

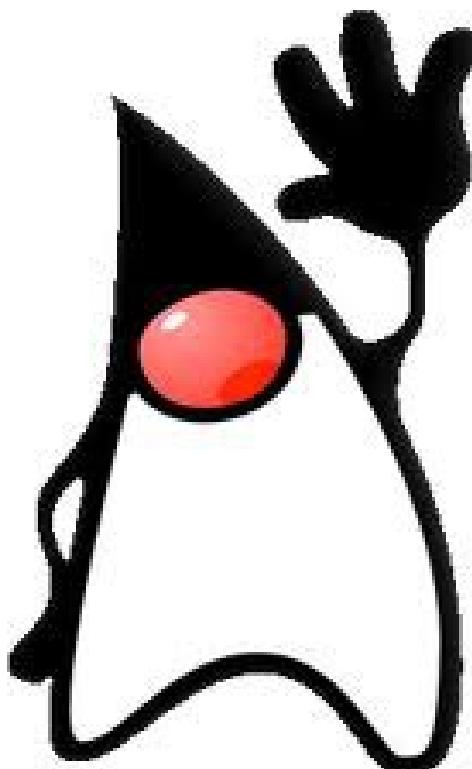


# **CIFP César Manrique**

**CFGS DAW – Desarrollo web en entorno  
servidor**



# EL LENGUAJE JAVA



**Hola  
soy  
Duke**



# CONTENIDOS

- Estructura básica de un programa
- Entrada / salida básica
- Estructura léxica
- Tipos de datos
- Conversiones y promociones
- Nombres (acceso a miembros)



# ELEMENTOS BÁSICOS DEL PROGRAMA





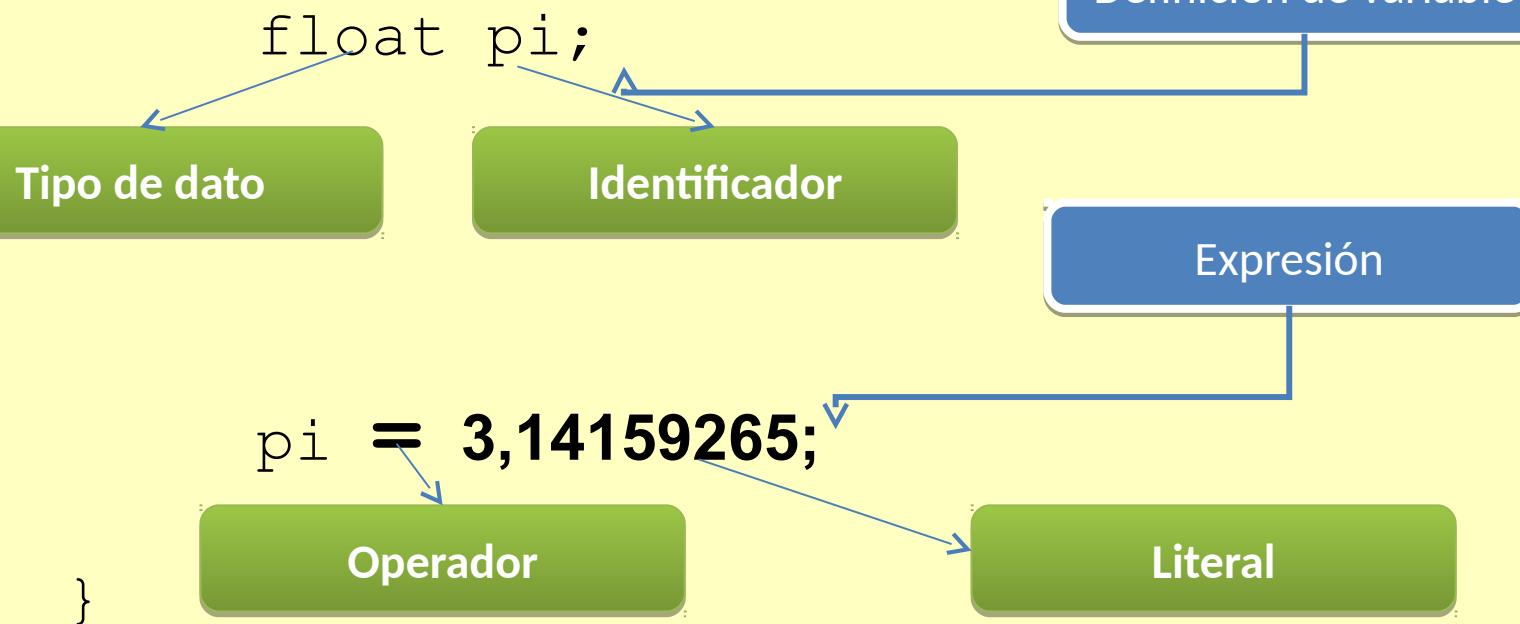
# ELEMENTOS BÁSICOS DEL PROGRAMA

- Los elementos básicos para crear un programa son:
- **Literales:** Valores directos que utilizamos en el programa (ej: el valor **3.141592** que representaría el número  $\pi$ )
- **Variables:** Indican posiciones de la memoria donde se guardan los datos
- **Operadores:** Nos permiten realizar operaciones con los valores literales y las variables (ej: el operador suma **+**)
- **Expresiones:** Combinación de variables y operadores (ej:  $e = v * t$  calcula el espacio recorrido con una velocidad y lo guardaría en e)

# LITERALES, VARIABLES, OPERADORES Y EXPRESIONES

- Una variable se define con un tipo de dato y un identificador.
- Una variable puede tomar un valor literal con el operador de asignación

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



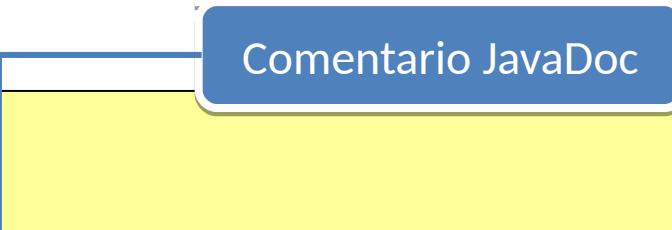
# MÉTODO MAIN O PUNTO DE ENTRADA (ENTRY POINT)

- Todo programa de escritorio o consola, tendrá al menos una clase, en la que debe existir un método denominado **main**.
- Este método tiene que tener la siguiente firma (*signature*)

**public static void main(String[] )**

Ejemplo:

```
 /**
 * @param args
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub
}
```



