LPS: Programación Orientada a Objetos con Java en Eclipse

Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

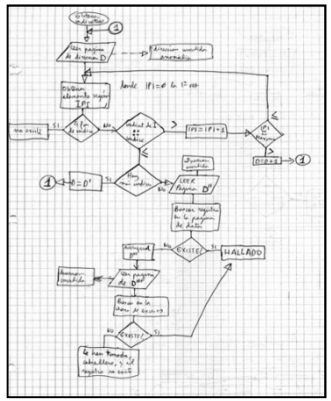
Universidad Complutense de Madrid
www.ucm.es

Programación Orientada a Objetos (POO)

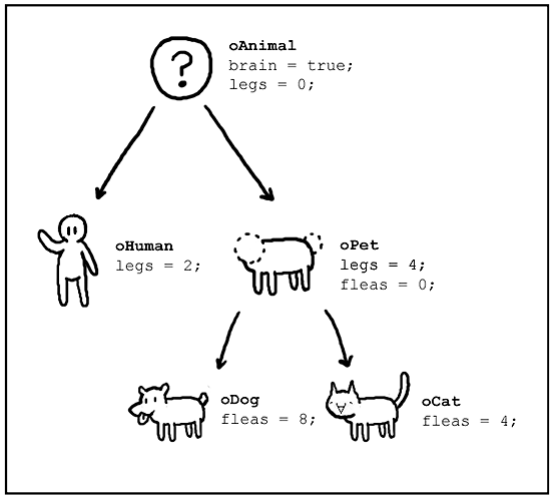
- Paradigma de programación distinto de la Programación Estructurada (PE)
  - Surge con la invención del lenguaje Simula (Dahl y Nygaard, 1967), se consolida con SmallTalk (Kay, Ingalls, Kaehler, Goldberg et al., 1972) y se populariza con C++ (Stroupstrup, 1983)
  - Muchos de los lenguajes usados actualmente siguen esta forma de ver la programación, como ActionScript, C#, JavaScript, PHP, Python, Ruby, Visual Basic .NET y Java

POO versus PE

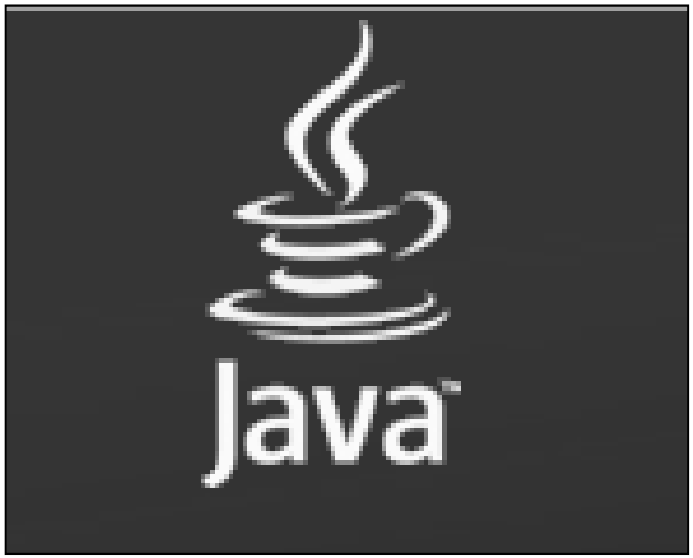
- Según la PE, programar es definir algoritmos usando tres estructuras básicas: secuencia, selección e iteración



- Según la POO, programar es *en primer lugar* definir cómo interactúan una serie de "objetos imaginarios" entre sí (mayor nivel de abstracción)

