

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: Could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, h=self.sck.accept()
                self.todo=2
                self.handle=h
                self.start()
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

# DAM/DAW

# PROGRAMACIÓN

## 03.2

# Introducción a las clases en Java

# Indice

- [Las clases en Java](#)
- [Creación de clases y objetos](#)
- [Los métodos](#)
- [Constructores](#)
- [Destrucción de objetos](#)
- [La referencia \*this\*](#)
- [Los métodos \*get\* y \*set\*](#)
- [Métodos estáticos](#)
- [La clase \*String\*](#)
- [Combinando llamadas a métodos](#)
- [Impresión de objetos: el método \*toString\(\)\*](#)
- [Paquetes de clases](#)

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sock.connect((addr,port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
    def listen(self,host,port,func):
        try:
            self.sock.bind((host,port))
            self.sock.listen(2)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error:Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x,ho=self.sock.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
            if (self.handle.isclosed()):
                self.handle.send(data)
            self.close(self)
        self.sock.close()
```

# Las clases en Java

---

- Antes de continuar el estudio de otras estructuras de la programación, como las referentes al control del flujo de nuestros programas, necesitamos hacer un pequeño alto y comprender los conceptos básicos de la **creación de clases** en Java
- Las clases definen la naturaleza de los objetos y son la base sobre la que se levanta Java, como lenguaje OOP que es.
- Dentro de nuestras clases definiremos, tanto los datos almacenados por sus objetos (**atributos**), como el código (**métodos**) que actuará sobre ellos
- Debemos “entender” las clases como las **plantillas** o **planos** necesarios para la **creación de objetos**. Cada nueva clase que definamos, se convertirá en un nuevo **tipo** que podremos usar para declarar variables y crear objetos (**instancias**) de esas clases. Esto es lo que hace que Java sea considerado un lenguaje **extensible**

## Creación de Clases y Objetos (I)

- En general, la creación de una nueva clase supone: indicar el **nombre** de la clase, declarar sus **atributos** y declarar e implementar sus **métodos**
- La forma *simplificada* de creación de una clase (**sin** *modificadores de acceso, herencia e implementación de interfaces*) es:

```
class NombreClase {  
    // declaración de atributos (variables de instancia)  
    tipo var1;  
    tipo var2;  
    // ...  
    tipo varN;  
  
    // declaración de métodos  
    tipo método1(lista_parámetros) {  
        // cuerpo del método 1  
    }  
    tipo método1(lista_parámetros) {  
        // cuerpo del método 2  
    }  
    // ...  
    tipo métodoN(lista_parámetros) {  
        // cuerpo del método N  
    }  
}
```

## Creación de Clases y Objetos (II)

- Para ilustrar el concepto, vamos a crear una pequeña clase para *encapsular* información sobre vehículos, como coches, camiones,...
- La clase se denominará **Vehiculo** y almacenará información referente a *número de pasajeros*, *capacidad del depósito* y *consumo* (litros a los 100km)
- De la *especificación* anterior es fácil *extraer* cuáles serán los principales atributos de nuestra clase. Una vez decididos de qué tipo serán, podemos crear una primera versión de nuestra nueva clase (la guardaremos en un archivo denominado *Vehiculo.java*):

```
//: Vehiculo.java

class Vehiculo {
    // declaración de atributos (variables de instancia)
    int numPasajeros;      // número de pasajeros
    int capDeposito;       // capacidad del depósito (litros)
    double consumo100KM;   // consumo (litros a los 100KM)
}
```

## Creación de Clases y Objetos (III)

- La definición de la clase anterior creará un nuevo **tipo de datos** denominado *Vehiculo*. Es decir, la declaración de la clase es **sólo** la descripción de un tipo. Por sí misma, **no crea** objetos *Vehiculo*
- Para **crear objetos** de la clase *Vehiculo* en nuestro programa, deberíamos hacer dos cosas:
  - Declarar una nueva variable del tipo *Vehiculo*
  - Usar el operador **new** para crear un objeto de la clase *Vehiculo* y **asignarlo** a la variable anterior

```
Vehiculo miCoche; // miCoche es una variable del tipo Vehiculo  
miCoche = new Vehiculo(); // se crea el nuevo objeto y se asigna
```

- Alternativamente, podríamos combinar ambas sentencias anteriores en una sola:

```
Vehiculo miCoche = new Vehiculo();
```

## Creación de Clases y Objetos (IV)

- En la sentencia anterior:

```
Vehiculo miCoche = new Vehiculo( )
```

la llamada al operador *new*, provoca que se *solicite* de la JVM la creación de un *nuevo* objeto del tipo *Vehiculo*.

- La JVM, a partir de la definición de la clase *Vehiculo*, que usará a modo de *plantilla*, asignará los recursos necesarios por el nuevo objeto e invocará al *método constructor* de la clase para inicializar sus atributos (fíjate en los paréntesis que acompañan a *Vehiculo*; indican una llamada a un método).
- Una vez *creado e incializado* el objeto, la JVM devolverá una *referencia* (*apuntador*) al nuevo objeto creado en memoria. Esta referencia es la que se *asignará* a la variable *miCoche*. Es decir, la variable *miCoche* no almacena el objeto en sí, sino una especie de *dirección de memoria* que le permite a la JVM localizar el objeto creado cada vez que usemos dicha variable



## Creación de Clases y Objetos (V)

- Cada vez que creamos una nueva instancia de una clase, estamos creando un objeto que contiene sus **propias** copias de cada variable de instancia. Es decir, cada objeto de tipo *Vehiculo* tendrá sus variables de instancia ***numPasajeros***, ***capDeposito*** y ***consumo100KM***
- Para acceder a dichas variables, usaremos el **operador punto** (.)
- Este operador **enlaza** el nombre de un objeto con el nombre de uno de sus miembros (atributos y métodos). La sintaxis general del operador (.) es:

*objeto.miembro*

- Por ejemplo, si quisiéramos asignar el valor 5 a la variable ***numPasajeros*** del objeto ***miCoche***, haríamos:

*miCoche.numPasajeros* = 5;

- Si bien el **operador** (.) nos permite acceder a cualquier **miembro** del objeto, veremos más adelante como aplicar **modificadores de acceso** para limitarlo



### Creación de Clases y Objetos (VI)

---

- Vamos probar nuestra nueva clase *Vehiculo* pero, tenemos un problema...
- Nuestra clase no incluye el método *main* que, como sabemos, es el punto de inicio de cualquier aplicación Java (de hecho, lo habitual, es que las clases no tengan dicho método). Entonces, ¿cómo la “ejecutamos”?
- Generalmente, una aplicación Java estará formada por un conjunto de clases (cada una en su correspondiente archivo) que representan las diferentes “entidades” con las que vamos a trabajar (en nuestro caso, vehículos). Entre ellas, habrá *una* cuya función principal será la de iniciar la ejecución de nuestra aplicación y, por tanto, la que contendrá el método *main*
- Definiremos una nueva clase, *VehiculoDemo*, desde la que iniciaremos la aplicación y crearemos objetos de tipo *Vehiculo* (la guardaremos en el archivo *VehiculoDemo.java*, en la misma carpeta que *Vehiculo.java*)

# Creación de Clases y Objetos (VII)