Comentario JavaDoc



## SALIDA BÁSICA A CONSOLA

- Para escribir en la salida estándar (la consola), utilizaremos la propiedad estática **out** de la clase **System**
- Ejemplo: **System.out.print("Hola mundo");**
- La propiedad **out** es un flujo de salida (Stream) que tiene dos métodos para escribir en la salida estándar:
  - **print**: Escribe un String o cadena de caracteres
  - **println**: Igual que el anterior, pero además inserta un salto de línea al final

```
public static void main(String[] args) {  
    System.out.print("Hola");  
    System.out.println("Adios");  
    System.out.print("¿Qué tal?");  
}
```



# ENTRADA BÁSICA DESDE CONSOLA

- La forma estándar (hasta la versión 1.4) era leer del flujo (*stream*) de entrada estándar, disponible en la propiedad **in** de la clase **System**. Para facilitar esta operación se creaba un objeto de tipo **BufferedInputStream** y se podían leer líneas enteras.
- Esto lo veremos más adelante, cuando veamos los stream con más profundidad, ahora utilizaremos una nueva clase (disponible desde Java 5), denominada **Scanner**
- La clase **Scanner** no pertenece al paquete por defecto (**java.lang**) por lo que si la utilizamos tenemos que declarar su importación

```
import java.util.Scanner;  
  
public class AppMain {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
    }  
}
```

Sentencia import



## USO DE LA CLASE SCANNER

- Tenemos que crear un objeto de esta clase. A este objeto tenemos que indicarle de qué stream va a leer, en nuestro caso leerá de la entrada estándar (**System.in**)

```
Scanner sc = new Scanner(System.in);
```

El operador **new** permite crear un objeto de una clase concreta (instanciarlo)

- A continuación tenemos métodos para leer distintos tipos de datos; por ejemplo `nextInt` para leer un entero.

```
int i;  
i = sc.nextInt();
```



## EJERCICIOS

- Crear un programa en Java que visualice el texto **Hola mundo**
- Crear un programa en Java que pida un número y lo devuelva multiplicado por 10
- Crear un programa en Java que pida un nombre (ej: María) y visualice un saludo: **Bienvenido/a María**
- Crear un programa en Java que pida dos números y muestre su suma
- Crear un programa que pida el año de nacimiento (yyyy), el año actual y muestre la edad



# ESTRUCTURA LÉXICA

Las palabras del lenguaje





# ESTRUCTURA LÉXICA

- **Codificación:** Los programas se escriben en **UNICODE**
  - **Secuencias de escape UNICODE:** \uhhhh ó \uHHHH (h dígito hexadecimal)  
Ej: \u00A9 → ©
- **Comentarios:** Documentación que describe el programa
  - **// Comentario en línea.** El texto que va a continuación, hasta fin de línea, se considera comentario y por tanto se excluye del proceso de compilación
  - **/\* .. \*/ Comentario de múltiples líneas o tradicional.** Todo el texto, abarque o no más de una línea, comprendido entre los grupos de caracteres /\* y \*/ se considera comentario y se excluye del proceso de compilación
  - **/\*\* ... \*/ Comentario Javadoc.** Funciona igual que el comentario multi-línea, pero el texto considerado comentario es procesado por la herramienta **javadoc** para generar documentación del programa.



<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>



# ESTRUCTURA LÉXICA

- **Identificadores:** Secuencia de letras y caracteres válidos en Java, usados para nombrar elementos del lenguaje (ej: variables, clases, paquetes, etc.).

Deben obedecer las siguientes reglas:

- Comienzan por una **letra java**; es decir: caracteres de la **a-z** o **A-Z**, guión bajo (**\_**) o signo del dólar (**\$**)
- No pueden ser iguales a ninguna **palabra reservada** del lenguaje (**keyword**), ni ser iguales a los literales booleanos: **true**, **false**, ni al literal nulo: **null**
- Se permite el uso de letras pertenecientes a la tabla UNICODE, que no existen en ASCII estándar (por ejemplo la **ñ** o la **Ñ**)



# EJERCICIO

- Comprobar las reglas de creación de identificadores, en la definición de variables en el lenguaje Java.
- Verificar si los siguientes identificadores son válidos para nombrar variables en Java (definir una variable de tipo entero **int xxx;**)
  - **miVariable**
  - **mi otraVariable** // espacio en blanco entre mi y otraVariable
  - **\_miVariable** // guión bajo de prefijo
  - **-miVariable** // guión alto de prefijo
  - **\$miVariable** // signo dólar de prefijo
  - **#miVariable** // almohadilla de prefijo
  - **ñoñeria** // uso de caracteres no ASCII estándar
  - **ñoñería** // uso de tildes
  - **miVar05** // uso de dígitos
  - **5var** // dígitos al comienzo del identificador
- El programa deberá mostrar cada identificador y decir si es o no válido



# PALABRAS RESERVADAS

- Identificadores reservados para la construcción de sentencias del lenguaje (bucles, condicionales, etc.)

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



# LITERALES

- Son valores, datos que se inscriben de forma directa en el código. Estos valores se suelen almacenar en direcciones de memoria, referenciadas por variables.
- **No es lo mismo, por tanto, un literal que una variable.** Una variable es una etiqueta que está vinculada a una dirección de memoria; el literal es un valor que se copiaría en esa dirección de memoria.
- Los literales pueden ser:
  - Enteros
  - Punto flotante
  - Booleanos
  - De tipo carácter
  - De tipo String (cadena de caracteres)
  - Literal null



# LITERALES NUMÉRICOS

- **Enteros.** En complemento a 2. Existe el tipo **int** (32 bits) o el tipo **long** (64 bits). Pueden representarse en decimal, octal o hexadecimal
  - Decimal: Los valores posibles son el cero (0) o un conjunto de dígitos cualquiera (el primero distinto de 0 ya que sino se consideraría **octal**)
  - Hexadecimal: Prefijo **0x**. (**0x2AB**)
  - Octal: Prefijo 0 (cero). (**07766**)
  - Si se quiere indicar que un literal es de tipo **long** se le añade un sufijo **L**. Ejemplo: **432L**
- **Reales.** Representados en **IEE754**: **float** (32 bit) o **double** (64 bit)
  - Ejemplos: **435.54**   **45e-3**   **32.45f**
  - Si se quiere indicar que un literal es de tipo **float** se le añade un sufijo **f** o **F**, en caso contrario será **double** (sufijo **d** o **D**).



# LITERALES DE TIPO CARÁCTER

- Un literal de tipo carácter puede ser un carácter simple o una secuencia de escape entrecomillada por comilla simple (''). Algunos ejemplos:

■ 'a'	Carácter
■ '%'	Carácter
■ '\t'	Secuencia de escape
■ '\\'	Secuencia de escape
■ '\"'	Secuencia de escape
■ '\u03a9'	Secuencia de escape UNICODE en hexadecimal
■ '\xFFFF'	Secuencia de escape UNICODE en hexadecimal
■ '\177'	Secuencia de escape UNICODE en Octal
■ 'Ω'	Carácter

# LITERALES TIPO STRING

- Una secuencia de cero o más caracteres entrecomillados por comillas dobles ("").

