de la izquierda es mayor estricto que el de la derecha.
<code>>=</code>	Devuelve verdadero si el valor de la izquierda es mayor o igual que el de la derecha.
<code><</code>	Devuelve verdadero si el valor de la izquierda es menor estricto que el de la derecha.
<code><=</code>	Devuelve verdadero si el valor de la izquierda es menor o igual que el de la derecha.

3.

TIPOS DE DATOS DE CARACTERES

TIPOS DE DATOS DE CARACTERES

Tipo	Descripción	Literal
<code>char</code>	Nos permite almacenar un carácter UNICODE (16 bits)	'a'
<code>java.lang.String</code>	No es un tipo básico. Nos permite manejar cadenas de caracteres inmutables.	"Hola"

El operador más usual con cadenas de caracteres es +, que nos permite concatenar dos valores.

4.

CAMBIOS DE TIPOS DE DATOS (CASTINGS)

CASTINGS

- ▶ En ocasiones, nos puede interesar realizar un cambio *explícito* de un tipo de dato.

`System.out.println(5/9);` → 0

- ▶ A esta operación se le llama *casting*

`System.out.println(double5/9);` ← 0.5555555555555556

- ▶ Los tipos de datos deben ser *compatibles*.



ESTRUCTURAS DE DECISIÓN

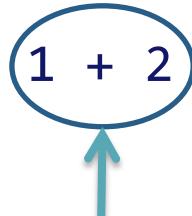
1.

EXPRESIONES, SENTENCIAS Y BLOQUES

EXPRESIONES

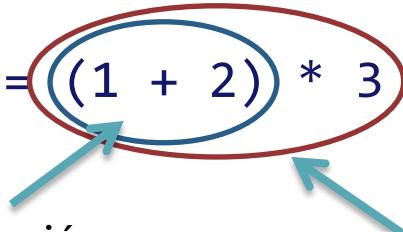
- ▶ Construcción hecha con variables, operadores y llamadas a funciones.
- ▶ Debe seguir la sintaxis de Java
- ▶ Se evalúa produciendo un solo valor.
- ▶ Se pueden componer para producir expresiones más grandes.

numero = 1 + 2



expresión

numero = (1 + 2) * 3



expresión

expresión
compuesta

SENTENCIA

- ▶ Se trata de una expresión (sencilla o compuesta) que termina en un ;
- ▶ Los programadores las conocen comúnmente como *líneas de código*.

```
int numero = (1 + 2) * 3;
```

BLOQUE DE CÓDIGO

- Una agrupación de cero o más líneas de código, delimitado por { }

```
public static void main(String[] args) {
    boolean condicion = true;
    if (condicion) { // inicio del bloque 1
        System.out.println("La condición es verdadera");
    } // final del bloque 1
    else { // inicio del bloque 2
        System.out.println("La condición es falsa.");
    } // final del bloque 2
}
```

2.

ESTRUCTURAS DE DECISIÓN

ESTRUCTURA IF-THEN

- ▶ Es la más básica en Java
- ▶ Nos permite evaluar una expresión como verdadera o falsa.
- ▶ En caso de que sea verdadera, se ejecuta un bloque de código.
- ▶ En caso de que no lo sea, dicho bloque no es ejecutado.

ESTRUCTURA IF-ELSE

- ▶ Nos permite evaluar una expresión como verdadera o falsa.
- ▶ En caso de que sea verdadera, se ejecuta un bloque de código.
- ▶ En caso de que no lo sea, *se ejecuta otro bloque de código diferente.*

ESTRUCTURA IF-ELSE-IF

- ▶ Se trata de una derivación de la anterior.
- ▶ Nos permite comprobar una segunda (o tercera, cuarta...) condición si la primera no se cumple.

ESTRUCTURA SWITCH

- ▶ Nos permite evaluar varias posibilidades sin tener que pasar por todas las anteriores.
- ▶ Funciona con byte, short, char, int, y String (entre otros).
- ▶ Después de cada caso, se usa la sentencia break, para que no ejecute el resto.
- ▶ Podemos usar un caso por defecto (parecido a else).



ESTRUCTURAS DE REPETICIÓN

BUCLE WHILE

- ▶ Nos permite repetir la ejecución de un bloque de sentencias.
- ▶ La repetición se realiza durante un número indeterminado de veces, mientras una expresión sea cierta.
- ▶ Una de las sentencias del cuerpo del bucle debe modificar alguna de las variables de la condición, para que, en alguna ocasión, la expresión sea falsa.

BUCLE WHILE

```
while(condicion) {  
    ...  
    ...  
}
```

BUCLE DO-WHILE

- ▶ Nos permite repetir la ejecución de un bloque de sentencias.
- ▶ La condición, a diferencia de la estructura *while*, se evalúa al final del bucle.
- ▶ El cuerpo del bucle se ejecuta siempre, al menos, una vez.

BUCLE DO-WHILE

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

BUCLE FOR

- ▶ Nos permite repetir un bloque de código un número **conocido a priori** de veces.
- ▶ Suele ser el más utilizado de los 3.
- ▶ Se podría implementar un bucle *for* con un *while*.
- ▶ Cuando trabajemos con colecciones o *arrays*, podremos usar su variante llamada ***for-each***.

BUCLE FOR

```
for(declaracion; condicion; incremento) {  
    ...  
    ...  
}
```

ORIENTACIÓN A OBJETOS

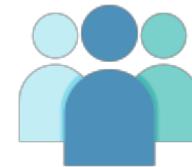
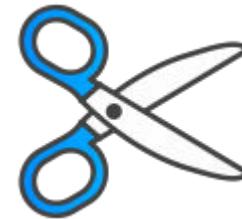
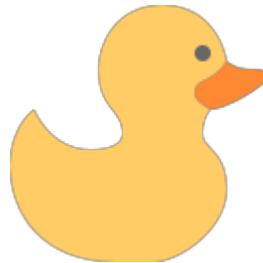


1.

EL MUNDO DE LOS OBJETOS

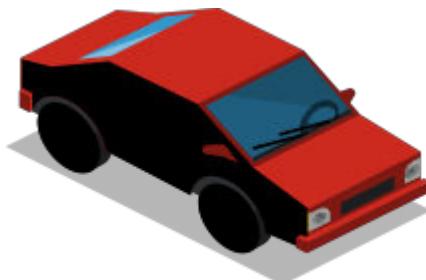
OBJETOS

- ▶ Un objeto es cualquier cosa sobre la que podemos emitir un concepto.



OBJETOS

- ▶ Podemos construir una representación de los objetos en nuestros programas.



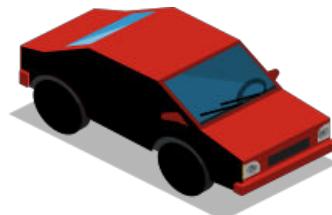
Este **automóvil de juguete** es de plástico, rojo, con 4 ruedas, 1 volante, que puede moverse adelante y atrás, ...

ESTRUCTURA Y COMPORTAMIENTO

- ▶ En general, todos los objetos tienen una estructura (como están conformado) y un comportamiento (realizan una serie de operaciones).

ESTRUCTURA

- Plástico
- 4 ruedas
- 1 volante
- ...



COMPORTAMIENTO

- Mover adelante
- Mover atrás
- ...

CLASE



Cada uno de estos elementos son **un objeto**. Pero todos ellos tienen algo en común; nos referimos a ellos como **GLOBO**.

CLASE

- ▶ Una clase es un molde con el que podemos construir objetos de un tipo.
- ▶ El *molde* determina las características y el comportamiento que podrá tener ese objeto.
- ▶ A ese molde, como tal, no lo consideramos un objeto.



CLASE vs. OBJETOS

ESTRUCTURA

- Nombre
- Edad
- Color piel
- Profesión
- Estado civil



COMPORTAMIENTO

- Hablar
- Caminar
- Mirar
- Nacer
- Morir



2.

PROGRAMACIÓN ORIENTADA A OBJETOS

PARADIGMA

- ▶ Paradigma significa modelo.
- ▶ Es la forma en la que se entiende que hay que estructurar un programa.
- ▶ Existen múltiples paradigmas.