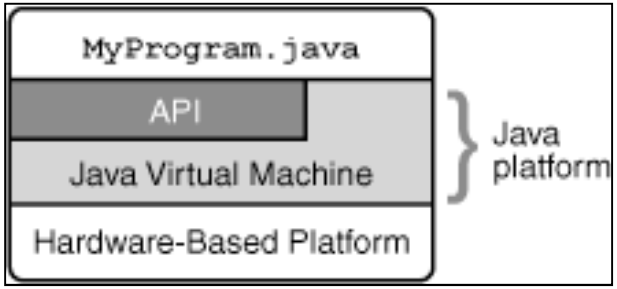


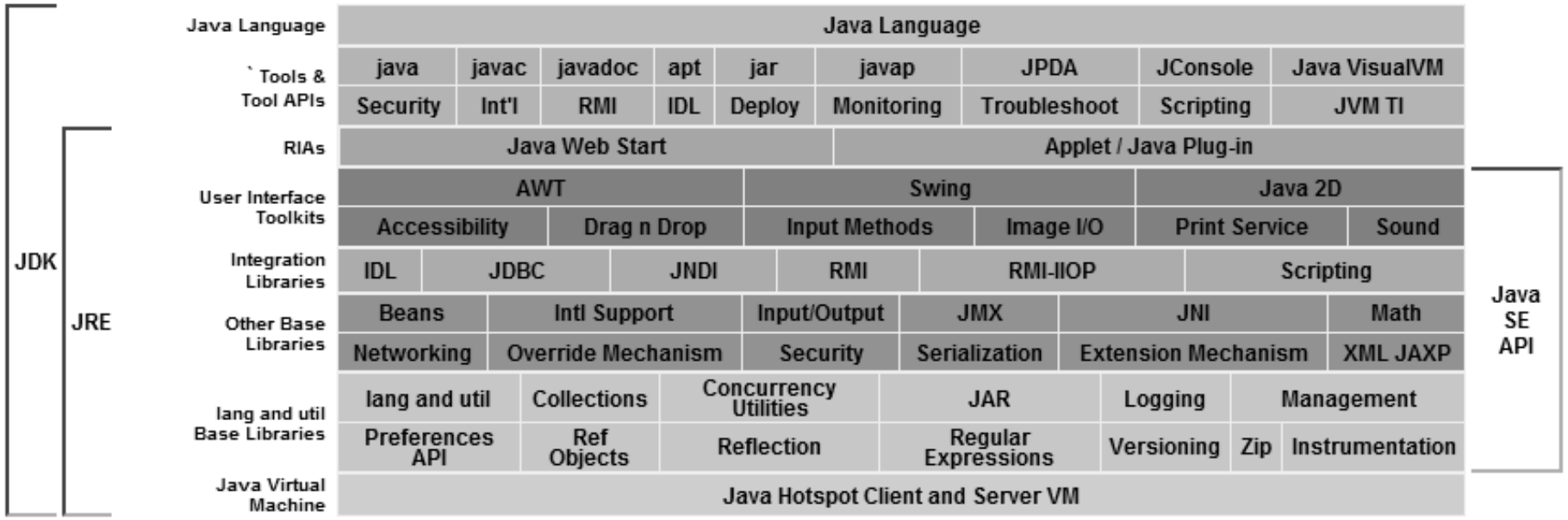
Java



- Java es el nombre de una gran plataforma tecnológica www.java.com
  - Lenguaje de programación orientado a objetos
  - Interfaz de programación de aplicaciones (Java API)
  - Bibliotecas software (Core Java)
    - Subprogramas fundamentales para manejar cadenas, ficheros, procesos, entrada/salida del sistema, etc.
  - Herramientas de desarrollo (JDK)
    - Compilador, depurador, generador de documentación, etc.
  - Entorno de ejecución (JRE)
    - Intérprete en forma de máquina virtual