## Algunos ejemplos:

- `""` // Cadena vacía
  - `"\""` // Cadena que contiene una sola “
  - `"This is a string"` // Cadena de 16 caracteres
  - `"This is a " +  
"two-line string"` // Una expresión de tipo constante, formada por  
// dos literales de tipo String



# SECUENCIAS DE ESCAPE

- Representación de caracteres especiales

▪ \b	// \u0008: backspace BS
▪ \t	// \u0009: horizontal tab HT
▪ \n	// \u000a: linefeed LF
▪ \f	// \u000c: form feed FF
▪ \r	// \u000d: carriage return CR
▪ \"	// \u0022: double quote "
▪ \'	// \u0027: single quote '
▪ \\	// \u005c: backslash \
▪ <i>OctalEscape</i>	// \u0000 to \u00ff: from octal value



## EJERCICIO

- Crear un programa en Java que, con una sola instrucción, obtenga las siguiente salidas:
  - : El SYSTEM\_ROOT de Windows está ubicado en:  
C:\Windows\System32\
  - : En un lugar de "La Mancha" de cuyo nombre no quiero acordarme  
Vivía un 'Hidalgo'
  - :

Procesador	Fabricante
Pentium	Intel
Athlon	AMD



## OTROS LITERALES

- **Booleanos.** Existen los literales **true** y **false**

```
boolean t = true;  
boolean f = false;
```

- **Nulo.** Existe el literal **null**; indica que una referencia no tiene valor.

```
sc = null;
```



# SEPARADORES

- Java incluye los siguientes símbolos

( )

Parámetros / argumentos de función

{ }

Delimitadores de bloque

[ ]

Acceso en arrays

;

Separador de sentencias

,

Separador en listas (parámetros, inicialización...)

.

Acceso a miembros



# OPERADORES

=	>	<	!	~	?	:					
==	<=	>=	!=	&&		++	--				
+	-	*	/	&		^	%	<<	>>	>>>	
+=	-=	*=	/=	&=	=	^=	%=	<<=	>>=	>>>=	



## EJERCICIO

- Realizar un programa que lea dos variables de tipo entero y muestre el resultado de sumarlas, restarlas y dividirlas. También se desea obtener el módulo de la división entera

```
int a, b;
```

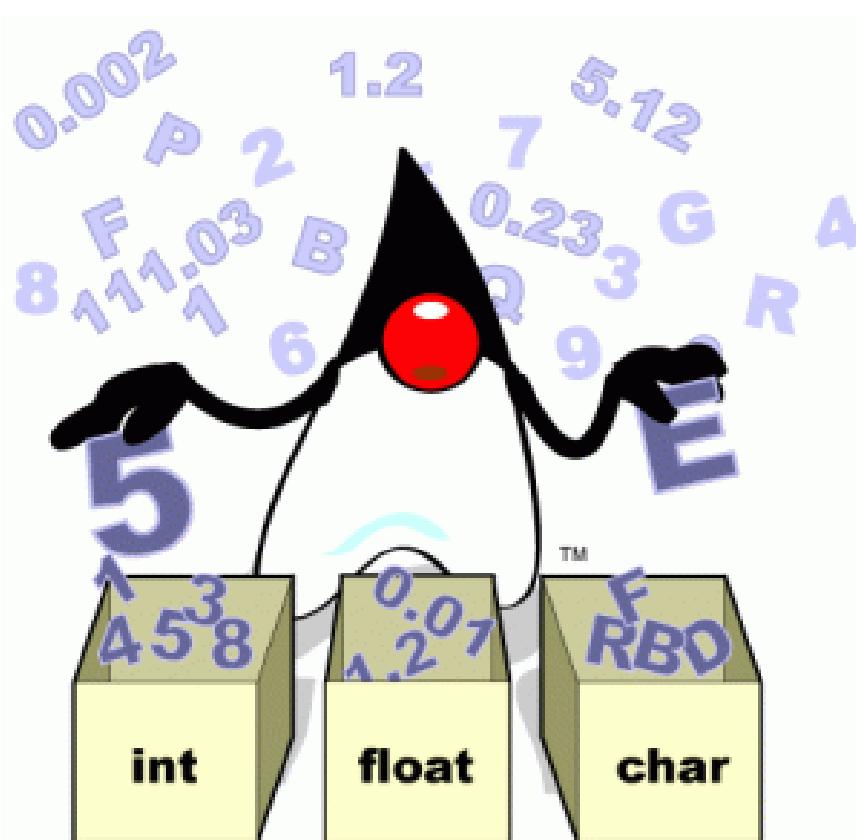
...

- Realizar un programa que declare una variable **j** con valor 255 y luego haga un AND a nivel de bits con un literal 127. ¿Cuál es el resultado obtenido? ¿Por qué?

```
int j = 255;
```

...

# TIPOS DE DATOS





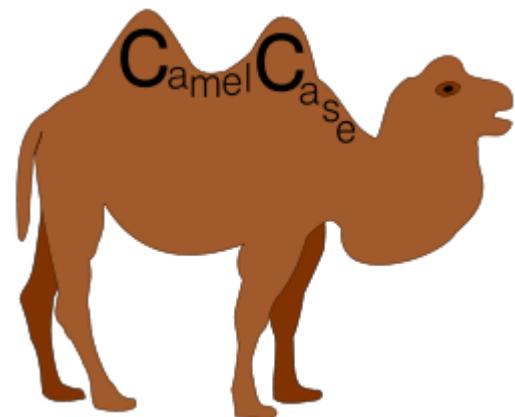
# TIPOS DE DATOS

- Java es un lenguaje **fuertemente tipado**, por lo que hay que declarar toda variable antes de usarla.
- La declaración de una variable implica especificar su tipo de datos y su nombre (identificador).
- Los tipos de datos establecen el tamaño en memoria de la variable, el rango de valores que puede almacenar, así como las operaciones soportadas. Así mismo clarifican el papel que juegan estas variables en las expresiones.
- Los tipos de Java se dividen en dos grandes grupos
  - **Tipos primitivos** (nombre en minúsculas)
  - **Tipos referencia** (nombre en Upper Camel Case)