```
class VehiculoDemo {  
    public static void main(String[] args) {  
        // Creamos dos nuevas instancias de la clase Vehiculo  
        Vehiculo miCoche = new Vehiculo();  
        Vehiculo furgoPepe = new Vehiculo();  
  
        // asignamos valores a los atributos de miCoche  
        miCoche.numPasajeros = 5;  
        miCoche.capDeposito = 45;  
        miCoche.consumo100KM = 6.8;  
  
        // asignamos valores a los atributos de furgoPepe  
        furgoPepe.numPasajeros = 7;  
        furgoPepe.capDeposito = 85;  
        furgoPepe.consumo100KM = 7.25;  
  
        // calculamos la distancia máxima que puede recorrer  
        double dist1 = miCoche.capDeposito * 100.0 / miCoche.consumo100KM;  
        double dist2 = furgoPepe.capDeposito * 100.0 / furgoPepe.consumo100KM;  
  
        System.out.printf("miCoche puede llevar %d ", miCoche.numPasajeros);  
        System.out.printf("personas hasta %.2f kms%n", dist1);  
        System.out.printf("furgoPepe puede llevar %d ", furgoPepe.numPasajeros);  
        System.out.printf("personas hasta %.2f kms%n", dist2);  
    }  
}
```

## Creación de Clases y Objetos (y VII)

Compilamos la nueva aplicación desde el directorio donde están las clases:

```
bowman@hal:~/work/src/java$ javac Vehiculo.java VehiculoDemo.java
```

(o simplemente...)

```
bowman@hal:~/work/src/java$ javac *.java
```

y ejecutamos la clase que contiene el método *main()*

```
bowman@hal:~/work/src/java$ java VehiculoDemo
miCoche puede llevar 5 personas hasta 661,76 kms
furgoPepe puede llevar 7 personas hasta 1172,41 kms
```

Observa que los “*datos*” del objeto *miCoche* son completamente independientes de los del objeto *furgoPepe*

miCoche →

numPasajeros	5
capDeposito	45
consumo100KM	6.8

furgoPepe →

numPasajeros	7
capDeposito	85
consumo100KM	7.25

### Los métodos (I)

- Aunque podemos definir clases que sólo contengan *variables de instancia*, normalmente dispondrán de varios **métodos**. De hecho, lo habitual será que la “*interacción*” con los objetos se realice a través de ellos
- Los métodos son **subrutinas** que manipulan los datos definidos por la clase y, en muchos casos, proporcionan el acceso a dichos datos para su lectura (**getters**) o modificación (**setters**)
- Cada método, identificado por un **nombre**, agrupará una serie de sentencias para la realización de algún tipo de acción o función. Así, cada vez que queramos volver a ejecutar dicha acción, no tendremos más que invocar por su nombre el método correspondiente.
- En general, los métodos devolverán algún valor con el resultado de su ejecución. Por ejemplo: *nextInt()* de *Scanner* devuelve un número entero
- Cada método definirá un lista de argumentos de modo que se pueda parametrizar cada una de sus ejecuciones. Por ej: *printf( argumentos )*

## Los métodos (II)

- La forma *simplificada* de declaración de un método de una clase (*sin modificadores de acceso o comportamiento*) es:

```
tipo método1(lista_parámetros) {  
    // cuerpo del método  
}
```

donde,

- *tipo*, es el tipo de datos del valor devuelto por el método.  
Si un método no devuelve ningún valor, se usará la palabra reservada *void* para indicarlo
- *lista\_parámetros*, es una secuencia de parejas *tipo nombre*, separadas por comas. Esencialmente, son variables que se crean automáticamente en el momento de la llamada al método y recogen el valor de los *argumentos* con los que se le haya llamado. Estas variables sólo son “visibles” dentro del propio método

### Los métodos (III)

- Volviendo con nuestro ejemplo anterior, vamos a “*mover*” el código que nos permite calcular la distancia máxima que puede recorrer el vehículo sin repostar a un método de la propia clase *Vehiculo*
- Es decir, vamos a hacer “*responsables*” a los objetos de la clase *Vehiculo* de realizar el cálculo correspondiente a partir de sus propios datos, “*librándonos*” a nosotros, “*usuarios*” de dicha clase, de tener que hacer el cálculo (ni siquiera saber cómo se hace, sólo nos interesa el resultado)
- Vamos a añadir a la clase *Vehiculo* un método llamado *autonomia()*. Dicho método será el encargado de calcular la distancia máxima que puede recorrer el vehículo y mostrarlo en pantalla. Este método no devolverá ningún valor, así que lo indicaremos con la palabra *void*
- Tampoco recibirá ningún argumento (toda la información que necesita está en el propio objeto). Aún así, fíjate como, tanto en la declaración como en la llamada, usaremos *paréntesis* (aunque estén vacíos)

## Los métodos (IV)

```
//: Vehiculo.java
class Vehiculo {
    // declaración de atributos (variables de instancia)
    int numPasajeros;        // número de pasajeros
    int capDeposito;         // capacidad del depósito (litros)
    double consumo100KM;     // consumo (listros a los 100KM)