Java Standard Edition

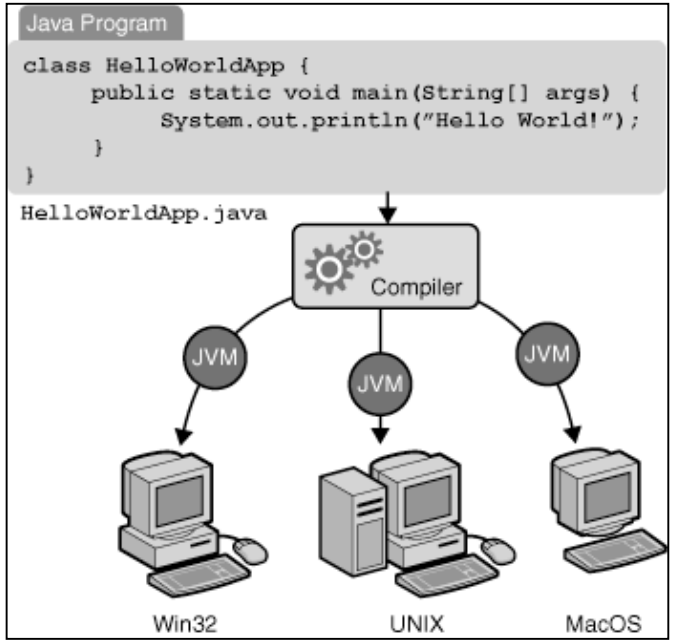


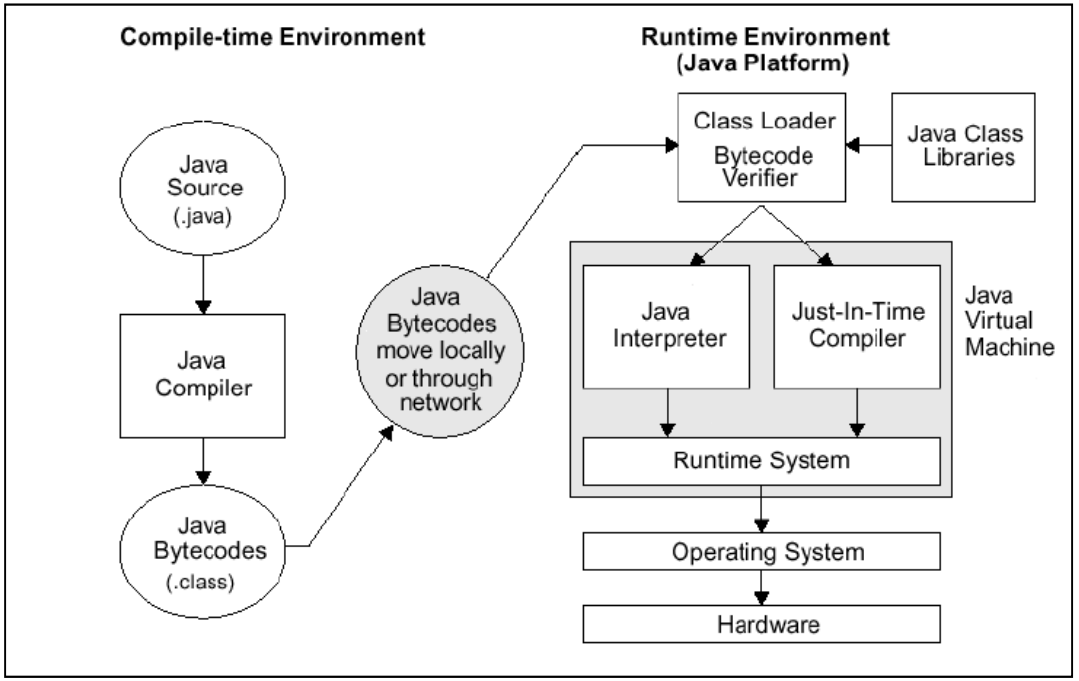


http://download-llnw.oracle.com/javase

Filosofía multiplataforma

- "Write Once, Run Anywhere"
  - Ofrecer un lenguaje y unas herramientas de alto nivel que permiten programar con independencia total de la plataforma subyacente (Sistema operativo y hardware)





Ejecución de una aplicación Java

- Los programas Java no se traducen a ficheros ejecutables como los de PC/Windows o Macintosh/MacOS, sino a ficheros ejecutables únicamente por la Máquina Virtual de Java (JVM)
  - Los programas Java se escriben en uno o más ficheros de texto (`*.java`)
  - El compilador traduce uno a uno estos ficheros de texto a ficheros binarios escritos en un lenguaje intermedio llamado Java Bytecode (`*.class`), muy cercano al código máquina
  - Se debe disponer de un ejemplar de la JVM implementada en la plataforma concreta donde se quiere ejecutar la aplicación
  - Dicho ejemplar de la JVM interpreta los ficheros binarios, ejecutando la aplicación de manera normal sobre la mencionada plataforma de destino

Herramientas de desarrollo

- JDK (Java Development Kit)
  - Varias herramientas que incluyen el compilador de Java (**`javac`**)
  - Para editar los ficheros fuente vale cualquier editor de texto
- JRE (Java Runtime Environment)
  - Ejemplar de la JVM implementado para la plataforma deseada (**`java`**)
  - Se requieren además unas bibliotecas de la JVM cuya ruta (por ejemplo **`c:\jdk1.6.0_21\bin`**) debe estar guardada en una variable de entorno (por ejemplo **`PATH`** dentro de la consola de Windows)
- Las especificaciones de Java son públicas, con lo que existen varias implementaciones, siendo la de Oracle la de referencia http://www.oracle.com/technetwork/java/index.html
  - Hay varias ediciones (distribuciones) cada una con varias versiones
  - Nosotros utilizaremos Java Standard Edition (SE) 6 http://download-llnw.oracle.com/javase/6/docs/index.html
- Aunque todo se puede usar desde la línea de comandos, también existen entornos de desarrollo que facilitan su uso

