

## 2. Programación Orientada a Objetos (POO)

### 2.5. Modularidad en Java y Documentación

---

#### **Programación Modular y Orientada a Objetos**

Felipe Ibañez y Juan Miguel Lopez

[felipe.anfurrutia@ehu.es](mailto:felipe.anfurrutia@ehu.es)

[juanmiguel.lopez@ehu.es](mailto:juanmiguel.lopez@ehu.es)

Dpto. de Lenguajes y Sistemas Informáticos

UPV/EHU

# Contenido

---

- Modularidad en Java
  - Paquetes: clases + interfaces + excepciones
  - Ocultación de la información
- Documentación en Java
  - Ejemplo
  - Definición
  - Creación automática

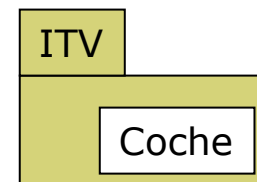
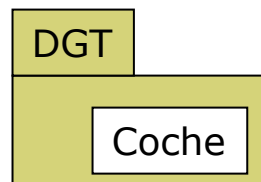
# Motivación

- ¿Qué pasa si queremos utilizar dos abstracciones o encapsulaciones distintas con el mismo nombre (de clase) en un mismo proyecto?

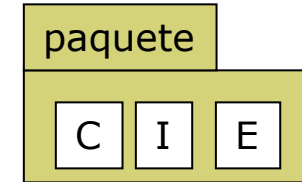
Coche
-marca:String -modelo: String -potencia: int -matricula: String -fechaMatricula1: Date -fechaMatricula2: Date
+getMatricula():String +cambiarMatricula(nueva:String) +cambiarFecha2(nueva:Date)

Coche
-matricula:String -fecha: Date -co2: int -frenosDelanteros: int -frenosTraseros: int -proximaRevisión: Date
+setCo2(pCo2: int) +setFrenosDelanteros(p: int) +setFrenosTraseros(p:int) +setProximaRevisión(d:Date)

- Problema:** colisión de nombres
- Solución:** evitarlos, definir distintos espacios



# Paquetes



- ❑ Definen espacios de nombres
  - Para evitar colisión de nombres
- ❑ Permiten organizar el código de una gran aplicación
  - Las clases(C), interfaces(I) y excepciones(E) relacionadas se declaran en el mismo paquete
  - Ayuda a encontrar dónde están clases e interfaces
- ❑ Sirven para definir bibliotecas de clases e interfaces
  - Reutilización: no volver a (re)inventar la rueda!
- ❑ Permite restringir el acceso a clases y operaciones de un paquete
  - Mayor seguridad del código

# Paquetes: declaración

---

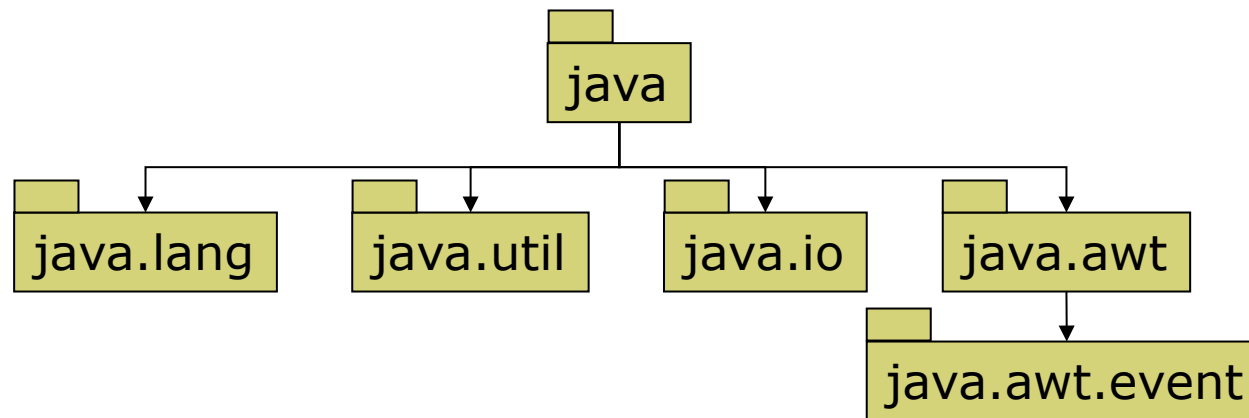
- El paquete al que pertenecen se declara al principio del fichero en el que se especifique la clase o interfaz:

```
package nombre; // declara todo lo que haya en el fichero  
                // como parte del paquete "nombre"
```