    // declaración de métodos
    void autonomia() {
        double dist = capDeposito * 100.0 / consumo100KM;
        System.out.printf("La autonomía es de %.2f%n", dist);
    }
}
```

- Fíjate como, en el método, nos referimos directamente a las *variables de instancia*, sin precederlas del nombre del objeto y el operador *punto*
- Ahora, nuestra clase *VehiculoDemo*, “*usuaría*” de *Vehiculo*, ya no tendrá que hacer ningún cálculo para saber qué distancia puede recorrer cada vehículo. Simplemente, invocará al método de cada uno de los objetos cuando necesite imprimir dicha información



### Los métodos (V)

```
//: VehiculoDemo.java
class VehiculoDemo {
    public static void main(String[] args) {
        // Creamos dos nuevas instancias de la clase Vehiculo
        Vehiculo miCoche = new Vehiculo();
        Vehiculo furgoPepe = new Vehiculo();

        // asignamos valores a los atributos de miCoche
        miCoche.numPasajeros = 5;
        miCoche.capDeposito = 45;
        miCoche.consumo100KM = 6.8;

        // asignamos valores a los atributos de furgoPepe
        furgoPepe.numPasajeros = 7;
        furgoPepe.capDeposito = 85;
        furgoPepe.consumo100KM = 7.25;

        System.out.print("miCoche puede llevar " + miCoche.numPasajeros + " per. ");
        miCoche.autonomia();

        System.out.print("furgoPepe puede llevar " + furgoPepe.numPasajeros + " per. ");
        miCoche.autonomia();
    }
}
```

## Los métodos (VI)

### ❑ Retornando de un método

- Cada vez que Java se encuentra con una llamada a un método, se produce un cambio en el flujo de ejecución “normal” del programa, esto es, la ejecución de sentencias consecutivas.
- La llamada al método provoca que la ejecución del programa “salte” a la primera de las sentencias del método invocado. Una vez finalizada la ejecución del método, la ejecución del programa continuará por la siguiente sentencia a la de la llamada

```
class VehiculoDemo {  
    // ...  
    miCoche.autonomia();  
  
    System.out.print("furgoPepe puede llevar " +  
        furgoPepe.numPasajeros + ". ");  
}
```

```
class Vehiculo {  
    // ...  
    void autonomia() {  
        double dist =  
            capDeposito*100.0/consumo100KM;  
  
        System.out.printf("La autonomía es  
            de %.2f%n", dist);  
    }  
}
```

3

1

2

### Los métodos (VII)

---

- Hay dos condiciones que pueden provocar el **retorno** de un método:
  - Se alcanza el final del método (como en nuestro ejemplo)
  - Se ejecuta una sentencia **return**
- Si la declaración del método indica que devuelve un valor (no *void*), la sentencia *return* irá **siempre** acompañada por un *literal* o *expresión* del tipo indicado en la declaración, y éste será el valor *retornado* por dicho método
- De hecho, vamos a modificar nuestro método por una nueva versión en la que, en lugar de imprimir la autonomía, nos devuelva su valor. Con esto logramos dos cosas:
  - **Independizar** la clase *Vehiculo* de la salida estándar
  - **Disponer** del valor de la autonomía, con lo que podemos emplearlo en otros cálculos o mostrarlo de formas diversas

# Los métodos (VIII)

```
//: Vehiculo.java
class Vehiculo {
    // declaración de atributos (variables de instancia)
    int numPasajeros;        // número de pasajeros
    int capDeposito;         // capacidad del depósito (litros)
    double consumo100KM;     // consumo (listros a los 100KM)

    // declaración de métodos
    double autonomia() {
        return capDeposito * 100.0 / consumo100KM;
    }
}
```

```
//: VehiculoDemo.java
class VehiculoDemo {
    public static void main(String[] args) {
        // . . .

        System.out.print("miCoche puede llevar " + miCoche.numPasajeros);
        System.out.printf(" personas hasta %.2f kms%n", miCoche.autonomia());

        System.out.print("furgoPepe puede llevar " + furgoPepe.numPasajeros);
        System.out.printf(" personas hasta %.2f kms%n", furgoPepe.autonomia());
    }
}
```

## Los métodos (IX)

### ❑ Paso de argumentos

- Es posible pasar uno o más **valores** (*argumentos*) en la llamada al método. La **variable** del método que recibe el valor se denomina *parámetro*
- Los *parámetros* se **declaran** en la definición del método dentro de los paréntesis que siguen a su nombre. La sintaxis es la misma que la empleada para la declaración de variables. Por ejemplo:  

```
double calculaAreaTriangulo(double base, double altura)
```
- La “**visibilidad**” de los *parámetros* está limitada al **ámbito** del método, es decir, no son “*accesibles*” desde fuera del cuerpo del mismo (diferentes métodos pueden emplear los mismos nombres de parámetros, pues son variables independientes).
- La única diferencia de un *parámetro* con un variable “*normal*”, es que ya se **inicializa** con el argumento pasado en la llamada al método

## Los métodos (X)

- Vamos a añadir un nuevo método a la clase *Vehiculo* que nos permita calcular la cantidad de combustible necesaria para recorrer una distancia

```
//: Vehiculo.java
class Vehiculo {
    // ...
    double litrosDist(int kms) {
        return kms * consumo100KM / 100;
    }
}
```

```
//: VehiculoDemo.java
class VehiculoDemo {
    public static void main(String[] args) {
        final int DIST_SCQ_MAD = 600;
        // . . .
        System.out.println("Para ir desde Santiago a Madrid (" + DIST_SCQ_MAD + " km):");
        System.out.printf("miCoche necesita: %.2f litros %n", miCoche.litrosDist(DIST_SCQ_MAD));
        System.out.printf("furgoPepe necesita: %.2f litros %n", furgoPepe.litrosDist(DIST_SCQ_MAD));
    }
}
```

- Cada vez que llamemos al nuevo método *litrosDist* tendremos que pasarle un *argumento* del tipo especificado en la declaración, o se producirá un **error de compilación** (en este caso, pasamos la constante *DIST\_SCQ\_MAD*)

## Los métodos (y XI)

### ❑ Métodos y Alcance de las variables

- Del mismo modo que ocurre con los parámetros del método, cualquier variable que se haya declarado en su interior, será “visible” únicamente dentro del método
- Se creará durante su declaración y se destruirá al abandonar el método, por lo que se “perderán” los valores contenidos en las mismas