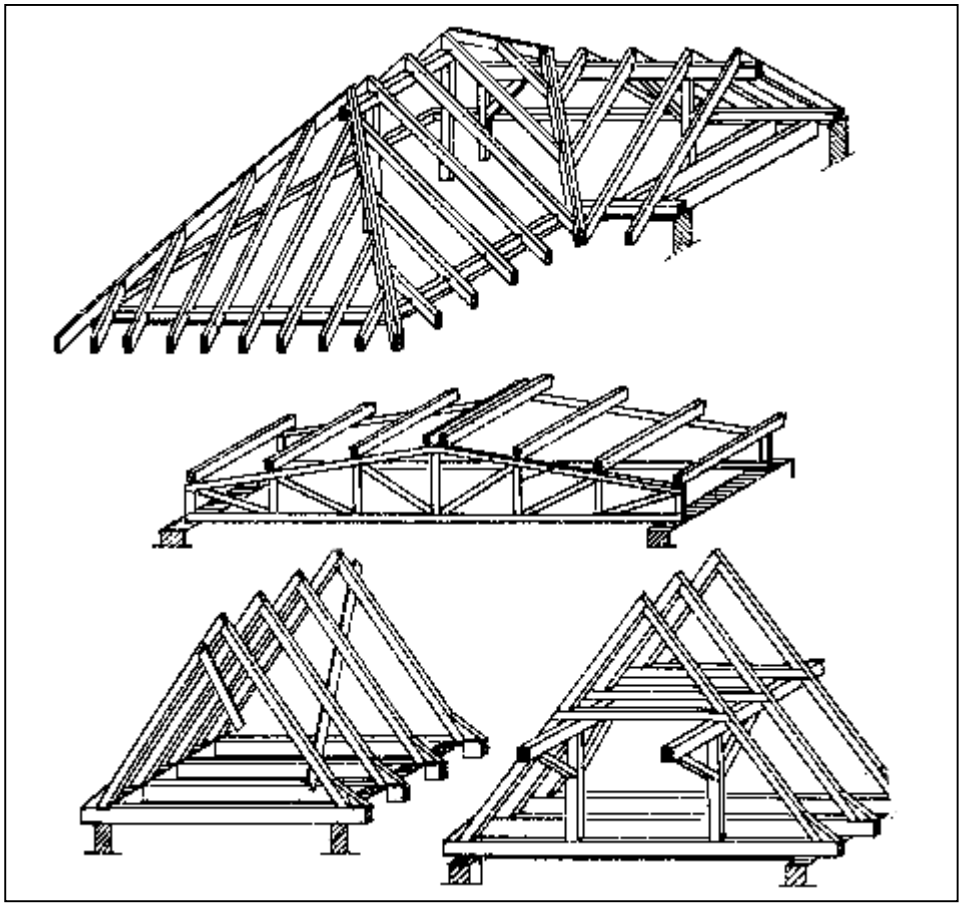
Organización de un proyecto

- Directorio con ficheros fuente (**`.java`**)
  - **`src`**
- Directorio con ficheros binarios (**`.class`**)
  - **`bin`**
- Directorio con bibliotecas (**`.class`** ó **`.jar`**)
  - **`lib`**
- Directorio con ficheros de pruebas
  - **`test`**
- Directorio con la documentación
  - **`doc`**

Distribución de una aplicación

- La aplicación está compuesta por todos los ficheros binarios, bibliotecas y recursos que utiliza el programa (pudiendo incluirse opcionalmente los ficheros fuente)
- El JDK incluye una herramienta llamada **`jar`** que permite empaquetar todos estos ficheros en uno solo (**`.jar`**) para facilitar su distribución y ejecución
  - Para que la aplicación empaquetada en fichero **`.jar`** se pueda ejecutar de forma más directa conviene añadir un sencillo fichero de texto **`META-INF/MANIFEST.MF`** que indica cual es el fichero binario que contiene la primera línea de código de la aplicación

El lenguaje Java: Aspectos estructurados



El lenguaje Java

- Lenguaje orientado a objetos de alto nivel
  - Creado por Sun Microsystems en 1995, liberado casi totalmente bajo licencia GNU GPL en 2007 y adquirido por Oracle en 2009
- De sintaxis conocida
  - Muy similar a C y C++ (aunque funcione de manera muy diferente)
- Portable
  - Los ficheros compilados se interpretan por cualquier ejemplar de la JVM
- Robusto y seguro
  - Fuertemente tipado
  - Gestión de memoria restringida al mínimo: el programador no puede usar punteros a direcciones de memoria
- Puro en cuanto a orientación a objetos (desde J2SE 5.0, 2004)
  - Todo lo que existe durante la ejecución de un programa Java es reconocible bajo alguna "Clase"
- Dotado de genericidad (desde J2SE 5.0, 2004)

El programa *¡Hola Mundo!*

```
/* Clase principal del programa ¡Hola Mundo! */
public class Principal {

  /* Método para saludar */
  private void saludar() {
      System.out.println("¡Hola Mundo!");
  }

  /* Método principal */
  public static void main(String[] args) {
      Principal principal = new Principal();
      principal.saludar();
  }
}
```

Comparación entre Java y C++

- Similitudes
  - Tipos básicos
  - Sintaxis (sobretodo de las estructuras de control)
- Diferencias
  - Ejecución y tratamiento de objetos
  - Declaración de clases, métodos y variables
  - Java no usa ficheros de cabecera
  - En Java no hay **`struct`** ni **`union`**
  - En Java no hay **`typedef`**
  - En Java no se usan *punteros a direcciones de memoria* (sino *referencias a objetos*, que es algo más abstracto)
  - En Java no hay sobrecarga de operadores
  - En Java se pueden usar caracteres especiales en identificadores (vocales con tilde, eñes, etc.)