IMPERATIVA

LÓGICA

ORIENTADA A
OBJETOS

ORIENTADA A
EVENTOS

ORIENTADA A
ASPECTOS

FUNCIONAL

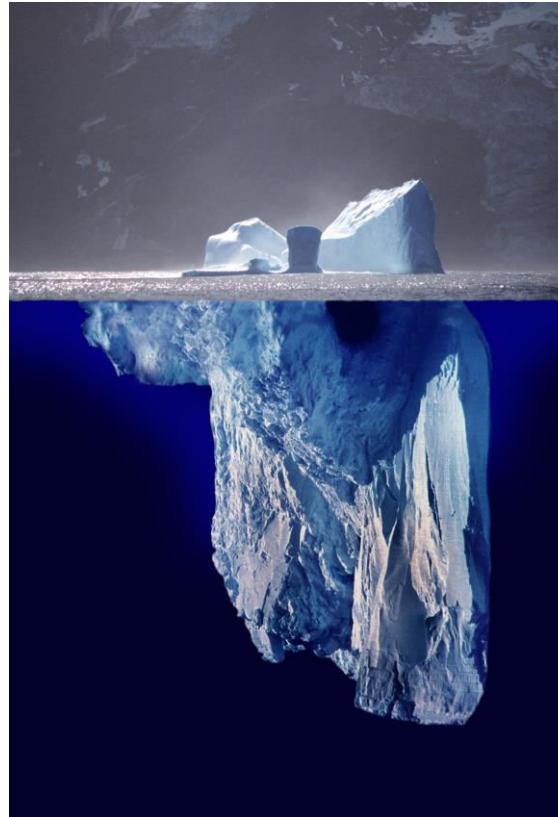
PROGRAMACIÓN ORIENTADA A OBJETOS

- ▶ Estructura todas las partes de un programa mediante objetos.
- ▶ Los objetos interaccionan entre ellos mediante un **paso de mensajes**.



ENCAPSULACIÓN

- ▶ Los objetos conocen solamente su estructura, no la de los demás.



PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

- ▶ Java es totalmente orientado a objetos.
- ▶ Nos permite manejar clases y objetos.
- ▶ Palabra reservada **class**
- ▶ La declaración e implementación de una clase estará en un mismo fichero.
- ▶ Los nombres de las clases usan notación *UpperCamelCase*.

DEFINICIÓN E IMPLEMENTACIÓN DE UNA CLASE

```
<modificador> class NombreDeLaClase {  
  
    //propiedades  
    int propiedad1;  
    String propiedad2;  
    float propiedad3;  
    //...  
  
    //metodos  
    void metodo1() {  
        //...  
    }  
  
    //...  
}
```

INSTANCIACIÓN DE OBJETOS DE UNA CLASE

- ▶ Construimos objetos con el *molde* de la clase.
- ▶ Sintaxis parecida a la declaración de una variable de tipo primitivo.
- ▶ Uso del operador **new**.

```
Persona persona = new Persona();
```

↑
Tipo (clase)

↑
Nombre del objeto
(referencia)

↑
Operador de
instanciación

←
Constructor

VALOR NULL

- ▶ NULL significa ausencia de información.
- ▶ Palabra reservada **null**, para comparar.
- ▶ Podemos declarar una referencia a un tipo de objeto, pero no construir ninguno.

Personas persona;
↑
Tipo (clase) ↑
 Nombre del objeto
 (referencia)

En este caso, esta referencia ahora mismo no nos permite acceder a ningún objeto, y almacena un valor NULL.

INTERACCIÓN ENTRE OBJETOS

- ▶ El paso de mensajes se realiza llamando a los métodos de un objeto desde otro.

```
persona.nacer();  
persona.hablar();  
persona.caminar();  
persona.morir();
```

MODIFICADORES DE ACCESO

- ▶ Nos permiten indicar quien puede hacer uso de una clase, o de sus atributos y métodos.
- ▶ *public*: cualquiera
- ▶ *private*: solo la propia clase
- ▶ *protected*: la propia clase y sus derivados
- ▶ Por defecto: las clases cercanas (que estén en el mismo paquete).

PAQUETE

- ▶ Es una unidad organizativa, que puede contener una o más clases.
- ▶ A nivel práctico, es un directorio (o un árbol de directorios).
- ▶ Nos permiten organizar las clases de forma lógica.
- ▶ Indicamos que una clase pertenece a un paquete en la cabecera de la misma (*package*).

LOS CONSTRUCTORES

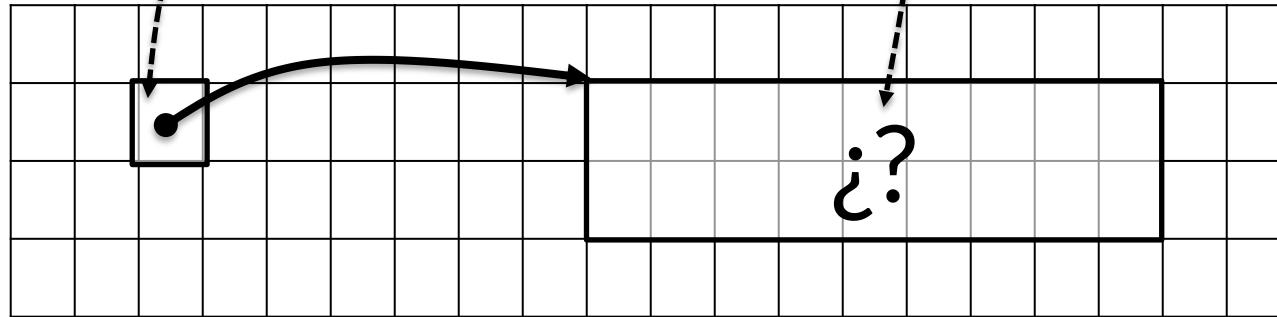


¿QUÉ SIGNIFICA CONSTUIR UN OBJETO?

Person **persona** = **new Persona();**

Tipos (clase) Nombre del objeto (referencia) Operador de instanciação Constructor

MEMORIA



CONSTRUCTORES

- ▶ Método especial, invocado con el operador **new**.
- ▶ Se ejecuta exclusivamente en el momento de creación de un objeto.
- ▶ Sirva para **inicializar valores**.
- ▶ Normalmente **public**.
- ▶ Con o sin argumentos.
- ▶ Puede haber varios en una misma clase.
- ▶ Eclipse nos ayuda a generarlos.

PUNTERO THIS

- ▶ Palabra reservada
- ▶ Sirve para hacer referencia a un objeto desde dentro de sí mismo.
- ▶ Ayuda a la encapsulación: variables miembro, métodos, constructores.

PROPIEDADES Y MÉTODOS DE UNA CLASE



MÉTODOS Y ATRIBUTOS

ESTRUCTURA

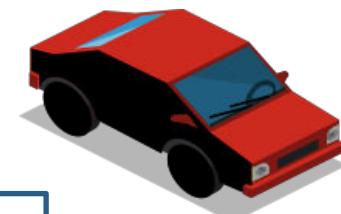
- Plástico
- 4 ruedas
- 1 volante
- ...

ATRIBUTOS

COMPORTAMIENTO

- Mover adelante
- Mover atrás
- ...

MÉTODOS



```
public class Coche {  
    private String color;  
    private String numRuedas;  
    //...
```

```
    public void adelante() {  
        //...  
    }
```

```
    public void atrás() {  
        //...  
    }  
    //...
```

ENCAPSULACIÓN

- ▶ El trato entre objetos se realiza a través de los métodos.
- ▶ Normalmente, los atributos de un objeto se deben consultar o editar a través de métodos.



PROPIEDADES

- ▶ Conforman la estructura de la clase

modificadorDeAtributo **tipoAtributo** **nombreAtributo;**

private
protected
public
Por defecto
...

char
int
float
double
String
Otra clase

Notación *camelCase*
Autodescriptivo

MÉTODOS

- Conforman el comportamiento de la clase

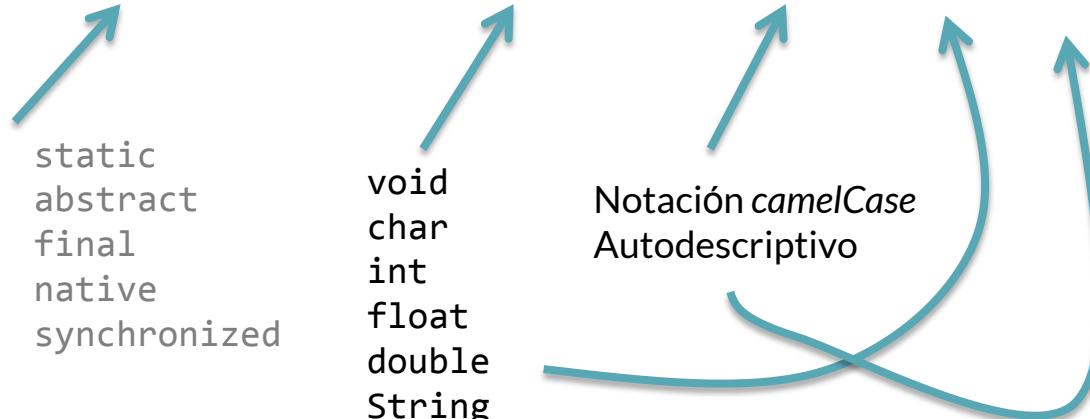
modificador[es]DeMetodo tipoRetorno nombreMetodo(tipo1 param1,){ }

public
protected
private
Por defecto

static
abstract
final
native
synchronized

void
char
int
float
double
String
Otra clase

Notación *camelCase*
Autodescriptivo



MÉTODOS



Método sin valor de salida

```
public void metodo(...) {  
    //...  
}
```

Método sin valores de entrada

```
public int metodo() {  
    return this.valor;  
}
```