# NOMBRADO DE ELEMENTOS

- Tipos referencia (Clases, Interfaces, etc.). **Upper Camel Case**
  - Ejemplo: **PersonaHumana**
- Variables y métodos. **Lower Camel Case**
  - Ejemplo: **incrementarSalario**
- Paquetes y tipos primitivos. Minúsculas
  - Ejemplo: **es.iespuerto.util.seguridad**



# TIPOS PRIMITIVOS

- Los tipos numéricos son los tipos enteros y los de punto flotante
- Los de tipo entero son: **byte**, **short**, **int** y **long**, cuyos valores son 8-bit, 16-bit, 32-bit y 64-bit enteros en complemento a 2, respectivamente **char**, cuyos valores son enteros de 16-bit sin signo, que representan códigos UTF-16.
- Los tipos de punto flotante son **float**, cuyos valores incluyen números IEEE 754 de 32 bits, y **double**, correspondientes a IEEE 754 de 64 bits.
- El tipo **boolean** tiene exactamente dos valores: true and false.
- Los tipos primitivos contienen un valor directo, correspondiente a un literal



```
int i = -56;
```

# TIPOS REFERENCIA

- No contienen un valor propio, sino la dirección de un objeto que sí contiene los datos. Los tipos referencia apuntan a objetos
- Inicialmente no apuntan a nada, tienen valor **null**

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

```
public static  
void main(String[] args) {  
  
    Punto p;  
    ...  
}
```

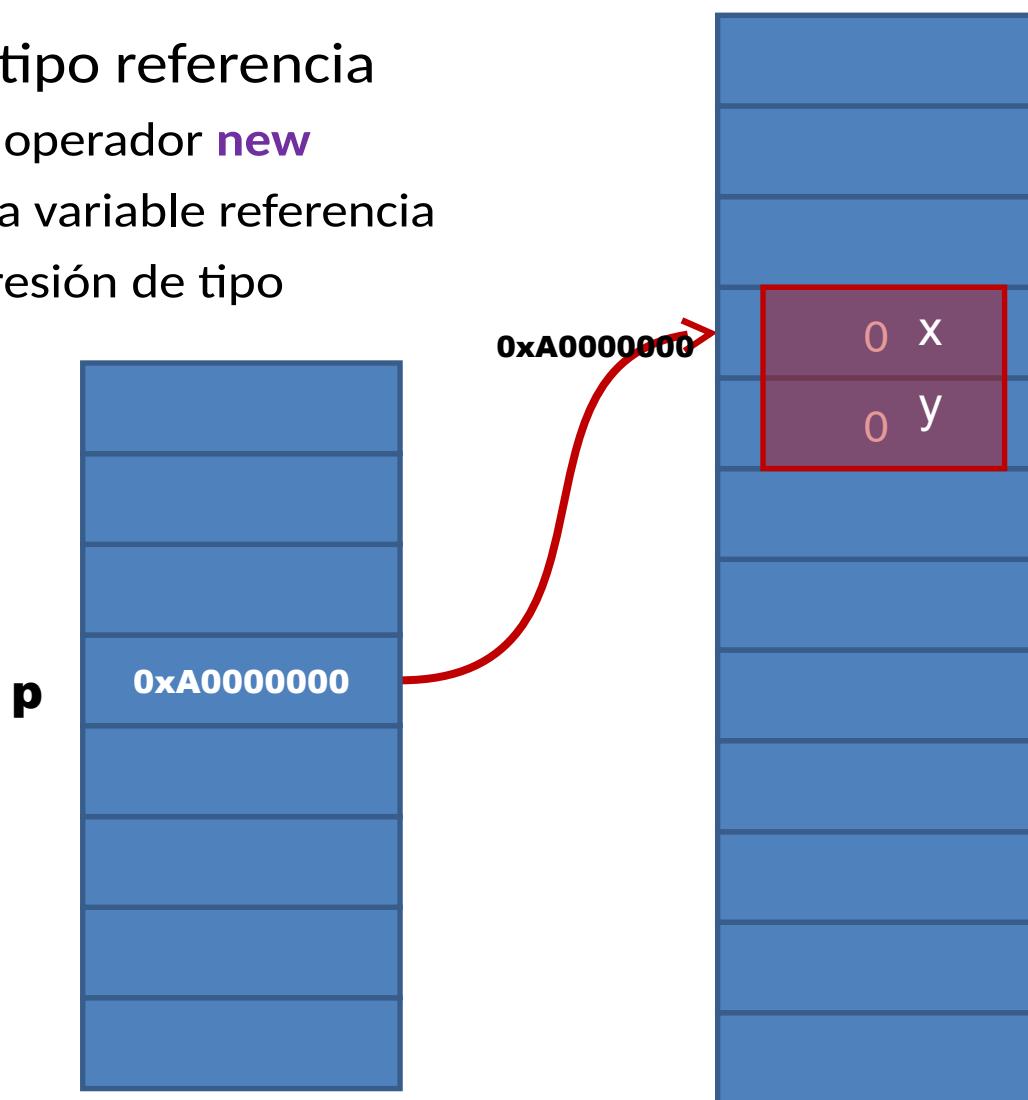


# TIPOS REFERENCIA

- Si se asigna un objeto a un tipo referencia
  - a. Asignando el resultado del operador **new**
  - b. Asignándole el valor de otra variable referencia
  - c. Asignándole cualquier expresión de tipo referencia a objeto

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

```
public static  
void main(String[] args) {  
  
    Punto p;  
    p = new Punto();  
    ...  
}
```





# TIPOS REFERENCIA Y OBJETOS

- A través de una variable de tipo referencia a un objeto, podemos acceder a cada uno de los miembros del objeto (propiedades y métodos)
- En el ejemplo anterior teníamos una variable `p` de tipo referencia, que referenciaba un objeto en memoria (una instancia de la clase Punto)
- Utilizando el operador de acceso a miembros (`.`), podemos leer o modificar los datos de cada uno de las propiedades en memoria del objeto

```
p = new Punto();  
p.x = 5;           // Modificamos el valor (LVALUE)  
p.y = 10;  
int coord = p.y; // Leemos el valor (RVALUE)
```



## EJERCICIO

- Utilizando el operador **new** para crear objetos y el operador de acceso a miembros, crear un programa (utilizando la clase Persona), que cree dos variables referencia de tipo Persona, de nombre **juan** y **marta**
- Deberá asignarse a marta los siguientes valores:
  - Nombre: Marta
  - Apellidos: Sánchez
  - Edad: 42
- A Juan
  - Nombre: Juan
  - Apellidos: Diego
  - Edad: 39
- Posteriormente se imprimirán por salida estándar los datos de ambas personas

```
public class Persona {  
    public String nombre;  
    public String apellidos;  
    public int edad;  
}
```



## NOTA SOBRE EL EJERCICIO

No es necesario importar el tipo Persona, dado que tanto la clase AppMain, como la clase Persona van a estar en el mismo directorio (veremos luego que en el mismo paquete Java)



# TIPOS ENVOLTORIO (WRAPPER) Y BOXING

- Asociados a cada tipo primitivo existen tipos referencia que envuelven a los tipos primitivos. En ocasiones nos interesa tratar a los tipos primitivos de la misma forma que a los tipos referencia y además los tipos envoltorio proporcionan una serie de operaciones útiles para nosotros.