Identificadores

- Permiten nombrar los distintos elementos de un programa
  - Variables, objetos, clases, paquetes, interfaces…
- Sintaxis
  - Comienzan con letra (incluyendo _ y $)
  - Van seguidos de letras o dígitos
  - Pueden tener cualquier longitud
  - Se distinguen mayúsculas de minúsculas
- Ejemplos
  - x
  - _var1
  - MAXIMO
  - $Caracter

Palabras reservadas

- Tienen un propósito especial en el lenguaje y por lo tanto no pueden utilizarse como identificadores

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

Variables

- Unidad básica de almacenamiento de información cuyo valor puede cambiar durante el programa
- Tienen asociado un determinado “tipo de datos”
- Declaración

  tipo identificador;

  tipo identificador [= valor_inicial] [, ident [= valor_ini]] ...;

```
int numero;   // No se ha producido la inicialización
int max = 5;
boolean sino = true;
```

Constantes

- Unidad básica de almacenamiento de información cuyo valor nunca cambia
- Se declaran con la palabra clave **`final`**

```
final float PI = 3.141592;
final int MAX = 255;
final int ABIERTO = 0, CERRADO = 1;
final boolean FALSO = false;
```

Tipos primitivos

- Enteros (con signo)

		MIN_VALOR	*MAX_VALOR*
`byte`	8 bits	-128	+127
`short`	16 bits	-32768	+32767
`int`	32 bits	-2147483648	+2147483647
`long`	64 bits	-2^{63}	$2^{63}-1$

- Reales (coma flotante, según estándar IEEE 754-1985):

`float`	32 bits
`double`	64 bits

- Lógico o booleano (1 bit)

`boolean`	1 bit

- Caracteres (según estándar ISO Unicode 1.1 de 16 bits)

`char`	16 bits

Por ejemplo `'a' 'A' '\n' '\t' '\u0058'` → `'X'`

Enumerados

- Para definirlos se usa la palabra **`enum`**
- El compilador lo traduce en clases que heredan de **`java.lang.enum`**
- Tiene más funcionalidad que en C/C++ ya que desde el punto de vista de la máquina virtual los tipos enumerados son también clases

```
public enum Direccion {
  Norte, Este, Sur, Oeste;
}
```

Operadores aritméticos y de asignación

- Aritméticos binarios

  `+  -  *  /  %`

- Aritméticos unarios
  - Pre/postincremento `++x  x++`
  - Pre/postdecremento `--x  x--`
- Operadores de asignación
  - Normal `x=y`
  - Adición `x+=y`
  - Sustracción `x-=y`
  - Multiplicación `x*=y`
  - División `x/=y`

Operadores lógicos y relacionales

- Operadores lógicos
  - Y lógica `x && y`
  - O lógica `x || y`
  - Negación `! x`
- Operadores relacionales (Comparaciones)
  - Identidad `x == y`
    - Mejor usar sólo entre valores de tipo básico, no con objetos
  - Diferencia `x != y`
  - Mayor que `x > y`
  - Menor que `x < y`
  - Mayor o igual que `x >= y`
  - Menor o igual que `x <= y`

Instrucciones de selección

```
if (expresion_booleana)
     instruccion
```

```
if (expresion_booleana)
     instruccion1
else
      instruccion2
```

```
switch (expresion)
{
     case expresión-constante1: instrucciones; break;
     case expresión-constante2: instrucciones; break;
     ...
     default: instrucciones
}
```

- La **expresion** puede ser de tipo **char**, **byte**, **short** o **int**

Instrucciones de iteración

- **while (expresión_booleana)**
  instrucción

- **do**
  instrucción
  while (expresión_booleana)

- **for (inicialización;condición;incremento)**
  instrucción

Comentarios

- Comentarios hasta el final de línea

```
// Comentario
```

- Comentarios de varias líneas

```
/* Comentario que como podéis
   ver, ocupa más de una línea */
```

JavaDoc

- Herramienta para generar documentación de APIs en HTML, insertando comentarios especiales en Java
  - Llevan un asterisco adicional al principio, y usan etiquetas específicas para marcar la información más relevante

```
/** Comentario de documentación
   @author Federico Peinado
   @see Consulta la lista de etiquetas disponibles */
```