GETTER/SETTER

- ▶ Métodos *especiales*, aunque los más sencillos.
- ▶ Nos permiten cambiar el valor de una propiedad o consultarla.
- ▶ Un *getter* y *setter* por propiedad
- ▶ Autogenerar con el IDE

```
public tipo getPropiedad() {           public void setPropiedad(tipo valor) {  
    return tipo;                      this.propiedad = valor;  
}                                     }  
                                         }
```

toString()

- ▶ Método especial
- ▶ Sirve para representar todo el objeto como una cadena.
- ▶ System.out.println(objeto) sin toString() y con él.
- ▶ Se puede autogenerar con el IDE

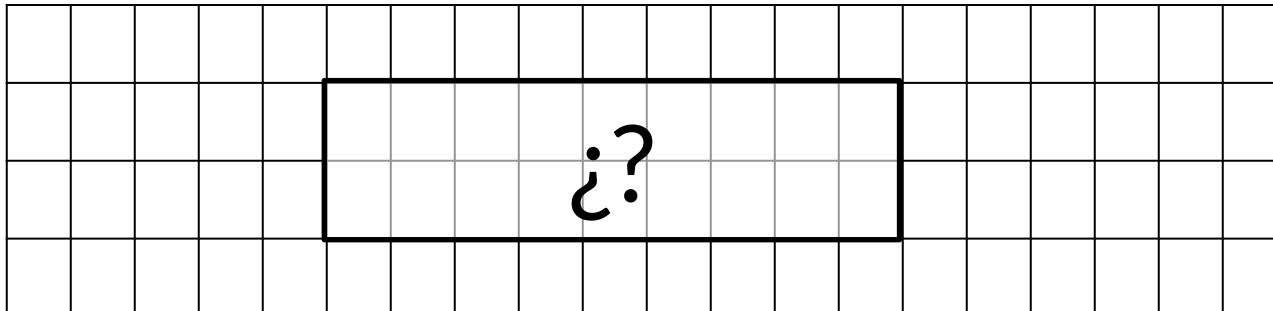


CICLO DE VIDA DE UN OBJETO

OTROS LENGUAJES DE PROGRAMACIÓN

- ▶ En otros lenguajes, las operaciones tocantes a la memoria eran *cometido* del programador.

(1) RESERVA DE MEMORIA



OTROS LENGUAJES DE PROGRAMACIÓN

- ▶ En otros lenguajes, las operaciones tocantes a la memoria eran cometido del programador.

(2) USO

Nombre= Luis Miguel
Apellidos= López

OTROS LENGUAJES DE PROGRAMACIÓN

- ▶ En otros lenguajes, las operaciones tocantes a la memoria eran cometido del programador.

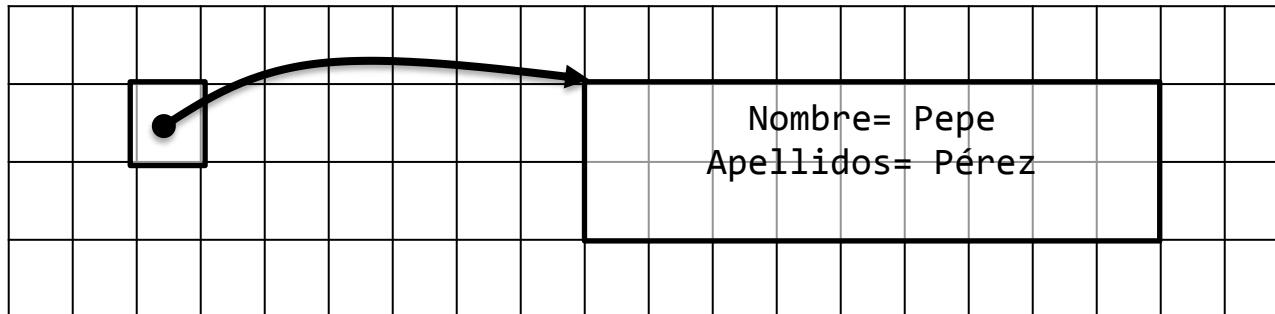
(3) DESTRUCCIÓN DEL OBJETO

Nombre= Luis Miguel
Apellidos= López

JAVA

- ▶ Java nos permite centrarnos solo en el uso
- ▶ La instancia ya realiza la reserva de la memoria necesaria.

```
Persona persona = new Persona("Pepe", "Pérez");
```

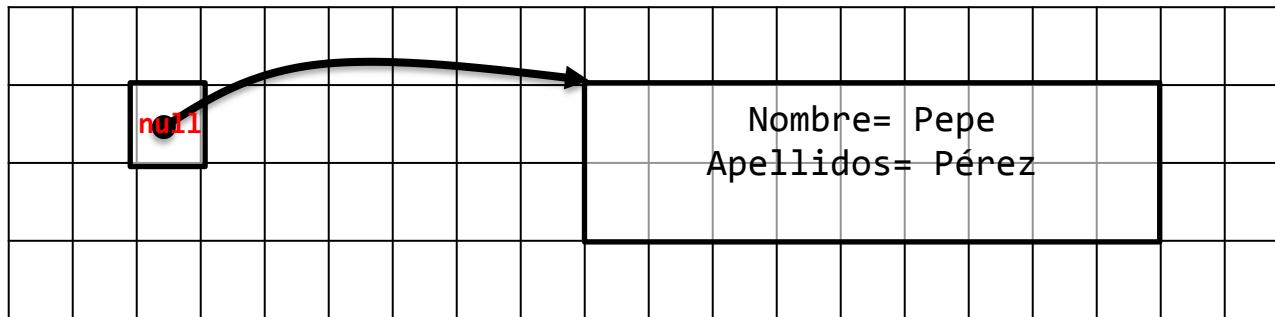


TIEMPO DE VIDA DE UN OBJETO

- ▶ Un objeto está vivo durante todo el ámbito (bloque de código) en el que ha sido instanciado.
- ▶ Durante todo ese tiempo, Java nos permite acceder a él con normalidad.
- ▶ Cuando termina ese ámbito, el objeto debe ser *enviado a la basura*.

Garbage Collector

- ▶ Java posee un recolector de basura, que se encarga de eliminar aquellos objetos que ya no son útiles.
- ▶ Para clases complejas, podemos añadir código de liberación de recursos.



USO DE CLASES ENVOLTORIO PARA TIPOS PRIMITIVOS



CLASES ENVOLTORIO

- ▶ Java tiene una clase para cada uno de los tipos de datos primitivos.

Tipo de dato primitivo	Clase envoltorio	Tipo de dato primitivo	Clase envoltorio
boolean	Boolean	int	Integer
char	Character	long	Long
byte	Byte	float	Float
short	Short	double	Double

CLASES ENVOLTORIO

- ▶ Tiene métodos interesantes para transformar valores primitivos en cadenas de caracteres y viceversa.
- ▶ Más adelante veremos que son útiles si vamos a trabajar con colecciones y otros tipos de contenedores de objetos.

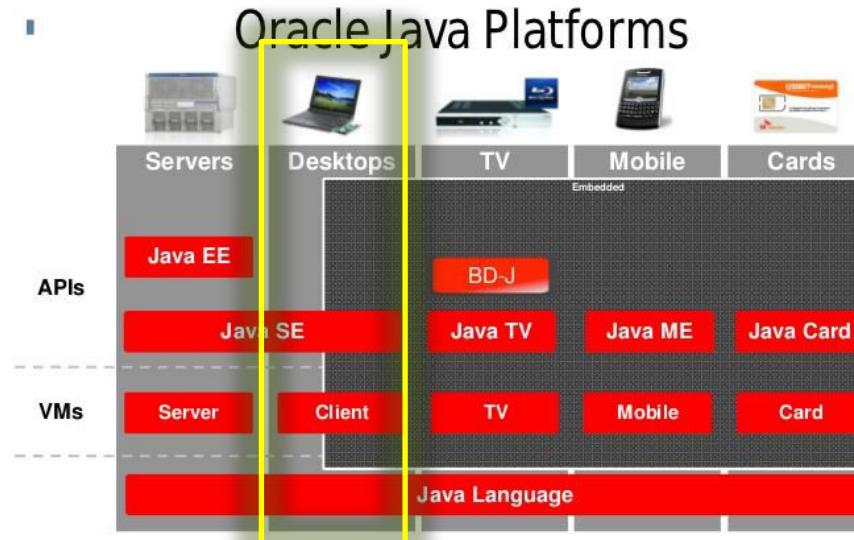


API JAVA DOCS

¿CUANTAS CLASES INCLUYE JAVA SE?