Primitivo	Wrapper	Primitivo	Wrapper
boolean	Boolean	byte	Byte
char	Character	short	Short
int	Integer	long	Long
float	Float	double	Double



## BOXING / UNBOXING

- Conversión automática entre tipos primitivos y tipos referencia correspondientes
- Boxing

```
Integer miEntero;
```

```
miEntero = 4;
```

- Unboxing

```
int j = miEntero;
```



# CONVERSIONES A Y DESDE STRING

- Las clases que se corresponden con los tipos primitivos tienen métodos para realizar conversiones entre distintos tipos de datos.
- Estos métodos son de tipo estático (lo veremos más adelante). Al ser de tipo estático se invocan directamente desde la clase. La forma de realizar la llamada es:

**NombreClase.nombreMétodoEstático(lista\_argumentos)**

```
String s = "65536";
```

```
int i = Integer.parseInt(s); // convierte la cadena en  
// un entero
```

```
String t = String.valueOf(65535); // lo contrario
```



## EJERCICIO

- Lectura de argumentos por línea de comandos. Leer un número pasado como argumento en la llamada e imprimir en la salida el mismo número multiplicado por 100

El método **main** tiene un parámetro que nos permite recuperar argumentos pasados en la llamada.

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

El argumento args es un array (lo veremos más adelante) que almacena todos los parámetros pasados.

Los arrays de Java permiten acceder a sus elementos de manera parecida a como hacíamos en Python con las listas.

args[0] devuelve el primer elemento del array args

The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'C:\Temp>java InputMain 5'. The output shows '500' on the next line, indicating that the program has multiplied the input value by 100. The prompt 'C:\Temp>' is visible at the bottom.



## CASTING (AHORMADO)

- Una expresión de **casting** convierte, en tiempo de ejecución, el valor de un tipo numérico a otro tipo compatible.
- El cast puede ser de tipo **narrowing** (de mayor a menor rango de representación) o **widening** (lo contrario a lo anterior)
- Ejemplos:
  - int a = **(int)** 99.354; // a vale 99; **narrowing**
  - float f = **(float)** a; // f vale 99.0; **widening**
  - String s = **(String)**99;  // cast no válido, no compatible usar `valueOf`



# TIPOS REFERENCIA Y EL API DE JAVA





# API DE JAVA

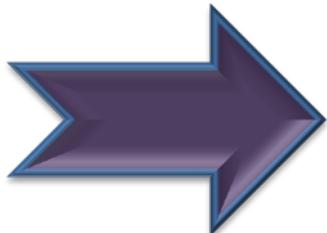
- API: Acrónimo que significa interfaz de programación de aplicaciones (*Application Program Interface*)
- Conjunto de funciones y procedimientos (métodos, en programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción
- El API de Java es el conjunto de clases, agrupadas en diferentes paquetes, que permiten extender la funcionalidad del lenguaje (lo que puede hacer éste) hacia cualquier área del desarrollo de software
- El API de la versión 6 está disponible en el siguiente enlace:  
<http://download.oracle.com/javase/> (enlace API documentation)



1. <http://download.oracle.com/javase/>

## TIPOS REFERENCIA: CLASES Y OBJETOS

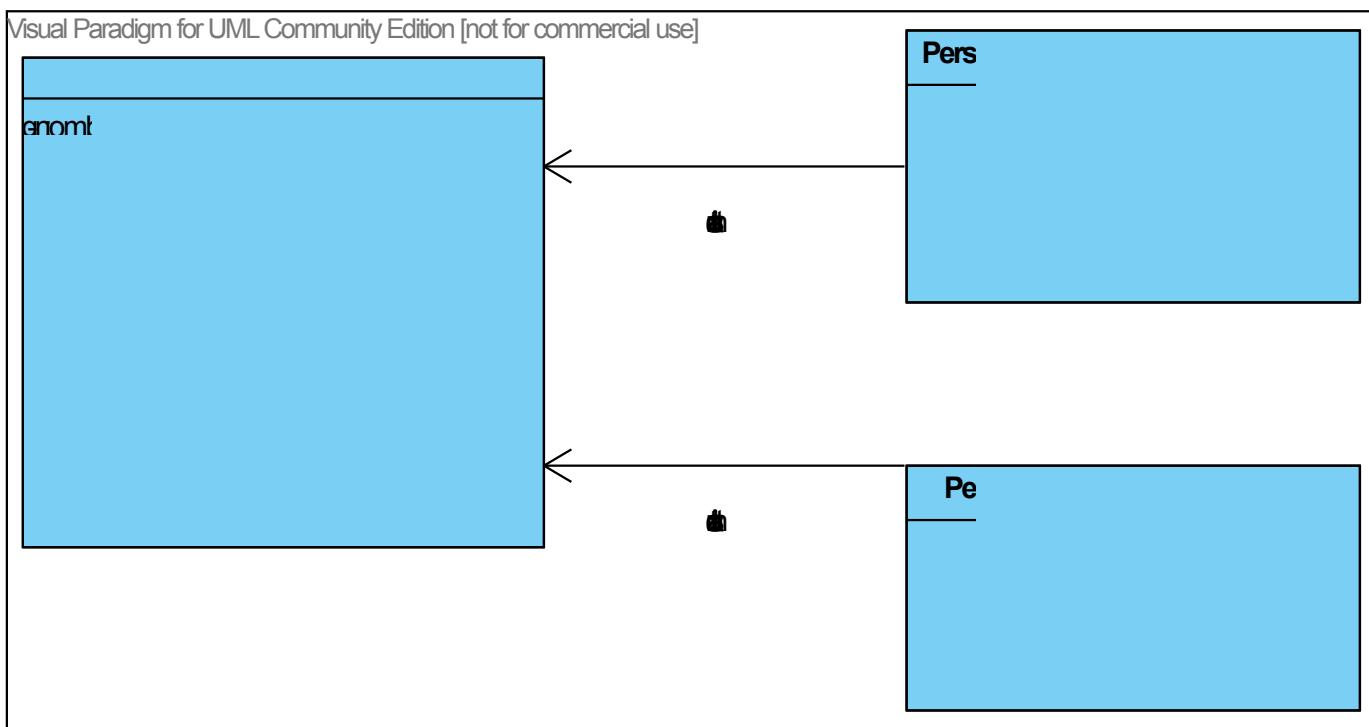
- El API de Java está formado por un elevado número de clases, organizadas en paquetes; para “usar” estas clases tenemos que ver cómo acceder a sus propiedades y métodos
- Una **clase** es como el molde a partir del cual se crean nuevos elementos (los **objetos**)