Tag	Descripción	Uso
@author	Nombre del desarrollador.	nombre_autor
@deprecated	Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores.	descripción
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.	nombre_parametro descripción
@return	Informa de lo que devuelve el método, no se puede usar en constructores o métodos "void".	descripción
@see	Asocia con otro método o clase.	referencia (#método(); clase#método(); paquete.clase; paquete.clase#método()).
@throws	Excepción lanzada por el método	nombre_clase descripción
@version	Versión del método o clase.	versión

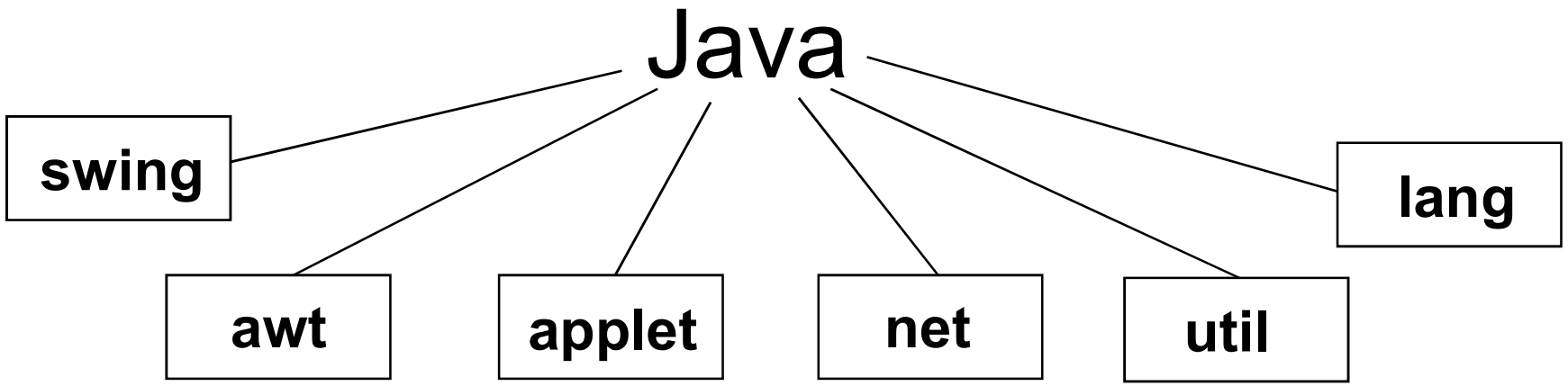
http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html

Organización en paquetes

- Los paquetes (**`packages`**) son la manera de organizar espacios de nombres dentro de Java
  - Agrupación de clases, interfaces… y otros subpaquetes
  - En disco, la estructura de paquetes se organiza en directorios
    - Las clases del paquete **`lps.p1.logica`** se guardan en el directorio **`src/lps/p1/logica`**
  - Por convenio, su nombre empieza por minúsculas
  - Las posiciones de los nombres suelen ir a la inversa de cómo se forman los nombres de los dominios en Internet
    - Ejemplo: **`org.apache.xml`**
  - El paquete **`default`** contiene todas las clases no definidas explícitamente en un paquete
  - Así se usan los nombres de los paquetes:
    - Nombre completo **`java.util.Date`**
    - Usando cláusula **`import`**

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

Paquetes de Core Java



Paquete **`lang`**

Clases con funcionalidades básicas (arrays, cadenas de caracteres, entrada/salida, excepciones, hilos…)

Paquete **`util`**

Utilidades (números aleatorios, vectores, propiedades del sistema…)

Paquete **`net`**

Conectividad y trabajo con redes (sockets, URLs…)

Paquete **`applet`**

Desarrollo de aplicaciones directamente ejecutables en navegadores web

Paquetes **`awt`** y **`swing`**

Desarrollo de interfaces gráficas de usuario

y muchos más…

El lenguaje Java: Aspectos orientados a objetos

Clases

- Por convenio, sus nombres empiezan con mayúsculas
- La clase se declara utilizando la palabra reservada **`class`**, normalmente anteponiendo **`public`** para indicar que es *pública* y que se puede acceder a ella desde cualquier punto
  - Cada clase se crea en un fichero con el mismo nombre que ella y de extensión **`.java`**
  - En el caso de las *clases internas* (ya sean locales o anónimas) veremos que se pueden usar otros tipos de acceso…
- Cada método y atributo de la clase son los que habitualmente especificarán su *tipo de acceso* particular
  - **`public`** – Público, accesible desde cualquier parte del programa
  - **`protected`** – Protegido, accesible desde el propio paquete y las subclases que pueda tener en cualquier otro paquete
  - *Por defecto (sin tipo)* – "Amigable", accesible desde el propio paquete
  - **`private`** – Privado, accesible desde dentro de la propia clase