> 4000

Java pone a
nuestra
disposición más
de 4000 clases
en su versión 8

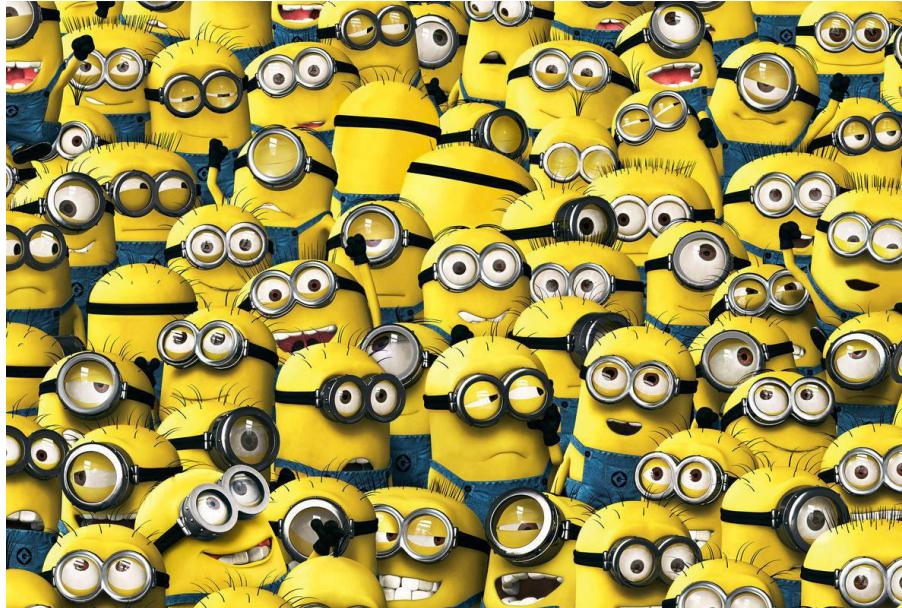


¿CUANTAS CLASES INCLUYE JAVA SE?

> 4000

¿Para qué sirve
cada una?

¿Cuáles son sus
métodos y
atributos?



DOCUMENTACIÓN DE JAVA

- ▶ Cada versión de Java tiene publicada una ayuda online para consultar la documentación de cada una de sus clases, en formato HTML.
- ▶ Se le conoce como API JAVA DOCS (o también como API).

<https://docs.oracle.com/javase/8/docs/api/index.html>



USO DE LA CLASE **STRING**

CREACIÓN DE **STRINGS**

- ▶ Java crea un **String** con un literal encerrado entre dos comillas dobles.
- ▶ Estos **String** son inmutables. Todas las operaciones que hagamos con ellos, darán como resultado uno nuevo, no la modificación del anterior (altamente ineficiente).

```
String saludo = "Hola Mundo";
```

CONCATENACIÓN DE STRINGs

- ▶ Concatenar es juxtaponer un **String** a continuación de otro.
- ▶ Se puede hacer con el operador **+** y el método **concat(...)**.

"Cadena " + "concatenada"

"Mi nombre es ".concat("Pepe")

LONGITUD DE UN STRING

- ▶ Se trata del número de caracteres que lo conforman.
- ▶ Cuentan también los espacios en blanco, tabuladores, signos de puntuación, ...
- ▶ Método **length()**

```
String saludo = "Hola Mundo!";
System.out.println(saludo.length()); //Debe imprimir 11
```

MAYÚSCULAS Y MINÚSCULAS

- ▶ Transforman una cadena completa a mayúsculas o minúsculas.
- ▶ Métodos **toLowerCase()** y **toUpperCase()**

CADERAS FORMATEADAS

- ▶ Nos permiten insertar valores dentro de una cadena a posteriori.
- ▶ Evitan concatenaciones tediosas.
- ▶ Uso del método **format(...)**

```
String.format("Hola, soy %s %s y quiero saludarte diciéndote %s",  
             nombre, apellidos, mensaje);
```

COMPARACIÓN DE **STRINGS**

- ▶ Java provee de múltiples métodos para comparar dos cadenas de caracteres.

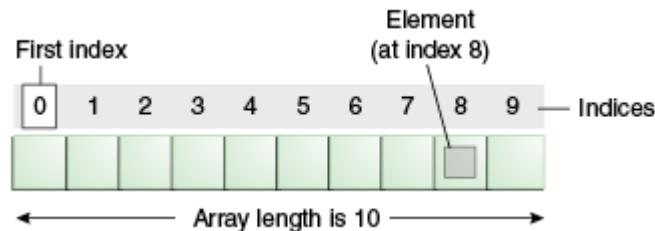
<https://docs.oracle.com/javase/8/docs/api/index.html>

CREACIÓN Y USO DE ARRAYS



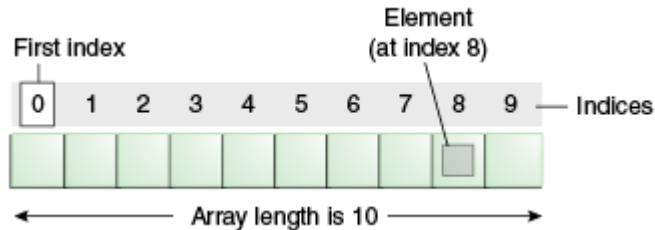
¿QUÉ ES UN ARRAY?

- ▶ Es un contenedor de elementos de un mismo tipo, con un tamaño fijo.
- ▶ Su longitud se establece en el momento de crearlo.



¿QUÉ ES UN ARRAY?

- ▶ Cada ítem de un array se suele llamar elemento.
- ▶ Se comienza a contar en cero.



CREACIÓN DE UN **ARRAY**

- ▶ Debemos indicar el tipo de dato y el tamaño.
- ▶ Tenemos que usar el operador **new**.

```
int[] unArray = new int[10];
```

Tipo de dato
del array

Los corchetes
indican que es
un array

Nombre
del array

Tamaño
del array

OPERADOR []

- ▶ Nos permite acceder a una posición concreta de un array.
- ▶ Es de lectura/escritura

```
//Asignación de un elemento  
unArray[1] = 100;
```

```
//Lectura de un elemento  
System.out.println(unArray[1]);
```

INICIALIZACIÓN DE UN **ARRAY**

- ▶ Consiste en darle valores iniciales
- ▶ Lo podemos hacer de varias formas.
 - ▶ Elemento a elemento
 - ▶ Atajo mediante la sintaxis con { }
 - ▶ Mediante un bucle (cuando la lógica del programa lo permita).

RECORRIDO DE UN ARRAY

- ▶ Todo *array* tiene una propiedad, **.length**, que nos dice su número de elementos.
- ▶ Lo podemos hacer con un bucle **for clásico** o un bucle **for mejorado**.

```
for(int i = 0; i < unArray.length; i++) {  
    System.out.println(unArray[i]);
```

```
}
```

```
for(int i : unArray) {  
    System.out.println(i);  
}
```

ARRAY DE OBJETOS

- ▶ Podemos crear arrays de cualquier **clase**.
- ▶ La sintaxis es idéntica a los tipos primitivos.

```
Persona[] unArray = new Persona[10];
```

- ▶ La inicialización requiere del uso de **new**.

```
unArray[1] = new Persona(...);
```

ARRAYs MULTIDIMENSIONALES

- ▶ Podemos crear arrays de más de una dimensión.
- ▶ Tan solo tenemos que añadir otra pareja de corchetes.

```
int[][] biArray = new int[10][20];
```

- ▶ Para acceder a sus elementos, tenemos que usar también el doble corchete.

```
biArray[3][4] = 78;
```

ARRAYs MULTIDIMENSIONALES

- ▶ Para recorrerlos necesitamos bucles anidados.

```
for(int i = 0; i < biArray.length; i++) {  
    for(int j = 0; i < biArray[0].length; j++) {  
        System.out.println(unArray[i][j]);  
    }  
}
```

MANIPULACIÓN RÁPIDA DE ARRAYs

- ▶ Java nos provee de una clase, `java.util.Arrays`, que posee muchos métodos para manipular arrays
- ▶ Ordenación, búsqueda, copia, ...