- A nivel de lenguaje de programación define una especie de agrupación de variables y métodos que luego todo objeto va a tener

# CLASES Y OBJETOS

- En el ejemplo definimos una clase Persona, diciendo que todo objeto que se cree luego va a tener un nombre, unos apellidos, ...
- Luego se crearán objetos a partir de esa clase y todos van a llevar un nombre, unos apellidos... cada uno con unos valores concretos y propios de cada objeto





# COMPOSER TEXTOS

- Java dispone de varias formas de componer largas cadenas de texto
- **Operador de concatenación (+)**
  - Literales cadena: “Hola” + “ Mundo” → “Hola Mundo”
  - Literal cadena y entero: “3” + 1 → “31”
  - Literal cadena y real: “3” + 1.0 → “31.0”
  - Literal cadena y boolean: “3” + true → “3true”
- **Función concat** (sólo admite expresiones tipo String)
  - String s = “hola”;  
String t = “ mundo”;  
s.concat(t) → “hola mundo” // el objeto al que apunta s no se modifica
  - s.concat(“ mundo”); → “hola mundo” // el objeto al que apunta s no se modifica
  - “Hola”.concat(“ Mundo”); // mecanismo de boxing



## OBJETOS DE CLASE STRINGBUFFER

- El uso del operador (+) es muy ineficiente cuando tenemos que ir componiendo una cadena de texto en base a múltiples trozos (literales, valores de variables, etc.)
- Cada vez que se resuelve la expresión se crea un nuevo objeto String en memoria, si este objeto es sólo un intermediario para unir otro trozo de la cadena estaríamos desperdiciando memoria
- Por ello es mucho más eficiente usar objetos de la clase **StringBuffer**. Un objeto de StringBuffer es una especie de String, pero que es mutable: modificable. Podemos ir añadiendo trozos de la cadena sin la sobrecarga que hemos explicado del operador (+)



## EJEMPLO DE USO DE STRINGBUFFER

- En primer lugar tenemos que crear (instanciar) un nuevo objeto de la clase **StringBuffer**
  - `StringBuffer sb = new StringBuffer();`
- A continuación añadimos fragmentos con el método **append**
  - `sb.append("Trozo de cadena");`
- Para obtener el resultado final llamamos al método **toString**

```
// Queremos imprimir lo siguiente: Resultado de a(4) + b(3) = 7
int a = 4;
int b = 3;
StringBuffer sb = new StringBuffer("Resultado de a(");
sb.append(a);
sb.append(") + b(");
sb.append(b);
sb.append(") = ");
sb.append(a+b);
System.out.println(sb.toString());
```



## EJERCICIO

- Programa que reciba un argumento por línea de comandos, cuyo valor sea una expresión que indique una variable de entorno (**según convenio UNIX**), y muestre por la consola su valor correspondiente.
- Para realizar el programa es preciso consultar la documentación del API de Java en relación a los métodos necesarios de la clase **String** relacionados con el tamaño de la cadena (**length**) y la extracción de subcadenas a partir de la misma (**substring**).
- También puede ser conveniente consultar los métodos para concatenación de cadenas (**concat**), aunque puede usarse el operador (+) para concatenar literales o variables de tipo cadena
- Finalmente será necesario consultar la clase **System** para ver qué método nos proporciona información sobre una variable de entorno.
- Ejemplo de uso:

C:\>java EchoApp \$Path

Salida

Valor de Path= C:/dai/prj/java/install/jre6/bin/client;



# PROPIEDADES Y MÉTODOS DE CLASE Y DE INSTANCIA

- Para utilizar los métodos de una clase tenemos que saber que pueden ser de dos tipos: métodos de **clase** y métodos de **instanci**a
- Los métodos de instancia trabajan con un objeto concreto, mientras que los de clase son independientes de los objetos, no necesitan la información contenida en cada uno de los objetos.
- Métodos y propiedades de **instanci**a (ejemplo con clase Persona)
  - Trabajan sobre un objeto de la clase, se accede a través de la variable referencia

```
Persona p = new Persona("Rafael", "Nadal", 29);
p.crece(1);           // crece 1 año
System.out.println(p.getEdad());
```

- Métodos y propiedades de **clase** (ejemplo con clase Persona)
  - Se invocan utilizando el nombre de la clase, no precisan que haya ningún objeto creado

```
Persona.getPlaneta(); // devuelve "Planeta Tierra"
```



# EJEMPLO MÉTODOS Y PROPIEDADES DE INSTANCIA VS. CLASE

- Ejemplo: clase **Integer**
  - Tiene una propiedad de clase que es **MAX\_VALUE**, esta propiedad tiene como valor el máximo entero positivo representable con este tipo de datos en Java

```
int i = Integer.MAX_VALUE; // igual para todos los enteros
```

- Tiene un método de instancia que es

```
Integer miEntero = new Integer(5); // hay que crear
// el objeto
// antes de usarlo
```

```
int a = miEntero.intValue();           // devuelve el
// valor de miEntero
```



## EJERCICIO

- Realizar un programa que muestre el rango de representación de los tipos de datos numéricos más comunes (**int**, **long**, **float** y **double**)
- El programa deberá mostrar el valor mínimo y máximo representable con los tipos de datos anteriores.
- Asimismo deberá indicar el número de bits usados en los tipos de datos **int** y **long**
- Salida deseada

Tipo	Rango
int	[-2147483648, 2147483647]
long	[-9223372036854775808, 9223372036854775807]
float	[1.4E-45, 3.4028235E38]
double	[4.9E-324, 1.7976931348623157E308]

Tipo	Bits
int	32
long	64



## EJERCICIO: NOTAS

- Para calcular el número de bits necesario para representar un valor numérico se utiliza el logaritmo en base 2.
  - Imaginemos Por ejemplo  $\log_2(64)=5.97\dots$  lo cual indica que para representar 64 valores distintos en binario necesitaríamos al menos 6 bits.
- En Java no existe ningún método que proporcione logaritmos en base 2; no obstante sabiendo que la fórmula para el cambio de base es:
$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$
podemos calcular el logaritmo en cualquier base utilizando logaritmos decimales (base 10)
  - Ejemplo:
$$\log_2(1024) = \frac{\log_{10}(1024)}{\log_{10}(2)} = 10$$
- En Java la función para obtener el logaritmo decimal de un número es el método estático **log10** de la clase **Math**