- Si no se declara un paquete específico entonces se considera que pertenece a un paquete por defecto (*default*) que no tiene ningún nombre
  - El paquete por defecto sólo se suele utilizar en aplicaciones pequeñas o temporales
  - Se recomienda acostumbrarse a definir paquetes para todas las aplicaciones

# Paquetes: organización (1)

- Los paquetes se organizan de forma jerárquica en paquetes y subpaquetes
  - paquete.subpaquete.subpaquete

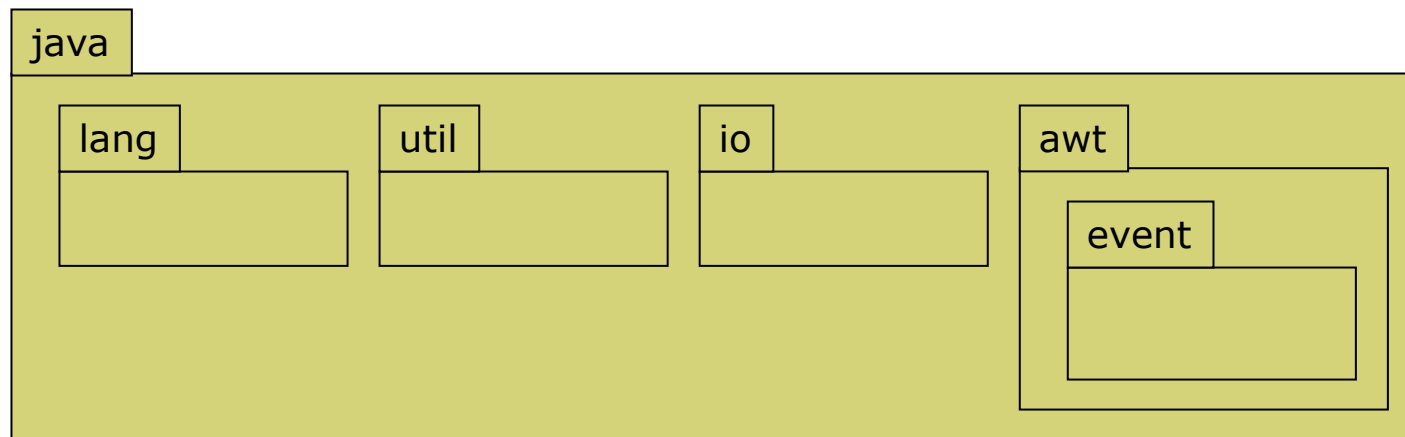


# Paquetes: organización

java



- Esta jerarquía se refleja en la forma de almacenamiento en disco a través de directorios
  - Ej. /java/awt/event



## Paquetes: declaración (2)

---

- Convención para el nombrado de paquetes (para conseguir nombres exclusivos):

- dominio.empresa.departamento.proyecto

Autor del paquete

(nombre del dominio Internet al revés)

- Ejemplo: es.ehu.lsi.felipe.pmyoo
- En disco: /es/ehu/lsi/felipe/pmyoo



# Paquetes: uso

## □ Utilización de nombres (públicos) de un paquete:

### ■ Usando el nombre completo:

```
class ImprimeFecha1 {  
    public static void main (String[] args) {  
        java.util.Date ahora = new java.util.Date();  
        System.out.println(ahora);  
    }  
}
```

### ■ Usando la cláusula *import*:

```
import java.util.Date;  
class ImprimeFecha2 {  
    public static void main (String[] args) {  
        Date ahora = new Date();  
        System.out.println(ahora);  
    }  
}
```

# Compilador: interpretación

---

□ Ejemplo: paquete.subpaquete.subpaquete.Clase

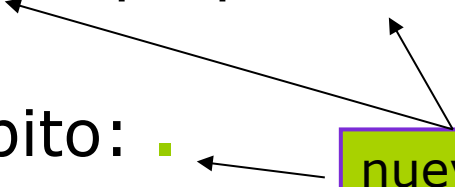
□ Operador de resolución de ámbito: .