Ejemplo de clase

```
[Modificadores] class NombreClase [extends SuperClase]
[implements Interfaz1[,InterfazN]]{

   // definición de los atributos de la clase
   [tipoAcceso] tipo1  identificador1;
   [tipoAcceso] tipo2  identificador2;

   // definición de los métodos de la clase
   [tipoAcceso] tipoDevuelto nombreMetodo1 (listaArgumentos) {
       //instrucciones del método1
   }

   [tipoAcceso] tipoDevuelto nombreMetodo2 (listaArgumentos) {
              //instrucciones del método2
   }
}
```

Constructores

- Son como “métodos especiales” que inicializan un objeto al crearse
  - Son llamados automáticamente al llamar a *new* y no devuelven nada
  - Sus nombres son iguales a los de la clase (varían sólo en los argumentos)
  - Si no definimos ningún constructor *explícitamente*, Java proporciona uno *implícito* (sin argumentos y con un comportamiento “por defecto”)

```
public class Rectangulo{
  int _x;
  …
  int _alto;
  // Constructor
  public Rectangulo(int x1, int y1, int ancho, int alto){
    _x = x1;
    _y = y1;
    this._ancho = ancho;
    this._alto = alto;
  }
```

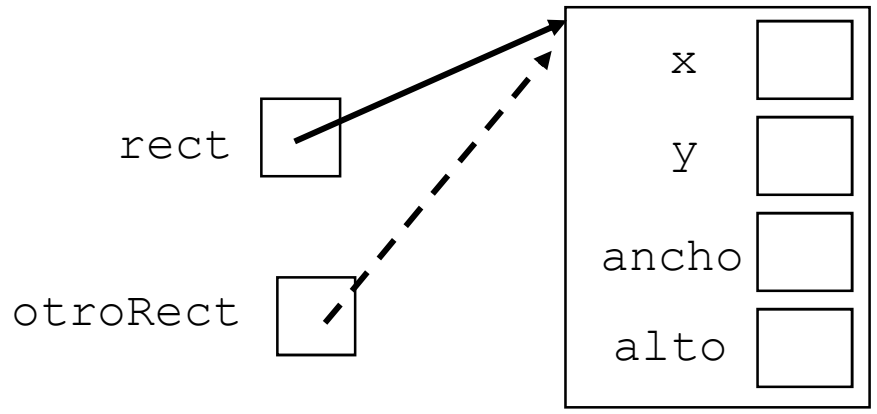
Creación de objetos

- Para crear un objeto hay que usar la instrucción **`new`**
  - No se pueden crear objetos directamente en la pila de memoria (*stack*)
  - Se usa **`new`** hasta para crear los componentes de un *array*
  - Siempre hay que especificar el constructor al que se llama, incluso si es uno sin argumentos
- **`New`** crea un ejemplar de la clase indicada y devuelve su *referencia*
  - Se reserva espacio en el montículo de memoria (*heap*) para sus atributos

```
// Crear la referencia
Rectangulo rect;

// Crear el objeto
rect = new Rectangulo();

// Asignar la referencia
Rectangulo otroRect;
otroRect = rect;
```



Llamadas a métodos

- Se hacen desde un objeto a otro mediante el operador de acceso (.)
  `referenciaObjeto.nombreMetodo(listaArgumentos);`
  - Los argumentos de tipos simples son pasados por valor
  - Los argumentos que son objetos son pasados por referencia

```
public class Rectangulo {
   int _x; …
   int calcularSuperficie(){…}
   void mostrarValores(){…}
   public static void main(String args[]) {
         Rectangulo rect;
         rect = new Rectangulo();
         rect._x = 5; rect._y = 7; rect._ancho = 4; rect._alto = 3;
         int area = rect.calcularSuperficie();
         rect.mostrarValores();
         System.out.println("Superficie: " + area );
         System.out.println("x= " + rect._x + ” y= " + rect._y );
         System.out.println("ancho= " + rect._ancho + ” alto= " + rect._alto );
   }
}
```