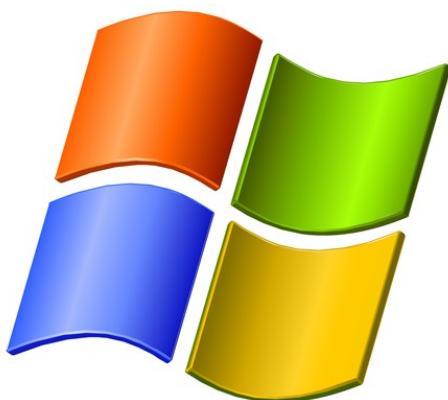


## EJERCICIO: NOTAS

- Existen funciones de redondeo en Java, disponibles en la clase **Math**; buscar la más conveniente para el problema a resolver
- Utilizando el operador de ahormado (**casting**) podemos convertir de un tipo de datos a otro. Por ejemplo podemos convertir de un double a un entero
  - Ejemplo: int miDatos = **(int)** 54.35;
- Para componer los mensajes a mostrar en pantalla pueden utilizarse tanto la función **concat** de la clase **String**, como el operador (+) aplicado a variables y literales de tipo texto.

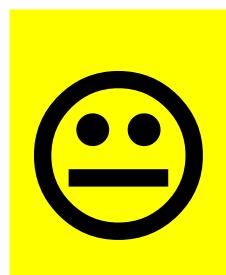


# GENERACIÓN DE EJECUTABLES



## ¿GENERACIÓN DE EJECUTABLES?

- Java no es un lenguaje compilado; por tanto no genera un ejecutable que pueda ser lanzado directamente por el sistema operativo  
Necesita una instancia activa de la máquina virtual (**java**)
- Podemos empaquetar la aplicación en archivos **.jar** e indicar en el manifiesto el punto de entrada a la aplicación
- Existen herramientas que permiten generar un envoltorio (*wrapper*) que permita ver el **.jar** como un ejecutable





# EL CLASSPATH

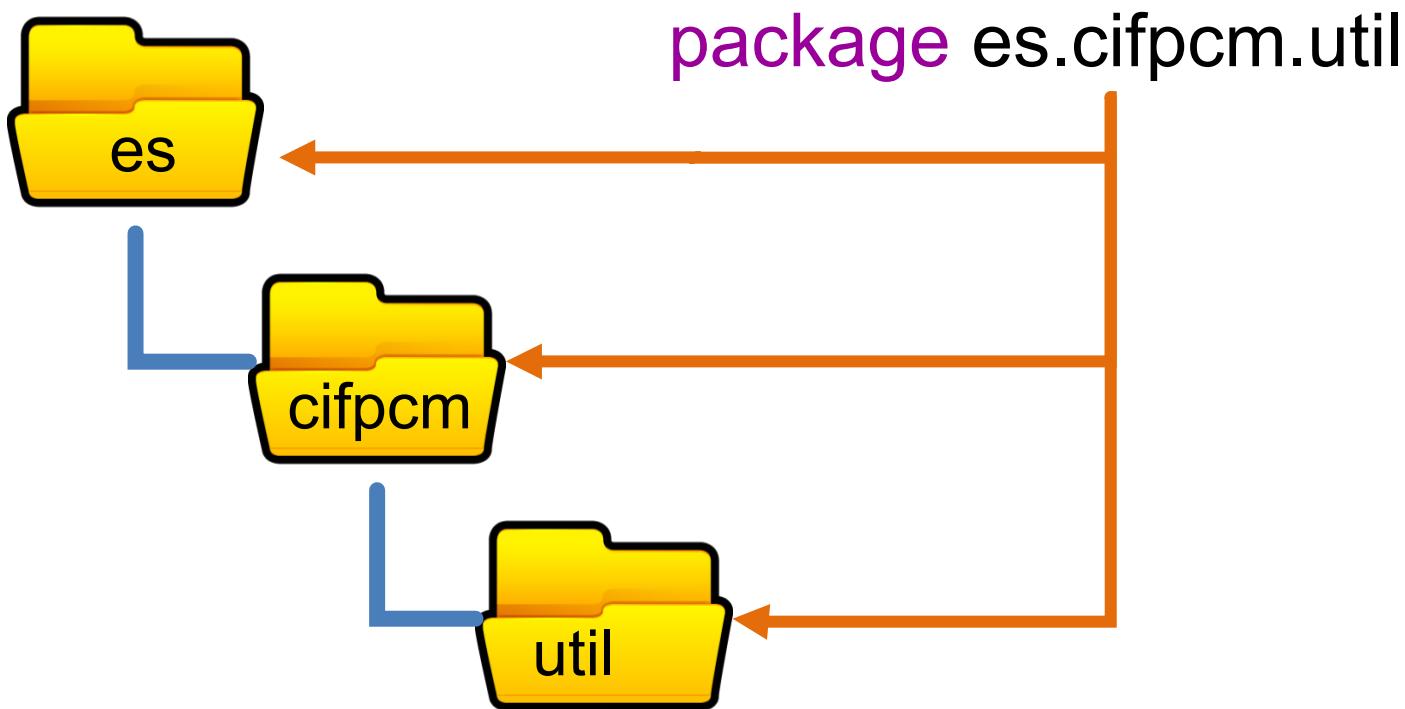
- A la hora de ejecutar un programa, la máquina virtual necesita acceso a todas las clases que componen el mismo
- A diferencia de la compilación estática que realizamos con el *linker*, Java carga las clases dinámicamente, por lo que puede ser que en el momento de la ejecución éstas no estén disponibles
- Cuando se inicia la máquina virtual, ésta busca las clases en:
  1. **Bootstrap classes** – Clases que conforman la plataforma Java, incluye las clases en rt.jar y otros archivos .jar claves en la plataforma
  2. **Extension classes** - Clases que usan el mecanismo de extensión de Java. Se encuentran empaquetadas en archivos .jar en el directorio de extensión.
  3. **User classes** - Clases creadas por desarrolladores y terceras partes (*third parties*) que no utilizan el mecanismo de extensión. Su ubicación se indica a través de la opción de línea de comando **-classpath** (preferido) o usando la variable de entorno **CLASSPATH**



1. <http://www.chuidiang.com/java/classpath/classpath.php>
2. <http://download.oracle.com/javase/6/docs/technotes/tools/findingclasses.html>