- Paquetes dentro de paquetes
- Clases dentro de paquetes
- Métodos dentro de clases
- Variables dentro de métodos y clases

□ Ejemplo: java.lang.System.out

- Es la variable *out*,
- de la clase *System*,
- del paquete *lang*,
- del paquete *java*

nuevo  
operador



# Compilador: interpretación (2)

---

## □ *import*

- No es necesario para el paquete java.lang
  - Siempre se asume: `import java.lang.*;`
- \* permite importar todas las clases e interfaces de un paquete
  - `import java.util.*;` // todas las clases e interfaces de util
  - `import java.*;` // ERROR: no vale para subpaquetes
- Ejemplo: Para importar la clase Applet, hay dos posibilidades:
  - `import java.applet.Applet;` // directamente la clase
  - `import java.applet.*;` // todos los nombres del paquete



El compilador busca en `./classes/java/applet/*`

# Compilador: problemas

---

## ❑ **Error** de compilación:

- Class not found

## ❑ **Solución:** variable de entorno CLASSPATH

- Indica en qué directorios (ej. /classes) o ficheros zip empezar a buscar paquetes
- Se inicializa así:

- ❑ En Unix (por ejemplo en csh):

```
set classpath=(/usr/jdk1.1/lib/classes.zip; ...)
```

- ❑ En Windows, en Propiedades del Sistema-> Opciones avanzadas-> Variables de entorno:

```
CLASSPATH=C:\JAVA\JDK1.1\lib\classes.zip; .;
```

- Directamente en la línea de comandos:

```
> javac -classpath="C:\JAVA\JDK1.1\lib\classes.zip"  
MiClase.java
```

# Paquetes

---

## □ Paquetes estándar de Java

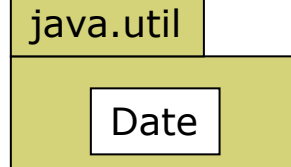
- `java.lang` // Clases e interfaces básicas (se importa por defecto)
- `java.applet` // Clase Applet e interfaces para interacción con navegador
- `java.awt` // Abstract Windowing Toolkit
- `java.io` // E/S
- `java.net` // Clases para comunicación a través de protocolos Internet
- `java.rmi` // Programación distribuida
- `java.security` // Seguridad en Java
- `java.util` // Utilidades
- `javax.swing` // Componentes gráficos para Java

## □ Documentados en línea (Java API):

<http://DirectorioInstalacionJava/docs/api/index.html>

- Se puede descargar desde <http://www.oracle.com/technetwork/java/javase/downloads/index.html> como un fichero zip que se suele descomprimir en el directorio donde se haya instalado el JDK

# Paquetes: Ejemplo



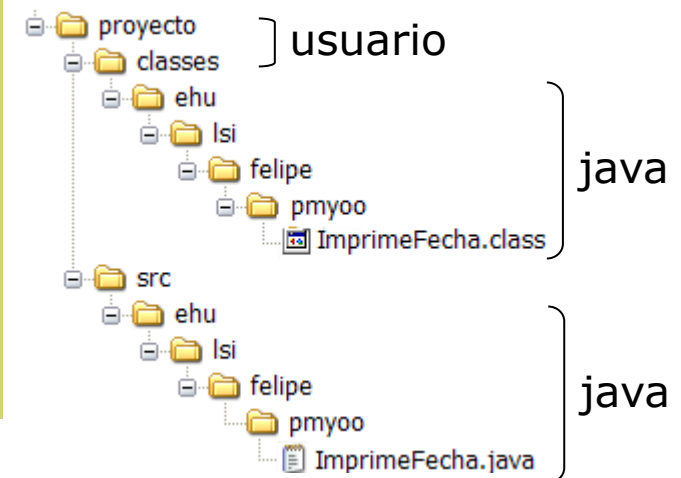
```
package ehu.lsi.felipe.pmyoo;
```

```
import java.util.Date;
```

```
class ImprimeFecha {  
    public static void main (String[] args) {  
        Date ahora = new Date();  
        System.out.println(ahora);  
    }  
}
```