<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays .html>

ARGUMENTOS Y TIPOS DE RETORNO



1.

TIPOS DE RETORNO

FINAL DE UN MÉTODO

- ▶ Completar sus sentencias
- ▶ **return ...;**
- ▶ Error (excepción)

Los métodos que *no devuelven nada* deben devolver **void**.

TIPO DE RETORNO PRIMITIVO

- ▶ int, char, boolean, ...
- ▶ También podemos devolver un array (multidimensional)

```
public class Rectangulo {  
  
    //atributos y otros métodos  
    public float getArea() {  
        return base*altura;  
    }  
  
}
```

TIPO DE RETORNO CLASE

- ▶ Cualquier clase definida por Java o por nosotros.
- ▶ También podemos devolver un array (multidimensional)

```
public class Rectangulo {  
  
    //atributos y otros métodos  
    public Punto[] getPuntos() {  
  
    }  
  
}
```

2.

PASO DE ARGUMENTOS

ARGUMENTOS DE UN MÉTODO

- ▶ Puede no recibir argumentos
- ▶ El máximo es 255
- ▶ Tipos primitivos, arrays o clases

```
public Rectangulo rectanguloDePuntos(Punto[] esquinas) {  
    //...cuerpo del método...  
}
```

NÚMERO ARBITRARIO DE ARGUMENTOS DE UN MÉTODO

- ▶ Métodos que no sabemos a priori cuantos argumentos de un mismo tipo van a recibir.
- ▶ Se les conoce como *varargs*.
- ▶ Notación de ...
- ▶ Internamente, funciona como un array

```
public Poligono poligonoDePuntos(Punto... esquinas) {  
    int numeroLados = esquinas.length;  
    //...cuerpo del método...  
}
```

PASO DE ARGUMENTOS POR VALOR Y POR REFERENCIA



PASO DE ARGUMENTOS

- ▶ Por *valor*: se realiza una copia de las variables. Al finalizar el método, esta copia se destruye.
- ▶ Por *referencia*: no se realiza una copia de las variables. Si se modifican, quedan modificados al finalizar el método.

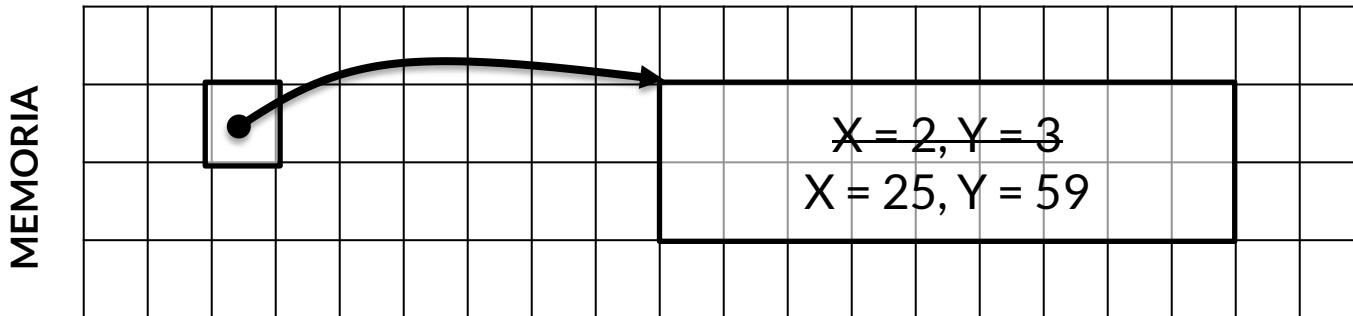
PASO DE TIPOS PRIMITIVOS

- ▶ Se hace por **valor**.

```
public class PasoPorValor {  
    public static void main(String[] args) {  
        int x = 3;  
        //invocamos el argumento y le pasamos x  
        pasoPorValor(x);  
        //imprimimos x y vemos si el parámetro ha cambiado  
        System.out.println("Después de invocar pasoPorValor, x = " + x);  
    }  
    // cambiamos el valor en el método  
    public static void pasoPorValor(int p) {  
        p = 10;  
    }  
}
```

PASO DE OBJETOS

- ▶ También se hace por **valor**.
 - ▶ No cambia la referencia, pero el *interior* del objeto sí se puede modificar.



MODIFICADORES DE ATRIBUTOS Y MÉTODOS



¿QUIÉN ACCEDE A MIS ATRIBUTOS, MÉTODOS o CLASES?

Modificador	Clase	Paquete	Subclase	El Mundo
public	■	■	■	■
protected	■	■	■	■
<i>Sin modificador</i>	■	■	■	■
private	■	■	■	■



Sí puede acceder



No puede acceder

MODIFICADORES DE ACCESO A NIVEL DE CLASE

- ▶ Nuestras clases deben ser **public** o sin modificador.
- ▶ En caso de ser **public**, cualquiera podrá utilizarlas (más recomendado)
- ▶ Sin modificador, solamente las clases del mismo paquete podrán usarlas.

MODIFICADORES DE ACCESO A NIVEL DE MÉTODOS

- ▶ Intentemos escoger siempre la versión más restrictiva posible.
- ▶ Para métodos que realizan operaciones *auxiliares* a otros métodos, podemos escoger *private*.
- ▶ La mayoría de los métodos serán **public**. Son su interlocutor con el exterior.
- ▶ Los constructores deben ser **public**.

MODIFICADORES DE ACCESO A NIVEL DE ATRIBUTOS

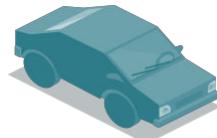
- ▶ Deben ser **private**, salvo para constantes.
- ▶ Hay que tener una muy buena razón para no usarlo.
- ▶ Los atributos públicos aumentan el acoplamiento del código, y limitan la flexibilidad de refactorización.

ATRIBUTOS O MÉTODOS ESTÁTICOS



ATRIBUTOS DE OBJETO Y DE CLASE

- ▶ Los objetos son instancias de una clase.
- ▶ Cada objeto tiene una copia de los atributos.
- ▶ ¿Y si quisiéramos tener un **atributo común** a todos los objetos de una clase?
- ▶ **static**



ATRIBUTOS ESTÁTICOS

- ▶ Están asociados a la clase, no a una instancia de ella.
- ▶ Son llamados **atributos estáticos**.
- ▶ **Compartidos** para todas las instancias de esa clase.
- ▶ Pueden ser manipulados por cualquier instancia.
- ▶ También pueden ser manipulados sin crear instancias de esa clase.

MÉTODOS ESTÁTICOS

- ▶ Similares a las variables estáticas (static)
- ▶ Se pueden invocar sin crear una instancia de esa clase.
- ▶ **Clase.metodoEstatico(...);**
- ▶ Para acceder a una variable estática, necesitamos un método estático.
- ▶ Clases con métodos auxiliares (como por ejemplo, *java.util.Arrays*).

CONSTANTES

- ▶ Se suelen definir como estáticas.
- ▶ static final ...

```
static final double PI = 3.141592653589793;
```

- ▶ No se puede modificar su valor (error)
- ▶ Nombre en mayúsculas, separando palabras con guiones bajos.

SOBRECARGA DE MÉTODOS Y CONSTRUCTORES



SOBRECARGA DE MÉTODOS

- ▶ Java soporta que una clase tenga dos (o más) métodos con **el mismo nombre**.
- ▶ Deben tener diferente *firma*: número de argumentos o tipos diferentes.
- ▶ No se pueden tener dos métodos con mismo nombre y misma firma (indistinguibles).
- ▶ Java permite su uso, pero no recomienda su uso masivo (código menos legible).

SOBRECARGA DE CONSTRUCTORES

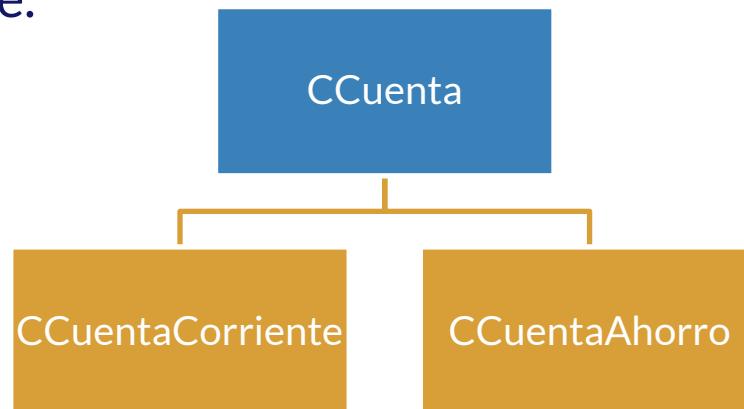
- ▶ Muy habitual.
- ▶ Nos permiten construir un mismo objeto de diferentes formas.
- ▶ Siguen las mismas normas que la sobrecarga de métodos.