```
int suma(int a, int b) {    // se crean e inicializan las variables de los parámetros
    int result;             // se crea la variable result al entrar en el método
    result = a + b;         // se inicializa la variable result con un nuevo valor
    return result;          // devolvemos el valor actual de la variable result
} // todas las variables locales del método (a, b, result) se destruyen al salir

public static void main(String[] args) {
    System.out.println(suma(3, 2)); // imprime 5
    System.out.println(suma(-2, 3)); // imprime 1
    System.out.println(result);     // error: cannot find symbol
}                                  // la variable result no es visible desde main
```



## Constructores (I)

- En el ejemplo anterior, los valores de las *variables de instancia* de cada objeto *Vehiculo* se establecieron de forma manual. Por ejemplo:

```
miCoche.numPasajeros = 5;  
miCoche.capDeposito = 45;  
miCoche.consumo100KM = 6.8;
```

- Sin embargo, esta no es la forma más apropiada de hacerlo. Además de ser propensa a errores, el acceso directo a los atributos del objeto *viola* el principio de la **encapsulación** de nuestros objetos. La forma adecuada de hacerlo será mediante la definición y empleo de **constructores**
- Un *constructor* no es más que un **método** de la clase que se invoca automáticamente cuando se crea un objeto de la misma (operador **new**), siendo su función principal la de **inicializar** sus *variables de instancia*
- El constructor se llama **igual** que la clase, no devuelve ningún valor (tampoco se indica *void*) y puede tener cualquier número de parámetros

### Constructores (II)

---

- Todas las clase tienen un constructor, se haya definido explícitamente o no. Es decir, Java proporciona un *constructor por defecto* a todas las clases que **no** hayan definido el suyo propio
- La función del *constructor por defecto* es inicializar las *variables de instancia* del nuevo objeto a un valor **por defecto**:
  - **0**, para tipos numéricos
  - *false*, para *booleans*
  - *null*, para tipos referenciados
- Normalmente, nuestras clases dispondrán de *constructores parametrizados*, de forma que podamos “*personalizar*” la creación del objeto mediante una serie de valores pasados como *argumentos*. Si no se indicó un valor para alguno de los atributos, se inicializará con el valor por defecto de su tipo
- Una clase puede definir todos los constructores que quiera, con la salvedad de que tengan una lista diferente de parámetros (*sobrecarga*)

## Constructores (III)

### ❑ Ejemplos

```
// Clase sin constructores  
// (constructor por defecto)  
class A {  
    int a;  
}
```

```
// Clase con un constructor  
class A {  
    int a;  
    A() {  
        a = 5;  
    }  
}
```

```
// Clase con más de un construc  
class A {  
    int a;  
    A() { a = 5; }  
    A(int val) { a = val; }  
}
```

```
class DemoA {  
    public ... main(...) {  
        A var = new A();  
  
        System.out.println(  
            "var.a = " + var.a);  
    }  
}
```

```
class DemoA {  
    public ... main(...) {  
        A var = new A();  
  
        System.out.println(  
            "var.a = " + var.a);  
    }  
}
```

```
class DemoA {  
    public ... main(...) {  
        A var1 = new A();  
        A var2 = new A(-3);  
        System.out.println(  
            "var1.a = " + var1.a +  
            "\nvar2.a = " + var2.a);  
    }  
}
```

#### SALIDA

```
var.a = 0
```

```
var.a = 5
```

```
var1.a = 5  
var2.a = -3
```

## Constructores (IV)

### ❑ un constructor para *Vehiculo*...

```
//: Vehiculo.java
class Vehiculo {
    // declaración de atributos (variables de instancia)
    int numPasajeros;        // número de pasajeros
    int capDeposito;         // capacidad del depósito (litros)
    double consumo100KM;     // consumo (litros a los 100KM)

    Vehiculo(int numP, int capD, double cons100) {
        numPasajeros = numP;
        capDeposito = capD;
        consumo100KM = cons100;
    }

    double autonomia() {
        return capDeposito * 100.0 / consumo100KM;
    }

    double litrosDist(int kms) {
        return kms * consumo100KM / 100;
    }
}
```

## Constructores (y V)

- Ahora, cada vez que queramos crear con *new* un objeto del tipo *Vehiculo*, estaremos obligados a proporcionar los *argumentos* correspondientes a los tres parámetros del único constructor de la clase

```
//: VehiculoDemo.java
class VehiculoDemo {
    public static void main(String[] args) {
        final int DIST_SCQ_MAD = 600;

        // Creamos dos nuevas instancias de la clase Vehiculo
        Vehiculo miCoche = new Vehiculo(5, 45, 6.8);
        Vehiculo furgoPepe = new Vehiculo(7, 85, 7.25);

        System.out.print("miCoche puede llevar " + miCoche.numPasajeros);
        System.out.printf(" personas hasta %.2f kms%n", miCoche.autonomia());
        System.out.print("furgoPepe puede llevar " + furgoPepe.numPasajeros);
        System.out.printf(" personas hasta %.2f kms%n", furgoPepe.autonomia());
        System.out.println("Para ir desde Santiago a Madrid (" + DIST_SCQ_MAD + " km):");
        System.out.printf("miCoche: %.2f litros %n", miCoche.litrosDist((double)DIST_SCQ_MAD));
        System.out.printf("furgoPepe: %.2f litros %n", furgoPepe.litrosDist((double)DIST_SCQ_MAD));
    }
}
```

# Destrucción de objetos

- Hemos visto como los objetos son creados dinámicamente, asignando los recursos de memoria necesarios, mediante el uso del operador *new*
- Al iniciarse, la JVM reserva un espacio de la memoria principal del sistema, denominado *heap memory*, que usará para asignar a los objetos y *arrays* que se creen en tiempo de ejecución
- Dado que el tamaño de la *heap memory* es fijo (configurable mediante la opción *-Xmx* del comando *java*) y, en último caso, limitado por el tamaño de la memoria física del sistema real, Java debe disponer de un mecanismo que le permita gestionar ese espacio de la mejor manera
- A diferencia de otros lenguajes, donde el programador debe encargarse de la liberación del espacio asignado, la JVM incluye un sistema denominado *recolector de basura* (*garbage collector*). Se encarga de eliminar, de forma automática y transparente para el programador, aquellos objetos que ya no se usan (sin *referencias*) y liberar los recursos asignados

## La referencia *this* (I)

- Cada vez que invocamos un método de un objeto, dicho método recibe de forma *implícita* un argumento con la *referencia* al *propio objeto*. Esta *referencia* se denomina *this*
- La referencia *this* nos permite acceder a cualquiera de los atributos y métodos del objeto utilizando el *operador punto*
- Suele usarse con frecuencia en constructores para distinguir las *variables de instancia* de los *parámetros del método*, pues suelen emplearse los mismos nombres. Por ejemplo, el constructor de *Vehiculo* podría ser:

```
class Vehiculo {  
    int numPasajeros;        // número de pasajeros  
    int capDeposito;         // capacidad del depósito (litros)  
    double consumo100KM;     // consumo (litros a los 100KM)  
  
    Vehiculo(int numPasajeros, int capDeposito, double consumo100KM) {  
        this.numPasajeros = numPasajeros;  
        this.capDeposito = capDeposito;  
        this.consumo100KM = consumo100KM;  
    }  
    // . . .  
}
```

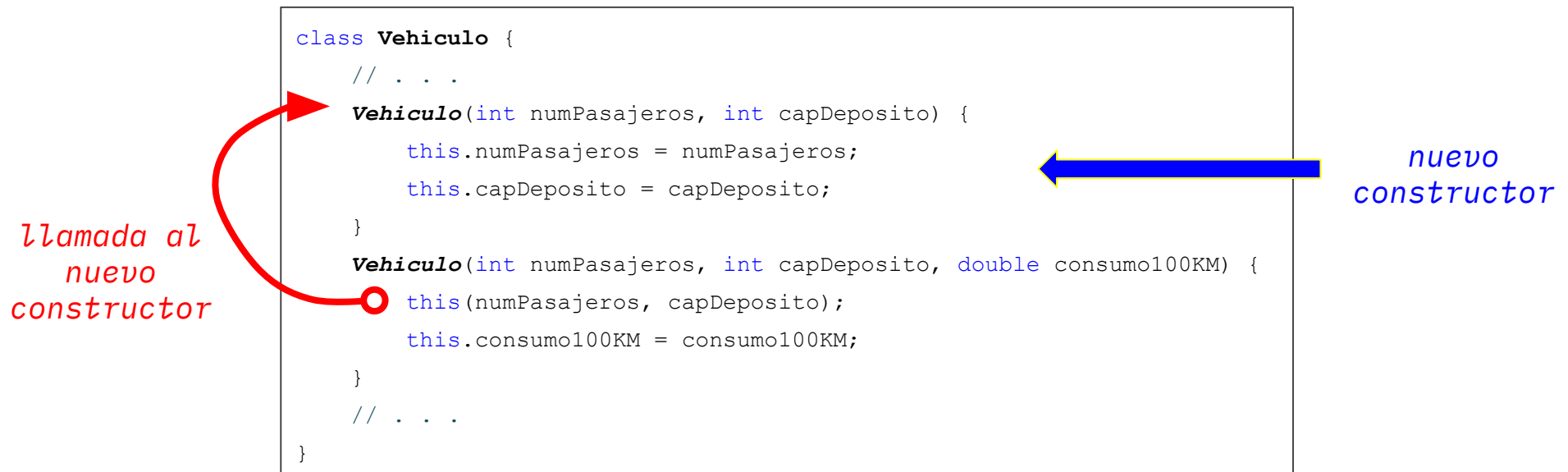
*variables de  
instancia*

*parámetros  
del  
constructor*



## La referencia *this* (y II)

- Otro uso habitual es el de realizar llamadas entre constructores de la misma clase, favoreciendo la reutilización del código de los mismos
- Para la llamada al *constructor* se emplea la notación *this(lista\_argumentos)*
- Supongamos que queremos un nuevo constructor en nuestra clase *Vehiculo* que nos permite crear vehículos indicando sólo el número de pasajeros y consumo, pero no la capacidad del depósito...



```
class Vehiculo {  
    // . . .  
    Vehiculo(int numPasajeros, int capDeposito) {  
        this.numPasajeros = numPasajeros;  
        this.capDeposito = capDeposito;  
    }  
    Vehiculo(int numPasajeros, int capDeposito, double consumo100KM) {  
        this(numPasajeros, capDeposito);  
        this.consumo100KM = consumo100KM;  
    }  
    // . . .  
}
```

llamada al nuevo constructor

nuevo constructor

### Los métodos *get* y *set* (I)

- Con objeto de garantizar la **encapsulación**, no deberíamos poder acceder directamente a la **estructura interna** de los objetos sino a través del **interfaz** correspondiente. Así, el diseñador es libre de modificarla.
- Todas las variables miembro deberían permanecer **privadas**, es decir, ocultas al exterior. Los lenguajes OOP suelen incluir modificadores para configurar el acceso a las variables (*public*, *private*, *protected*)
- Para aquellas que precisen ser “accedidas” para leer o modificar su valor, la clase debe proporcionar los métodos de interfaz correspondientes
- Tradicionalmente, estos métodos reciben el nombre de **getters** (lectura) y **setters** (modificación). Su nombre se forma a partir de los prefijos **get** o **set**, y del nombre de la variable miembro. Por ej: **getVar()**
- Aunque los veremos con más detalle cuando introduzcamos los modificadores de acceso, a modo de ejemplo, vamos a añadir estos métodos a nuestra clase *Vehiculo* para alguno de sus atributos

# Los métodos *get* y *set* (y II)

```
//: Vehiculo.java
class Vehiculo {
    // declaración de atributos (variables de instancia)
    int numPasajeros;        // número de pasajeros
    int capDeposito;         // capacidad del depósito (litros)
    double consumo100KM;     // consumo (litros a los 100KM)

    // . . .