La clase ImprimeFecha está declarada dentro del paquete ehu.lsi.felipe.pmyoo

La clase System está dentro del paquete java.lang  
(que se importa por defecto)  
Se podría poner java.lang.System.out.println



# Ocultación de la información

---

## □ Principio conservador :

*“A menos que exista una razón importante para que un miembro sea accesible, será declarado privado”*

## Ocultación de la información (2)

---

- ❑ La ocultación de la información en Java se realiza a través de los modificadores de visibilidad
- ❑ Los modificadores de visibilidad se aplican a nivel de **clase** y a nivel de **miembro** (atributo o método)



# Modificadores de Visibilidad (clase)

---

- Una clase puede declararse:
  - **public**: en este caso todas las otras clases la pueden utilizar
  - con la visibilidad a nivel de **paquete** (por defecto), esto es, no se puede utilizar fuera del paquete

# Modificadores de Visibilidad

## (atributos y métodos)

---

- ❑ La visibilidad de las variables miembro (atributos) y los métodos (operaciones) de una clase puede ser:
  - **package** (*por defecto*): sólo accesibles desde su paquete (no se puede acceder desde sus subpaquetes)
  - **public**: accesibles desde todas las clases
  - **private**: sólo accesibles desde los métodos de la clase
  - **protected**: el miembro es accesible desde las clases de su paquete y desde las subclases

# Modificadores de Visibilidad (resumen)

---

	<b>Clase</b>	<b>Subclase</b>	<b>Paquete</b>	<b>Resto</b>
private	X			
por defecto	X		X	
protected	X	X	X	
public	X	X	X	X

# Ocultación de la información

---

- ▣ Usando los modificadores de visibilidad conseguimos:
  - ▣ evitar accesos directos
  - ▣ controlar los accesos
  - ▣ no tener problemas al **cambiar la implementación del paquete** mientras no se cambie la interfaz