HERENCIA



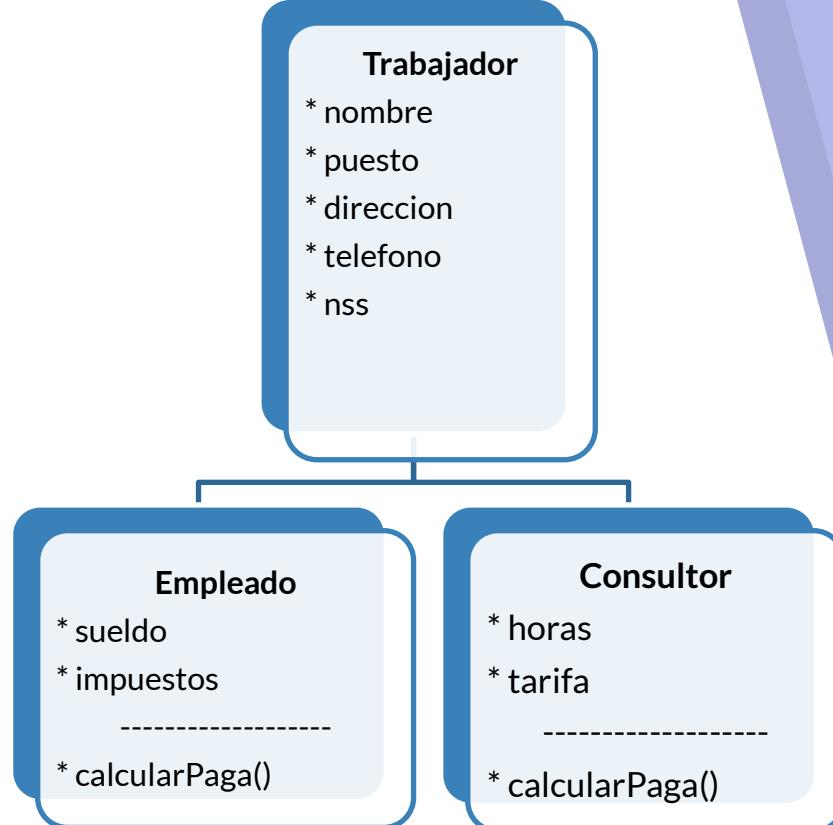
HERENCIA

- ▶ Funcionalidad fundamental en POO.
- ▶ Mecanismo de extensión de clases.
- ▶ Jerarquía de clases.
- ▶ Superclase y subclase.
- ▶ Reutilización.
- ▶ Un solo parente.
- ▶ **extends**



HERENCIA

- ▶ Una clase que extiende a otra hereda sus atributos y sus métodos (no constructores).
- ▶ Puede añadir atributos y métodos nuevos.



HERENCIA y ACCESO

- ▶ Una clase que extiende a otra hereda todos sus atributos y métodos.
- ▶ Solamente puede acceder a los que sean *public* y *protected* (y por defecto si está en el mismo paquete).
- ▶ ***protected*** está poco recomendado para las propiedades. Mejor ***private*** y acceso a través de métodos ***public***.

HERENCIA y SOBREESCRITURA

- ▶ Una clase que extiende a otra hereda puede añadir tantos métodos o atributos como necesite.
- ▶ Si un nuevo atributo (o método) se llama igual que otro de una superclase, lo solapa, y ya no puede accederse al de la clase padre.

HERENCIA y **final**

- ▶ Si no queremos que nadie pueda heredar de una de nuestras clases, podemos marcarla como **final** en su definición.

```
public final class ClaseFinal {  
}
```



POLIMORFISMO

REFERENCIAS Y SUBCLASES

- ▶ Una subclase puede ser accedida a través de una referencia de una superclase.
- ▶ Esto es muy útil, sobre todo, para usar como atributos de métodos.

```
public static void saludar(Trabajador t) {  
    System.out.println("Hola, " + t.getNombre());  
}
```

OCULTACIÓN DE MÉTODOS

- ▶ Si una subclase añade un método con mismo nombre y firma que otro de la clase base, oculta a este.
- ▶ ¿Qué sucede en caso de el uso de referencias de clase base, pero instanciación de objetos derivados?

```
Trabajador empleado;  
empleado = new Empleado("Larry Ellison", "Presidente",  
                         "Redwood", "", "", 100000.0, 1000.0);  
empleado.calcularPaga();
```

POLIMORFISMO

- ▶ Java escoge, en tiempo de ejecución, el tipo de objeto.
- ▶ Si ese tipo ha ocultado un método de la superclase, llama a la *concreción*.
- ▶ En otro caso, llama al método de la clase base.



USO DE super

ACCESO A LA SUPERCLASE

- ▶ Si uno de nuestros métodos sobreescribe un método de la clase base, podemos invocar este a través de la palabra **super**.

```
// overrides printMethod in Superclass
public void printMethod() {
    super.printMethod();
    System.out.println("Printed in Subclass");
}
```

CONSTRUCTORES Y SUPER

- ▶ Un constructor de una subclase puede usar **super** para invocar a un constructor de su clase base.
- ▶ Si una subclase no lo invoca, la JVM lo hace por él. La clase base debe tener entonces un constructor sin parámetros.

```
public Empleado(...) {  
    super(nombre, puesto, direccion, telefono, nSS);  
    this.sueldo = sueldo;  
    this.impuestos = impuestos;  
}
```

INTERFACES Y CLASES ABSTRACTAS



1.

INTERFACES

INTERFACES EN JAVA

- ▶ Contrato de compromiso.
- ▶ Conjunto de operaciones que una clase se compromete a implementar.
- ▶ La interfaz marca qué métodos, con su firma
- ▶ Desde Java 8, pueden incluir también métodos con cuerpo (abstractos, estáticos y por defecto).
- ▶ También puede incluir constantes.

DEFINICIÓN DE INTERFACES

- ▶ **interface**
- ▶ Mismas normas de acceso que una clase.
- ▶ Mismas reglas de nombres que una clase.
- ▶ También existe herencia de interfaces (**extends**). En este caso, sí que puede ser múltiple.

```
public interface GroupedInterface extends Interface1, Interface2 {  
    // constant declarations  
    // base of natural logarithms  
    double E = 2.718282;  
    // method signatures  
    void doSomething (int i, double x);  
    int doSomethingElse(String s);  
}
```

IMPLEMENTACIÓN DE INTERFACES

- ▶ **implements**
- ▶ Una clase puede implementar más de una interfaz

```
public class RectanglePlus implements Relatable {  
    //...  
    public int isLargerThan(Relatable other) {  
        RectanglePlus otherRect = (RectanglePlus)other;  
        if (this.getArea() < otherRect.getArea())  
            return -1;  
        else if (this.getArea() > otherRect.getArea())  
            return 1;  
        else  
            return 0;  
    }  
}
```

INTERFACES COMO TIPOS

- ▶ Una interfaz puede ser el tipo de dato a usar para crear una instancia de un objeto.
- ▶ La clase del objeto a crear debe implementar dicha interfaz.
- ▶ Muy útil, sobre todo, para recibir argumentos en métodos.

```
RectanglePlus rectangleOne = new RectanglePlus(10, 20);  
Relatable rectangleTwo = new RectanglePlus(20, 10);
```

MÉTODOS POR DEFECTO

- ▶ Novedad en Java 8
- ▶ **default.**
- ▶ Un método puede tener una implementación por defecto, descrita en la interfaz.

```
public interface Interfaz {  
  
    default public void metodoPorDefecto() {  
        System.out.println("Este es uno de los nuevos  
                           métodos por defecto");  
    }  
  
}
```

MÉTODOS ESTÁTICOS

- ▶ Novedad en Java 8
- ▶ **static.**
- ▶ Misma sintaxis que los métodos estáticos en clases

```
public interface Interfaz {  
  
    public static void metodoEstatico() {  
        System.out.println("Método estático en un interfaz");  
    }  
}
```

2.