    void setConsumo100KM(double consumo100KM) {
        this.consumo100KM = consumo100KM;
    }

    double getConsumo100KM() {
        return this.consumo100KM;
    }

    // . . .
}
```

- Cuando diseñemos nuestras clases, deberemos *evitar* la tentación de añadir *getters* y *setters* para cada uno de los atributos de la clase. Es decir deberemos plantearnos si puede ser necesario *leer* o *modificar* un atributo. Por ejemplo, ¿tendría sentido un método *setCapDeposito()*?

### Métodos *estáticos* (I)

- Generalmente, accedemos a un miembro de una clase (atributo o método) a través de un objeto de la misma. Sin embargo, es posible declarar miembros que puedan ser usados sin estar asociados a un objeto concreto
- Basta con preceder la declaración del miembro con la palabra clave *static*.
- Estos miembros tendrán carácter *global* o *genérico* (comunes a cualquier instancia) y sólo podrán acceder a otros miembros *estáticos* de la clase
- Un ejemplo es el propio método *main*, que es invocado por la JVM. Al ser el *punto de inicio* de la aplicación, la JVM no lo podría llamar si previamente se tuviera que crear un objeto de la clase que lo contiene
- Para acceder a los miembros *estáticos* de una clase, no necesitamos crear un objeto de la misma. Simplemente utilizaremos el **nombre de la clase** y el *operador punto*. Por ejemplo: `Math.sqrt()`
- Son usos habituales la creación de constantes, variables globales y la creación de métodos de utilidad

## Métodos *estáticos* (y II)

### ❑ método estático de *Vehiculo* para convertir km a millas...

```
//: Vehiculo.java
class Vehiculo {
    // . . .
    static double kmEnMillas(double kms) {
        return kms * 0.621371;
    }
    // . . .
}
```

```
//: VehiculoDemo.java
class VehiculoDemo {
    public static void main(String[] args) {
        // . . .
        double autonomKM = miCoche.autonomia();
        double autonomMi = Vehiculo.KmAMillas(autonomKM);

        System.out.println("La autonomía de miCoche es:");
        System.out.printf("en kms: %.2f%n", autonomKM);
        System.out.printf("en millas: %.2f%n", autonomMi);
    }
}
```

Llamamos al método  
*estático* usando el  
nombre de la clase  
(*Vehiculo*), no el objeto  
*miCoche*

### La clase *String* (I)

- A diferencia de otros lenguajes, donde las *cadena de caracteres* se almacenan mediante *arrays*, Java nos proporciona una clase específica para su almacenamiento y tratamiento, la clase *String*
- Debemos tener presente que se trata de una clase, no de un *tipo primitivo*, por lo que, además de proporcionarnos toda una serie de métodos para el tratamiento de *cadena de caracteres*, todas las variables de tipo *String* serán de tipo *referencia* (punteros a objetos)
- Hasta el momento, hemos estado *creando objetos* de tipo *String* a partir de *literales* (encerrados entre comillas) usando la misma sintaxis que con cualquier variable de tipo primitivo. Por ejemplo:

```
String str = "Hola. Mundo!";
```

- Ésta es una forma de creación de objetos simplificada y específica de la clase *String*

## La clase *String* (II)

- Al igual que el resto de clases, podemos utilizar el operador *new* para la creación de objetos *String*. De hecho, será la forma en la que podremos invocar alguno de sus múltiples constructores

```
String str1 = new String("Hola. Mundo!");  
String str2 = new String(str1);    // Creamos un String a partir de otro
```

- La clase *String* proporciona gran número de métodos para el tratamiento de *cadena de caracteres*. Algunos son:

<i>boolean</i> equal( <i>str</i> )	Devuelve <i>true</i> si contiene los mismos caracteres que <i>str</i>
<i>int</i> length()	Devuelve el número de caracteres
<i>char</i> charAt( <i>index</i> )	Obtiene el carácter en la posición <i>index</i> (empieza en 0)
<i>int</i> compareTo( <i>str</i> )	Devuelve <i>menos que 0</i> si el <i>String</i> es menor ( <i>alfabéticamente</i> ) que <i>str</i> , <i>0</i> si es igual y <i>mayor que 0</i> si es mayor que <i>str</i>
<i>int</i> indexOf( <i>str</i> )	Busca <i>str</i> en la cadena y devuelve la posición del primer carácter. <i>-1</i> si no está
<i>int</i> lastIndexOf( <i>str</i> )	Busca la posición de la última ocurrencia de <i>str</i> en la cadena. <i>-1</i> si no está



## La clase *String* (III)

### ❏ Algunos ejemplos...

```
//: StrDemo.java

class StrDemo {
    public static void main(String[] args) {
        String str1 = "En un lugar de La Mancha";
        int len = str1.length();

        System.out.println("str1: \"" + str1 + "\"");
        System.out.println("str1 tiene " + len +
            " caracteres");
        System.out.println("El primero carácter es: '" +
            str1.charAt(0) + "'");
        System.out.println("Y el último: '" +
            str1.charAt(len - 1) + "'");

        // Comparando dos cadenas
        String str2 = new String(str1);

        System.out.println("str2: \"" + str2 + "\"");
        System.out.println("Comparando str1 y str2...");
        System.out.println("usando == --> " + (str1==str2));
        System.out.println("usando compareTo() --> " +
            str1.compareTo(str2));
    }
}
```

```
// Asignado una nueva cadena a str2
String str3 = "de cuyo nombre no quiero acordarme";
str2 = str1 + ", " + str3;
System.out.println("str2: \"" + str2 + "\"");
System.out.println("La primera ocurrencia de 'de': " +
    str2.indexOf("de"));
System.out.println("La última ocurrencia de 'de': " +
    str2.lastIndexOf("de"));
}