# Contenido

---

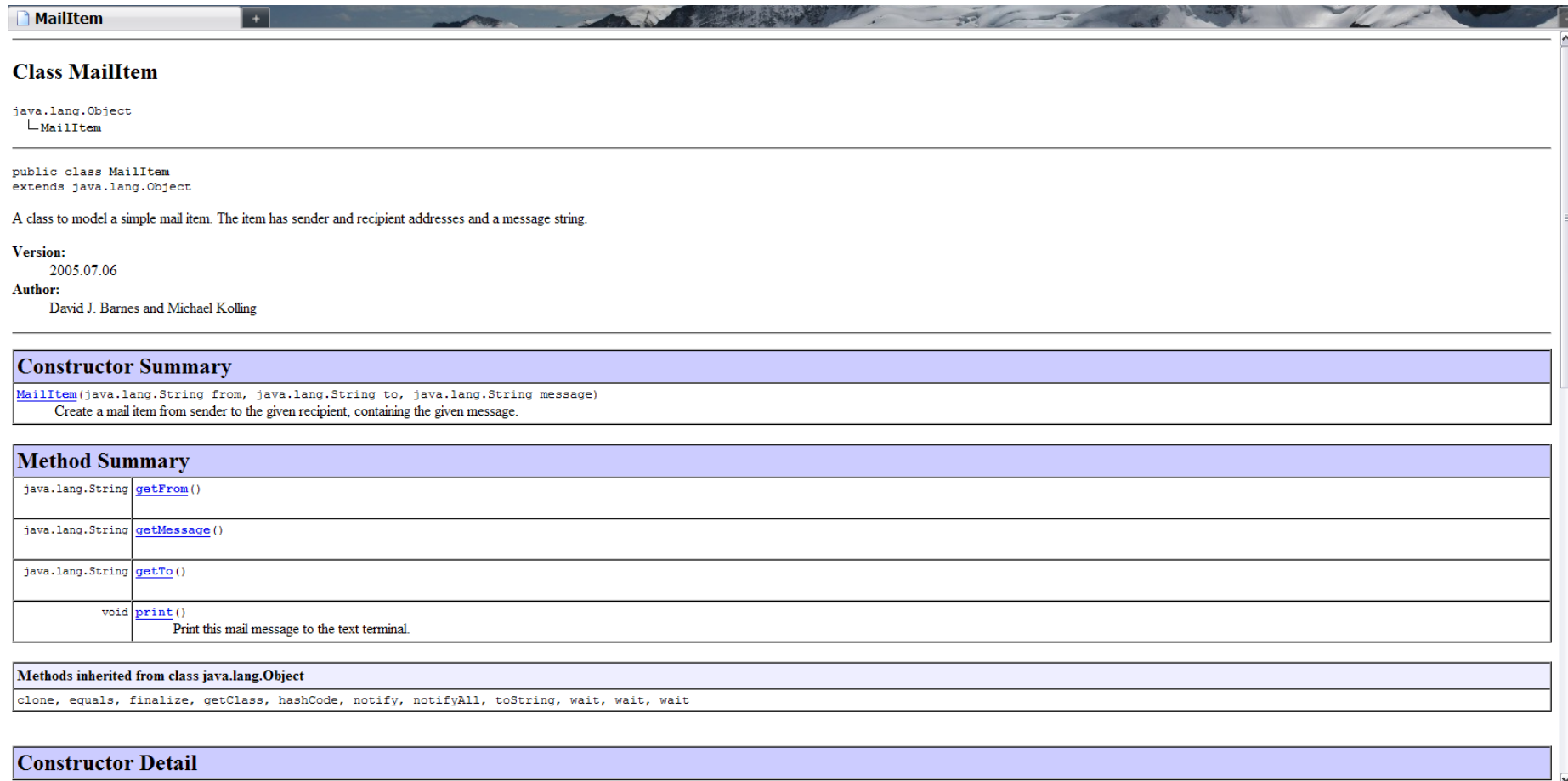
## □ Modularidad en Java

- Paquetes: clases + interfaces + excepciones
- Ocultación de la información

## □ Documentación en Java

- Ejemplo
- Definición
- Creación automática

# Documentación en Java: Ejemplo



The screenshot shows the JavaMail API documentation for the `MailItem` class. The page is titled "MailItem" and shows the class hierarchy: `java.lang.Object` is the superclass, and `MailItem` is the subclass. The class is defined as `public class MailItem extends java.lang.Object`. A description states: "A class to model a simple mail item. The item has sender and recipient addresses and a message string." The version is 2005.07.06, and the authors are David J. Barnes and Michael Kolling.

**Constructor Summary**

Constructor
<code>MailItem(java.lang.String from, java.lang.String to, java.lang.String message)</code> Create a mail item from sender to the given recipient, containing the given message.

**Method Summary**

Method
<code>java.lang.String getFrom()</code>
<code>java.lang.String getMessage()</code>
<code>java.lang.String getTo()</code>
<code>void print()</code> Print this mail message to the text terminal.