CLASES ABSTRACTAS

CLASES ABSTRACT

- ▶ Clase definida como **abstract**.
- ▶ No se pueden crear instancias de la misma.
- ▶ Puede tener métodos con implementación y atributos.

```
public abstract class AbstractaSencilla {  
  
    public void saluda() {  
        System.out.println("Hola mundo!!!");  
    }  
  
}
```

MÉTODOS ABSTRACT

- ▶ Deben estar en una clase definida como **abstract**.
- ▶ Definen la firma del método, pero sin implementación.
- ▶ Sus subclases se comprometen a implementarlo.
- ▶ Si no lo hacen, también deben ser abstractas.
- ▶ Pueden convivir con métodos normales.

```
public abstract class AbstractaConMetodos {  
  
    public abstract void saludo(String s);  
  
    public void saludar() {  
        System.out.println("Hola mundo!!!!");  
    }  
}
```

3.

CLASES ABSTRACTAS vs INTERFACES

INTERFACES	CLASES ABSTRACTAS
No se pueden instanciar	No se pueden instanciar
Métodos sin implementación	Métodos sin implementación
Métodos con implementación por defecto	Métodos con implementación por defecto
Atributos estáticos o constantes	Cualquier tipo de atributos
Métodos públicos o por defecto	Métodos públicos, privados, protegidos o por defecto.
Una clase puede implementar varios interfaces	Una clase solo puede heredar de otra

¿QUÉ USAR?

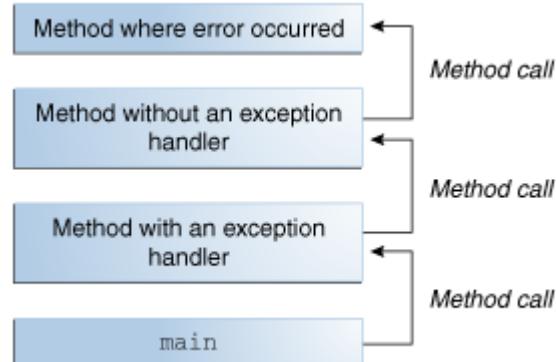
INTERFACES	CLASES ABSTRACTAS
Clases no relacionadas podrán implementar los métodos.	Compartir código con clases muy relacionadas.
Si se quiere indicar que existe un tipo de comportamiento, pero no sabemos quien lo implementa.	Las clases derivadas usarán métodos <i>protected</i> o <i>private</i> .
Si necesitamos tener <i>herencia múltiple</i> .	Queremos definir atributos que no sean estáticos o constantes.

EXCEPCIONES



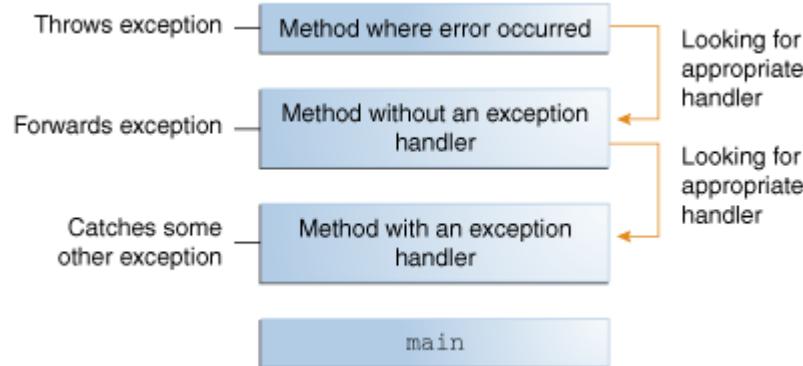
EXCEPCIÓN

- ▶ *Situación excepcional.*
- ▶ Altera la ejecución normal del programa.
- ▶ El método donde sucede crea un objeto, llamado *objeto de excepción*, y se lo pasa a alguien que pueda tratarlo.



EXCEPCIÓN

- ▶ Si existe quien pueda manejarlo, la recoge.



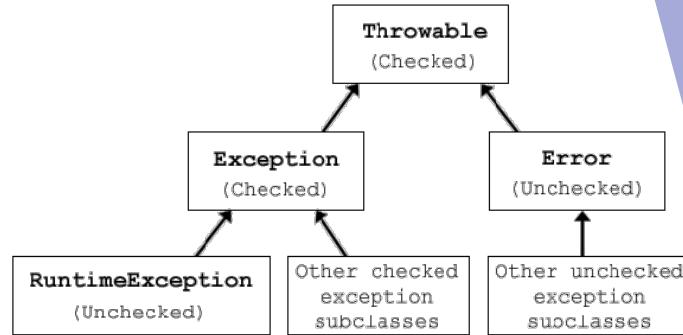
- ▶ Si no existe, se encarga la JVM.

USO DE EXCEPCIONES

- ▶ Permiten separar el código de tratamiento de errores del código normal.
- ▶ Evitan que haya errores inadvertidos.
- ▶ Permiten la propagación de los errores.
- ▶ Permiten agrupar en un lugar común el tratamiento de errores.

TIPOS DE EXCEPCIONES

- ▶ *Checked Exceptions:* excepciones que son recogidas y tratadas por programas bien escritos.
- ▶ *Error:* son externos a la aplicación, y no nos podemos anticipar a ellos.
- ▶ *Runtime error:* situaciones internas a la aplicación, y de las que no nos podemos recuperar



TIPOS DE EXCEPCIONES

Usaremos *Checked Exceptions* cuando:

- ▶ La excepción es la única manera de detectar el error.
- ▶ No queremos que pase inadvertido

Usaremos *Unchecked Exceptions* cuando:

- ▶ Podemos intentar mejorar el código para que no suceda dicho error
- ▶ La excepción sirve para detectar y corregir usos indebidos de la clase.
- ▶ Errores internos ante los que poco podemos hacer.



TRATAMIENTO DE EXCEPCIONES

TRATAMIENTO DE EXCEPCIONES

```
try {  
    instrucciones; ←  
} catch (Exception e) {  
    instrucciones; ←  
} finally {  
    instrucciones ←  
}
```

Código propio de la aplicación.

Código que trata la situación excepcional.

Código que se ejecuta tanto si se han finalizado las instrucciones de try como si ha habido una Excepción.

TRATAMIENTO DE EXCEPCIONES

- ▶ **finally** no es obligatorio.
- ▶ Puede haber más de un **catch**.
- ▶ Los tipos de excepción deben ir de más concretos a más genéricos.
- ▶ El operador | nos permite tratar más de un tipo de excepción en un **catch**.

BLOQUE TRY

- ▶ Debe envolver las sentencias que son susceptibles de provocar uno o varios tipos de excepción.
- ▶ Debemos agrupar las sentencias que vayan a tener un tratamiento idéntico de la situación excepcional.

BLOQUE CATCH

- ▶ Puede haber uno, o más de uno.
- ▶ Nos permite definir los manejadores de las excepciones.
- ▶ Cada bloque maneja uno o varios tipos de excepción.

BLOQUE FINALLY

- ▶ Se ejecuta siempre (si venimos de *try* o de *catch*).
- ▶ Se suele utilizar como código que asegura el cierre de recursos abiertos (ficheros, bases de datos, ...).

EXCEPCIONES MÁS COMUNES



EXCEPCIONES MÁS COMUNES

CLASE DE EXCEPCIÓN	USO
ArithmetricException	Errores en operaciones aritméticas
ArraryIndexOutOfBoundsException	Índice de array fuera de los límites
ClassCastException	Intento de convertir a una clase incorrecta
IllegalArgumentException	Argumento ilegal en la llamada a un método
IndexOutOfBoundsException	Índice fuera de colección
NegativeArraySizeException	Tamaño de array negativo
NullPointerException	Uso de referencia nula
NumberFormatException	Formato de número incorrecto
StringIndexOutOfBoundsException	Índice usado en String fuera de los límites

CLASES DE EXCEPCIÓN

- ▶ Heredan métodos de *Throwable*.
- ▶ Constructores que incluyen la posibilidad de paso de un mensaje.
- ▶ Métodos para obtener información de la excepción.
- ▶ ***printStackTrace()*** es el método que se invoca cuando no tratamos una excepción.

<https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>

LANZAMIENTO Y PROPAGACIÓN DE EXCEPCIONES



LANZAMIENTO DE EXCEPCIONES

- ▶ Cualquier código puede lanzar excepciones (hecho por java, por nosotros o de terceros).
- ▶ Si no vamos a tratar las excepciones en un método, tenemos que indicar que se relanzarán hacia arriba (**throws**).