/* Output:
str1: "En un lugar de La Mancha"
str1 tiene 24 caracteres
El primero de sus caracteres es: 'E'
Y el último: 'a'
str2: "En un lugar de La Mancha"
Comparando str1 y str2...
usando == --> false
usando compareTo() --> 0
str2: "En un lugar de La Mancha, de cuyo nombre no quiero
acordarme"
La primera ocurrencia de 'de' está en: 12
La última ocurrencia de 'de' está en: 26
*///:~
```

## La clase *String* (IV)

- Los objetos *String* internamente son **inmutables** (no podemos alterar sus caracteres). Aún así, dispondremos de métodos que nos permitirán tanto extraer *porciones* de la cadena, como obtener nuevas cadenas a partir de la antigua en la que se hayan reemplazado determinadas partes:

<i>String</i> substring( <i>index</i> )	Devuelve la subcadena resultante a partir de la posición <i>index</i>
<i>String</i> substring( <i>ini</i> , <i>fin</i> )	Devuelve la subcadena entre las posiciones <i>ini</i> y ( <i>fin</i> - 1)
<i>String</i> replace( <i>oldCh</i> , <i>newCh</i> )	Devuelve la cadena resultado de reemplazar todos los caracteres <i>oldCh</i> por <i>newCh</i>
<i>String</i> replaceAll( <i>regex</i> , <i>newStr</i> )	Devuelve la cadena resultado de reemplazar todas las subcadenas que coincidan con la <i>expresión regular</i> por la cadena <i>newStr</i>
<i>String</i> replaceFirst( <i>regex</i> , <i>newStr</i> )	Devuelve la cadena resultado de reemplazar la primera subcadena que coincida con la <i>expresión regular</i> por la cadena <i>newStr</i>

- La clase *String* nos proporcionará otros muchos métodos (conversión mayúsculas y minúsculas, dígitos a *strings*, eliminación de espacios,...). Referise al API: <https://docs.oracle.com/javase/8/docs/api/>

# La clase *String* (y V)

```
//: SubstrDemo.java
class SubstrDemo {
    public static void main(String[] args) {
        String str1 = "Vive como si fueras a morir mañana";
        String str2 = "Aprende como si fueras a vivir siempre";
        String str3 = str1 + ";\n" + str2;

        System.out.println("\"" + str3 + "\"");
        System.out.println("substring(22, 33): " + str3.substring(22, 34));
        System.out.println("substring(67, 80): " + str3.substring(61, 74));

        str3 = str3.replace("\n", " ");

        String str4 = "        Mahatma Gandhi        ";
        str4 = str4.trim().toUpperCase(); //: Composición de llamadas a métodos
        System.out.println(str4 + ": \"" + str3 + "\"");
    }
}

/* Output:
Vive como si fueras a morir mañana;
Aprende como si fueras a vivir siempre"
substring(22, 33): morir mañana
substring(67, 80): vivir siempre
MAHATMA GHANDI: "Vive como si fueras a morir mañana; Aprende como si fueras a vivir siempre"
*///:~
```

## Combinando llamadas a métodos

- En el ejemplo anterior, podemos observar cómo es posible **combinar** las llamadas a métodos de diferentes objetos en una única sentencia. Es decir, cada invocación se realizará a alguno de los métodos del **objeto** devuelto por la invocación **precedente**.

- Por ejemplo, en la sentencia:

```
str4 = str4.trim().toUpperCase()
```

- 1) Se realiza la llamada al método **trim()** del *String* referenciado por *str4*. Este método devuelve un **nuevo objeto** de tipo *String* a partir del original en el que se han eliminado los espacios iniciales y finales
- 2) Se llama al método **toUpperCase()** del objeto *String* devuelto en la llamada al método precedente. Este método devuelve un nuevo *String* con todos los caracteres en mayúsculas
- 3) Se asigna (=) a *str4* la *referencia* al nuevo objeto (perdiendo la original)

## Impresión de objetos: método *toString* (I)

- Hasta ahora, hemos empleado los diferentes métodos *print* de `System.out` para mostrar en la consola literales u objetos tipo *String*, o los contenidos de variables de tipos primitivos. Pero, ¿qué ocurre si intentamos imprimir algún otro objeto? Por ejemplo, de la clase *Vehiculo*...

```
//: VehiculoDemo.java
class VehiculoDemo {
    public static void main(String[] args) {
        Vehiculo miCoche = new Vehiculo(5, 45, 6.8);
        Vehiculo furgoPepe = new Vehiculo(7, 85, 7.25);

        System.out.println("miCoche: " + miCoche);
        System.out.println("furgoPepe: " + furgoPepe);
    }
}
```

```
bowman@hal:~/work/src/java$ java VehiculoDemo
miCoche: Vehiculo@3bb20724
furgoPepe: Vehiculo@a704ea6
```



## Impresión de objetos: método *toString* (II)

- Cada vez que intentamos imprimir un objeto, la JVM invoca al método *toString()* del mismo. Este método pertenece a la clase *java.lang.Object*, clase de la que *implícitamente* derivan todas las clases Java y, por tanto, de la que incorporan sus diferentes métodos.
- El método *toString()* devuelve una *representación de tipo String* del objeto. Por defecto, esta representación tiene el siguiente formato:

*Clase@Hash*

- donde,
  - *Clase*, es el nombre de la clase de la que el objeto es una instancia
  - *Hash*, es una representación hexadecimal del código *hash* del objeto (valor numérico *identificativo* del objeto)
- El valor obtenido sería equivalente a invocar:

*getClass().getName() + '@' + Integer.toHexString(hashCode())*

## Impresión de objetos: método *toString* (y III)

- Como cualquier otro método *heredado y no final*, el método *toString* puede ser sobrescrito (de hecho, así se recomienda en la documentación). La única condición, será mantener la misma declaración del método:

```
public String toString( )
```

- Vamos a modificar nuestra clase *Vehiculo*, añadiendo el siguiente método, para obtener una representación textual más “*amigable*” de sus objetos...