**Methods inherited from class java.lang.Object**

Method
<code>clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait</code>

**Constructor Detail**

- Nos sirve para entender la funcionalidad ofrecida por una clase o interfaz (API), que lo ha implementado otro programador

# Documentación en Java: Definición

```
/**
 * A class to model a simple mail item. The item has sender and recipient
 * addresses and a message string.
 * @author David J. Barnes and Michael Kolling
 * @version 2006.03.30
 */
```

Documentación  
para la clase

```
public class MailItem{
    private String from;
    private String to;
    private String message;
```

```
/**
 * Create a mail item from sender to the given recipient,
 * containing the given message.
 * @param from The sender of this item.
 * @param to The intended recipient of this item.
 * @param message The text of the message to be sent.
 */
```

Documentación para  
el método que tiene  
parámetros

```
public MailItem(String from, String to, String message) {
    this.from = from;
    this.to = to;
    this.message = message;
}
```

```
/**
 * @return The sender of this message.
 */
```

Documentación para el  
método que devuelve  
un valor

```
public String getFrom(){
    return from;
}
```

# Documentando el código: sintaxis Javadoc

---

- ❑ Para la clase ponerla inmediatamente antes de la clase y ser encerrado entre `/**` y `*/`
- ❑ Para los métodos: usar los rótulos
  - `@param` *variable* descripción
  - `@return` descripción
  - `@throws` descripción de clase
- ❑ Para los datos públicos: `/** ...*/`
- ❑ Comentarios Generales:
  - `@author` nombre
  - `@version` texto
  - `@since` texto
  - `@see` link, para referenciar otros recursos (clases, ...)

**Ojo:** Se parece a los comentarios (`/* ... */` ó `//`), pero date cuenta que la documentación requiere dos asteriscos `/**` al inicio



# Documentando el código: sintaxis Javadoc

- La herramienta **Javadoc** genera de manera automática documentación a partir de los programas fuente Java. Para ello, estos programas deben tener comentarios y etiquetas entendidas por Javadoc `/** ... */`

Etiqueta	Quién lo utiliza	Objetivo
<code>@author</code> name	Clases e interfaces	Autor del código. Se pone una etiqueta para cada autor.
<code>@deprecated</code>	Clases, métodos	Método anticuado. Mejor no utilizarlo.
<code>@exception</code> name description	Métodos	Excepciones que el método puede elevar. Se pone una etiqueta para cada excepción posible.
<code>@param</code> name description	Métodos	Para describir los parámetros, su utilización y su tipo. Se pone una etiqueta para cada parámetro.
<code>@return</code> description	Métodos	Para describir los valores devueltos por cada método y su tipo.
<code>@since</code>	Clases, métodos	Desde qué versión está. Ej: desde .JDK 1.1
<code>@see</code> ClassName	Clases, interfaces, métodos y atributos.	Pondrá la dirección para conectarse con esta clase en la documentación
<code>@see</code> ClassName#NombreMétodo	Clases, interfaces, métodos y atributos.	Pondrá la dirección para conectarse con este método en la documentación.
<code>@version</code> text	Clases e interfaces	Información acerca de la versión.

# Documentación en Java:

## Creación automática

---

- ❑ Se pueden usar todo tipo de etiquetas `<html>` incrustados en las descripciones
- ❑ ¿Cómo generar la documentación?:
  - `javadoc -d docDirectorio *.java`
- ❑ Para la documentación de un paquete:
  - `javadoc -d docDirectorio nombrePaquete`
- ❑ En BlueJ:
  - Tools → Project Documentation
- ❑ Se genera una carpeta `/doc` con toda la documentación de las clases del proyecto en formato HTML

# Documentación en java: referencias

---

- ❑ Más información en moodle: javadoc.pdf
- ❑ Sun Microsystems, "How to Write Doc Comments for the Javadoc Tool",  
<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>
- ❑ Emil Vassev, "DMS API Documentation", Concordia University, Montreal, Quebec, Canada, 2005
- ❑ Wikipedia. "Application Programming Interface API)".  
[http://en.wikipedia.org/wiki/Application\\_programming\\_interface](http://en.wikipedia.org/wiki/Application_programming_interface)
- ❑ Wikipedia. "Comparison of Documentation Generators".  
[http://en.wikipedia.org/wiki/Comparison\\_of\\_documentation\\_generators](http://en.wikipedia.org/wiki/Comparison_of_documentation_generators)