USO DE THROWS

- ▶ Un método cuyo código puede producir excepciones puede capturarlas y tratarlas, o relanzarlas para que sea otro quien las trate.
- ▶ **throws.** Lista separada por comas de tipos de excepción.

```
public static void writeList() throws IOException {  
}
```

EXCEPCIONES PROPIAS

- ▶ Podemos crear nuestros propios tipos, extendiendo a **Exception**.
- ▶ Nos permiten manejar nuestras propias situaciones.

```
public class SaldoNegativoException extends Exception {  
  
    public SaldoNegativoException(double saldo) {  
        super("La cuenta ha quedado en descubierto (" +  
              Double.toString(saldo) + ")");  
    }  
}
```

USO DE THROW

- ▶ Nos permite lanzar una excepción en un momento determinado.
- ▶ También se puede usar en el bloque catch, para tratar una excepción, pero aun así relanzarla.

```
public void sacarDinero(double cantidad) throws SaldoNegativoException {  
    saldo -= cantidad;  
    if (saldo < 0) {  
        throw new SaldoNegativoException(saldo);  
    }  
}
```



USO DE STRINGBUILDER

¿POR QUÉ STRINGBUILDER?

- ▶ String es inmutable.
- ▶ Un objeto StringBuilder es un String que se puede modificar.
- ▶ Métodos más eficientes.

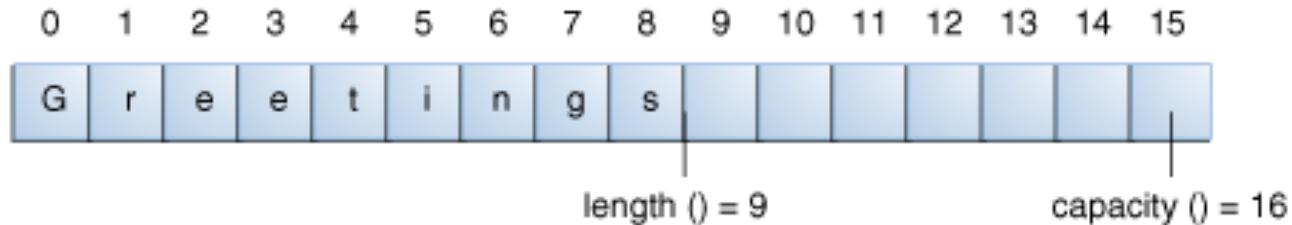
TAMAÑO Y CAPACIDAD

- ▶ Tamaño: **length()**. (Igual que String)
- ▶ Capacidad: número de caracteres que puede alojar. **capacity()**

Constructor	Descripción
StringBuilder()	Crea uno vacío, con capacidad = 16
StringBuilder(CharSequence cs)	Crea uno con los caracteres de cs, y 16 elementos vacíos adicionales.
StringBuilder(int initialCapacity)	Crea uno vacío, con la capacidad <i>initialCapacity</i>
StringBuilder(String s)	Crea uno con los caracteres de s, y 16 elementos vacíos adicionales.

TAMAÑO Y CAPACIDAD

```
// creates empty builder, capacity 16  
StringBuilder sb = new StringBuilder();  
// adds 9 character string at beginning  
sb.append("Greetings");
```



MÉTODOS DE TAMAÑO Y CAPACIDAD

Método	Descripción
<code>void setLength(int newLength)</code>	Cambia la longitud. Si <i>newLength</i> es menor que la actual, los últimos caracteres son truncados. Si es mayor, se añaden elementos vacíos.
<code>void ensureCapacity(int minCapacity)</code>	Nos asegura que la capacidad sea mayor o igual que <i>minCapacity</i> .

Algunos métodos, como **append()**, pueden aumentar la capacidad de nuestro `StringBuilder`.

OPERACIONES CON STRINGBUILDER

Método	Descripción
StringBuilder append(String s) StringBuilder append(tipoPrimitivo t)	Añade el argumento que hemos pasado al StringBuilder. Si el dato no es String, se convierte antes de pasarlo.
StringBuilder delete(int start, int end) StringBuilder deleteCharAt(int index)	Eliminan una secuencia de caracteres o un carácter.
StringBuilder insert(int offset, String s) StringBuilder insert(int offset, tipoPrimitivo t)	Inserta el segundo argumento en la cadena. El primer entero indica la posición.
StringBuilder replace(int start, int end, String s) void setCharAt(int index, char c)	Reemplaza un carácter o una serie de ellos
StringBuilder reverse()	Devuelve la cadena invertida
String toString()	Transforma el StringBuilder en un String.

JAVA 8 DESDE 0



MIS DATOS

- ▶ Luis Miguel López Magaña
- ▶ 15 años desarrollando aplicaciones Java
(Java SE, Java EE, Spring, Hibernate,
Android, ...)
- ▶ Profesor de FP desde hace 10 años.

REQUISITOS

- ▶ No es necesario ningún requisito en particular.
- ▶ Cualquier programador agradece tener capacidad analítica, algo de base en matemáticas y un punto de creatividad.

¿POR QUÉ JAVA 8?

- ▶ Según Oracle, 3.000 millones de dispositivos en el mundo usan Java



Sistemas de vuelo



Cajeros automáticos



Reproductores de BlueRay



Libros electrónicos
Kindle



Routers



Robótica

...

¿POR QUÉ JAVA 8?



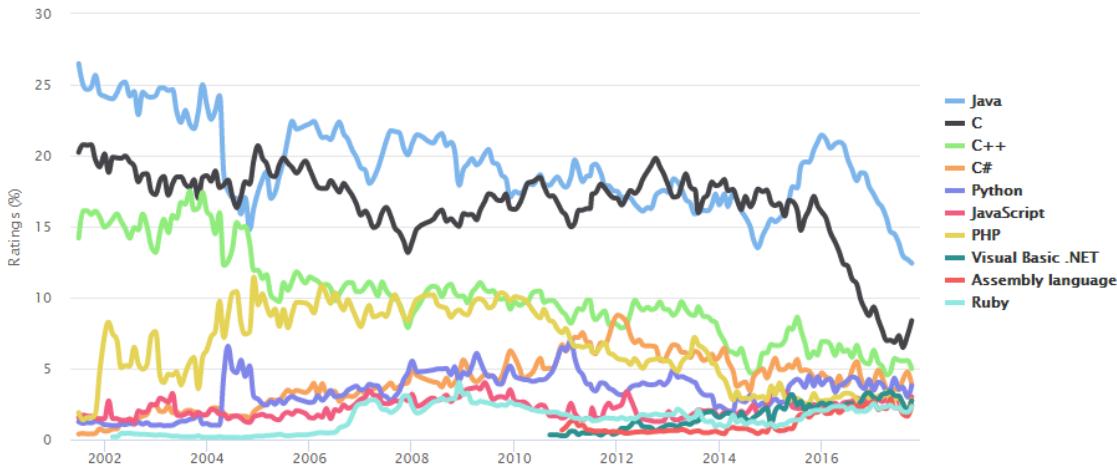
ANDROID

Si sabes programar en Java, ya tienes más de la mitad del camino hecho para aprender Android.

¿POR QUÉ JAVA 8?

TIOBE Programming Community Index

Source: www.tiobe.com



Java

Nº 1 en
ranking
TIOBE

<https://openwebinars.net/blog/cuanto-gana-un-programador-java-en-espana/>

CONTENIDOS

1. Comienza con Java: instalación, IDE y sintaxis inicial.
2. Tipos de datos, operadores y estructuras de control.
3. Clases y objetos
4. Manipulación y tratamiento de datos (API de Java, String, arrays)

CONTENIDOS

5. Revisitando clases y objetos:
argumentos, modificadores,
6. Uso de herencia (e interfaces y
polimorfismo)
7. Manejo de excepciones
8. Algunas clases del API de Java
(StringBuilder, fechas, ArrayList,
expresiones lambda).

PRÁCTICAS

Practicaremos la sintaxis de cada una de las diferentes lecciones.

3 proyectos de ejemplo

- ▶ Una sencilla calculadora
- ▶ Hundir la flota
- ▶ Gestión de un parking

¿QUÉ SERÉ CAPAZ DE HACER AL FINAL DEL CURSO?

- ▶ Programar aplicaciones Java que se ejecuten en diferentes sistemas operativos.
- ▶ Manejar todos los conceptos de programación orientada a objetos.
- ▶ Utilizar tipos de datos complejos, como fechas.

¿QUÉ SERÉ CAPAZ DE HACER AL FINAL DEL CURSO?

- ▶ Utilizar el nuevo estilo de programación funcional, incorporado en Java SE 8
- ▶ Manejar situaciones de error y darle un tratamiento adecuado.

¿QUÉ CURSOS PODRÉ REALIZAR AL TERMINAR ESTE?

- ▶ Java 8 para programadores Java.
- ▶ Java EE
- ▶ JSF
- ▶ Spring
- ▶ Hibernate
- ▶ Android
- ▶ ...