```
public String toString() {  
    String ret = "\n-----\n" ;  
    ret += "clase: " + getClass().getName() + ":\n";  
    ret += "pasajeros: " + numPasajeros + "\n";  
    ret += "depósito: " + capDeposito + " litros\n";  
    ret += "autonomía: " + capDeposito*kmPorLitro + " Km\n";  
    ret += "-----" ;  
    return ret;  
}
```

### Paquetes de clases (I)

---

- Al desarrollar aplicaciones, solemos agrupar partes relacionadas del programa. En Java, eso es función de los **paquetes**
- Los paquetes tienen dos funciones básicas:
  - Agrupar una serie de clases relacionadas de forma que se accede a ellas a través del paquete (Por ejemplo: *API de Java*).
  - Encapsular las clases de forma que puedan ser privadas y sólo accesibles desde dentro del paquete (volveremos sobre ello cuando veamos los diferentes modificadores de acceso)
- Los paquetes se basan en el concepto de *namespace*, que define una región declarativa. Dentro de un mismo *namespace*, no puede haber dos clases con el mismo nombre
- Ahora mismo, sólo nos interesa saber cómo crear paquetes, añadir clases a dichos paquetes y qué implicaciones tiene desde el punto de vista de la compilación y ejecución



# Paquetes de clases (II)

### ❑ Definiendo un paquete

- Todas las clases de Java pertenecen a algún paquete. Cuando no se indica explícitamente, se emplea el *paquete por defecto* (*global*)
- Para definir un paquete, añadiremos el comando:

`package nombre_paquete;`

al **inicio del archivo** `.java` con el código fuente. De ese modo la clase(s) definida(s) en el archivo pertenecerá al paquete `nombre_paquete`

- Java emplea el sistema de archivos para la gestión de los paquetes, con cada paquete en su propio directorio. Todas las clases pertenecientes al mismo paquete estarán ubicadas en el mismo directorio, que se llamará como el propio paquete (recuerda que Java es sensible a mayúsculas y minúsculas). En el caso del *paquete por defecto*, el directorio es el propio *directorio actual* donde se encuentre la clase.

## Paquetes de clases (III)

### ❖ Ejemplo:

- las clases A y B pertenecen al paquete: *mipkg*
- la clase C pertenece al paquete: *mipkg.otropkg*

```
//: A.java  
  
package mipkg;  
  
class A {  
    //...  
}
```

```
//: B.java  
  
package mipkg;  
  
class B {  
    //...  
}
```

```
//: C.java  
  
package mipkg.otropkg;  
  
class C {  
    //...  
}
```

Sistema de archivos:

```
/ [raíz del sistema]  
  |-- [directorio de fuentes/]  
    |-- mipkg/  
        |-- A.java  
        |-- B.java  
        |-- otropkg/  
            |-- C.java
```

### Paquetes de clases (IV)

---

- Vimos en el ejemplo que es posible crear una **jerarquía de paquetes**
- La única condición es **replicar** en el sistema de archivos esa misma jerarquía de carpetas, indicar con la sentencia *package* el nombre del paquete al que pertenece la clase y colocar los archivos *.java* en la subcarpeta correspondiente al paquete
- Por otro lado, la misma estructura de paquetes que hayamos definido para los archivos fuente, se trasladará a los archivos *.class* compilados.
- Es decir, el *bytecode* de la *clase A* del ejemplo, pertenece al paquete *mipkg*, del mismo modo que el *bytecode* de la *clase C* pertenecerá al paquete *mipkg.otropkg*
- Esto tiene, como veremos a continuación, ciertas implicaciones a la hora de **ejecutar** clases agrupadas en paquetes

## Paquetes de clases (V)

### ❑ Ejecutando clases de un paquete

- Para ejecutar una clase, tenemos que emplear su nombre *totalmente cualificado*, es decir, incluyendo el paquete al que pertenece. Hasta ahora, no nos habíamos preocupado porque se encontraban en el *paquete por defecto (directorio actual)*. Así, para ejecutar la *clase A* del ejemplo deberemos emplear *mipkg.A* y, para la clase C, usaremos *mipkg.otropkg.C*
- La JVM emplea la variable de entorno *CLASSPATH* (o la opción *-cp* del comando *java*) para localizar los directorios donde se encuentran las clases que queremos ejecutar. El valor por defecto del *classpath* es el *directorio actual* por lo que, si no lo queremos modificar, bastará con situarse en el directorio inmediatamente superior al del paquete y ejecutar desde ahí el comando *java*.
- El *classpath* puede contener múltiples directorios separados por (:)

## Paquetes de clases (VI)

### ❖ Ejecutando las clases *mipkg.A* y *mipkg.otropkg.C*

```
Sistema de archivos:  
  
/ [raíz del sistema]  
  |-- [dir_fuentes/]  
    |-- mipkg/  
      |-- A.java  
      |-- B.java  
      |-- otropkg/  
        |-- C.java
```

- Nos situaríamos en la carpeta *[dir\_fuentes]* y ejecutaríamos:

[para ejecutar la clase *mipkg.A*]      `java mipkg.A`

[para ejecutar la clase *mipkg.otropkg.C*]      `java mipkg.otropkg.C`

- Desde cualquier directorio del sistema podríamos ejecutarlas modificando el *classpath* para incluir el directorio donde se encuentran:

`java -cp [ruta completa hasta dir_fuentes/] mipkg.A`

`java -cp [ruta completa hasta dir_fuentes/] mipkg.otropkg.C`

## Paquetes de clases (VII)

### ❑ La sentencia *import*

- Para poder usar desde un clase cualquier otra clase **pública** de otro paquete, no tenemos más que *cualificarla* con su nombre de paquete
- Por ejemplo, para poder usar la clase *Scanner* del API de Java, deberíamos añadirle el paquete al que pertenece (*java.util*) cada vez que la usemos
- Del mismo modo, haríamos con cualquier clase de un paquete nuestro o de un tercero, con la precaución de establecer el valor adecuado del ***classpath*** para que la JVM pueda encontrar las clases correspondientes.
- Java, para evitarnos el escribir continuamente el nombre de los paquete y hacernos la vida un poco más fácil, incorpora el comando:

```
import paquete1[.paquete2[.paqueteN]].clase;
```

- Este comando provoca que dicha clase se “visible” para la JVM y pueda, a partir de ese momento, ser referida empleando únicamente su nombre

## Paquetes de clases (y VIII)

- Es importante resaltar que el uso de la sentencia *import* es simplemente una **conveniencia** para el programador, y no es técnicamente necesario para la escritura de programas en Java
- Debe situarse al **comienzo** del archivo fuente, a continuación de la sentencia *package* (si existe)
- Podemos *importar* **todas** las clases de un determinado paquete usando:  

```
import paquete1[.paquete2[.paqueteN]].*;
```
- Por ejemplo, la sentencia *import java.util.\*;* *importaría* todas las clases pertenecientes al paquete *java.util*
- Esta sintaxis debe ser usada con precaución, pues podría dar lugar a **conflictos de nombres** entre clases de igual nombre y distinto paquete
- Todas las clases pertenecientes al paquete *java.lang* se importan **automáticamente**, por lo que no es necesario el uso de *import*