# ORGANIZACIÓN DE LAS CLASES EN PAQUETES

- Las clases Java se organizan lógicamente en paquetes.
- Físicamente los paquetes se corresponden con carpetas en el sistema de archivos o en la organización interna del archivo .jar
- Por ejemplo el paquete siguiente se corresponde con la estructura de carpetas mostrada





# EMPAQUETADO DE CLASES EN ARCHIVOS .JAR

- El formato de archivos Java™ Archive (JAR) permite empaquetar múltiples archivos en un único archivo comprimido. Un fichero JAR contiene las clases y recursos auxiliares asociados a aplicaciones y applets
- Este formato reporta los siguientes beneficios
  - **Seguridad:** El contenido de un JAR puede firmarse digitalmente.
  - **Mejora en el tiempo de descarga:** Si un applet está empaquetado como JAR puede descargarse en una única transacción HTTP
  - **Compresión:** El formato JAR admite compresión, ocupando menos espacio de almacenamiento
  - **Extensiones:** El framework de extensión proporciona métodos para extender la funcionalidad de la plataforma base de Java, utilizando el formato JAR
  - **Precintado:** Los paquetes contenidos en un JAR pueden ser precintados de forma opcional. Esto significa que todas las clases de un paquete precintado deben estar contenidas en el mismo JAR
  - **Versionado:** Un fichero JAR puede contener datos acerca de los ficheros que contiene, como el proveedor e información de versión
  - **Portabilidad:** El mecanismo de gestión de los JAR es parte del API de Java



# HERRAMIENTA JAR

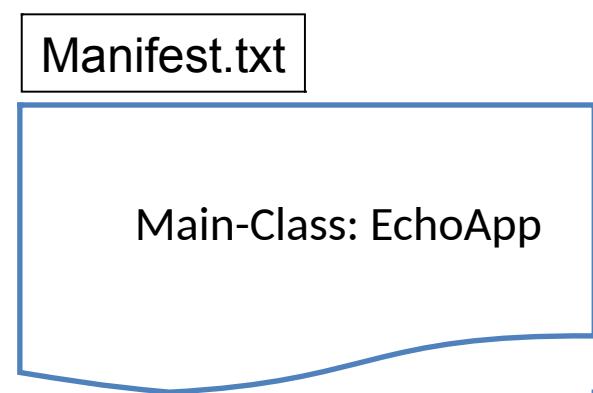
- Los ficheros JAR se empaquetan en formato ZIP
- Para realizar las tareas básicas con ficheros JAR (creación, extracción, ejecución...) se utiliza la herramienta **Java™ Archive** proporcionada como parte del Java Development Kit (**JDK**).
- La herramienta Java Archive se invoca mediante el comando **jar**

Operación	Comando
Crear un archivo JAR	<code>jar cf jar-file input-file(s)</code>
Visionar un archivo JAR	<code>jar tf jar-file</code>
Extraer el contenido de un archivo JAR	<code>jar xf jar-file</code>
Extraer ficheros específicos	<code>jar xf jar-file archived-file(s)</code>
Ejecutar una aplicación empaquetada como JAR (requiere la cabecera Main-class del manifiesto)	<code>java -jar app.jar</code>
Invocar un applet empaquetado como JAR	<code>&lt;applet code=AppletClassName.class archive="JarFileName.jar" width=width height=height&gt; &lt;/applet&gt;</code>



# GENERACIÓN DE UN JAR CON ARCHIVO DE MANIFIESTO

- Java admite un archivo de manifiesto opcional, que permite añadir información al archivo JAR
- El nombre del archivo de manifiesto se incluye como argumento en la llamada a la herramienta de línea de comando **jar**  
**jar cfm EchoApp.jar Manifest.txt EchoApp.class**
- Ejemplo de archivo de manifiesto (**meter espacio en blanco al final**)



1. <http://download.oracle.com/javase/tutorial/deployment/jar/manifestindex.html>



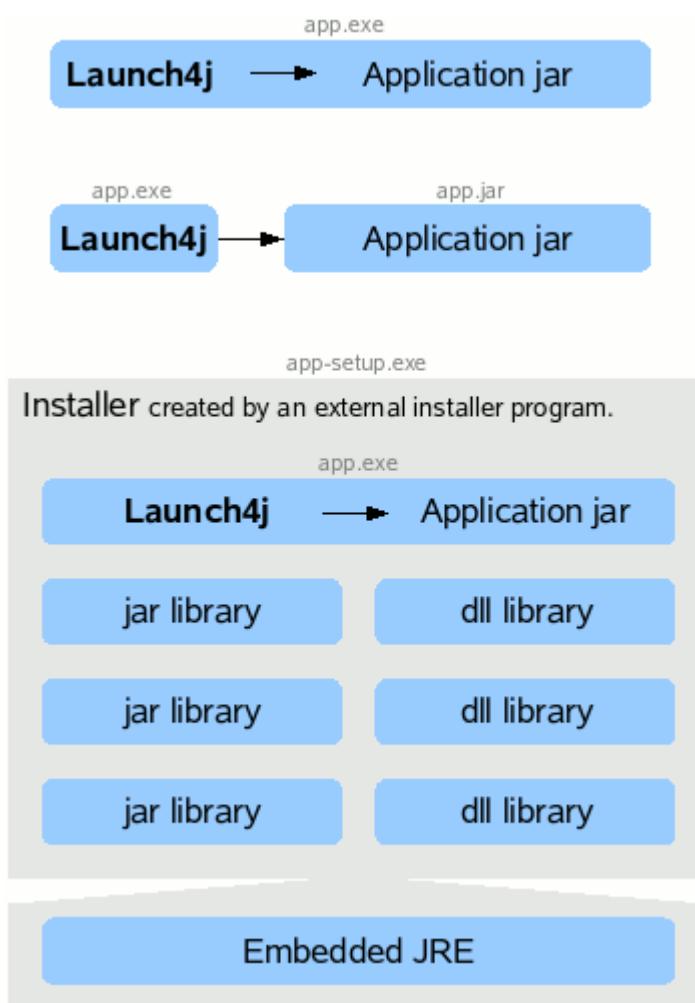
## EJERCICIO

- Empaquetar como JAR el ejercicio de la diapositiva 31 (archivos PersonaApp.class y Persona.class)
- Crear un archivo de manifiesto que indique que la clase principal es PersonaApp.class
- Ejecutar el archivo **.jar** resultante con el argumento adecuado en la invocación de la máquina virtual (**java**)
- Crear un archivo de texto con el nombre **ejecucionjar.txt**, en el que se detallen todos los pasos seguidos y se copien las instrucciones dadas a través del intérprete de comandos

# HERRAMIENTAS PARA GENERAR WRAPPER

- Herramienta **launch4j**

**launch4j**  
3.0.2





# PREGUNTAS