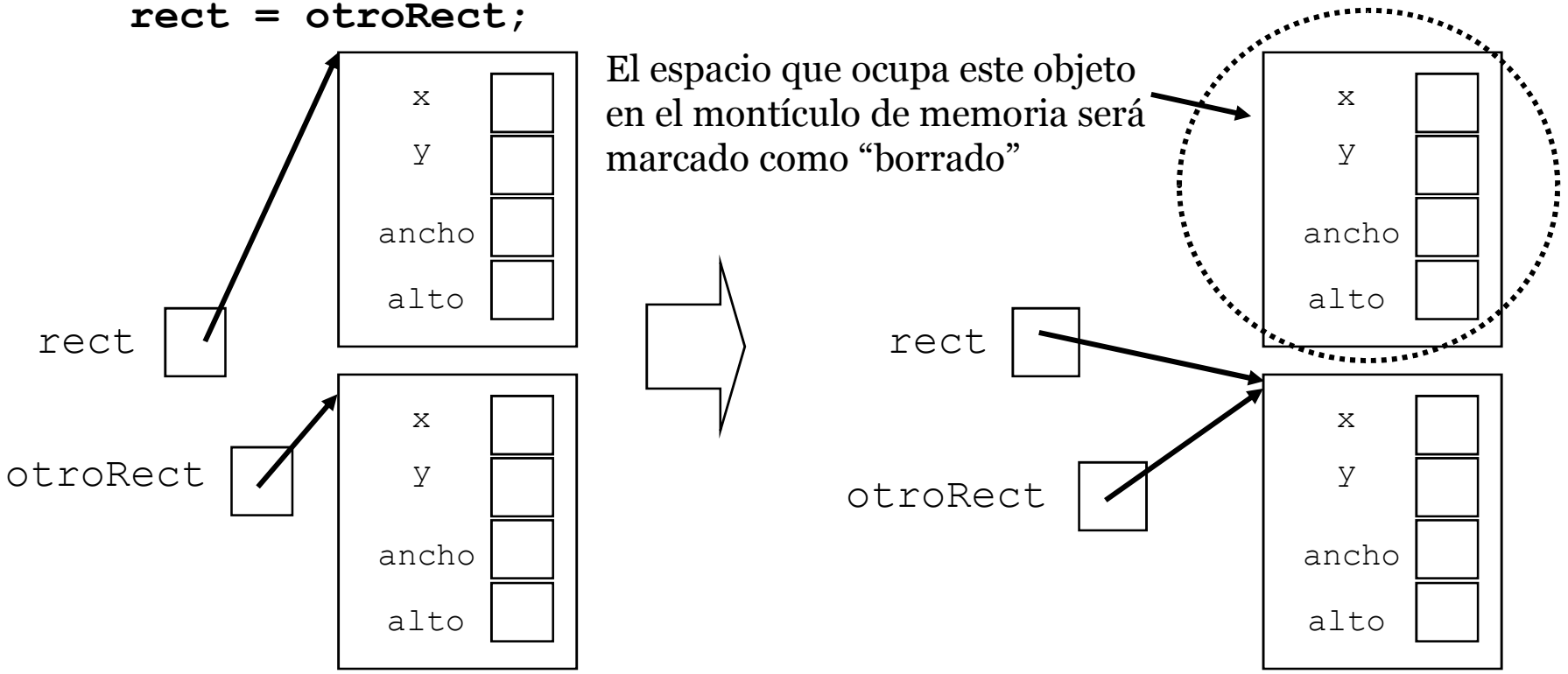
Destrucción de objetos

- Los objetos no se destruyen explícitamente sino que basta con "*olvidarlos*" (perder toda *referencia* a ellos)
- Java dispone de un recolector automático de basura (*automatic garbage collector*)
  - Cuando detecta que un objeto no es accesible desde ningún punto del programa, marca el espacio que ocupa como "borrado"
    - Es capaz de detectar también una cadena de referencias circulares que no sean accesibles desde el programa
  - Puede entrar en acción en cualquier momento de la ejecución del programa, el programador no ejerce ningún control sobre él
- Por todo esto, las clases no necesitan destructores
  - En casos especiales (para asegurar la liberación de recursos como ficheros o sockets) se puede sobreescribir un método llamado **`finalize`** que es llamado cuando va a destruirse un objeto

Recolector automático de basura

```
Rectangulo rect = new Rectangulo();
Rectangulo otroRect = new Rectangulo();

// Se pierde la referencia al primer rectángulo
rect = otroRect;
```



Herencia entre clases

- En Java la herencia es una relación *es-un* entre una superclase (o clase padre) y sus subclases (o clases hijas)
  - No existe herencia de implementación *privada*, como en C++
  - No existe herencia *múltiple* de clases (sólo se puede heredar de una)
- Constructores de una subclase
  - Al crear un ejemplar de una subclase, antes de ejecutar su constructor se llama automáticamente al constructor por defecto de la superclase
  - Se puede llamar expresamente a un constructor concreto de la superclase usando la referencia **`super`**
    - Desde la primera línea del constructor de la subclase, obligatoriamente
- Sobreescritura de métodos heredados
  - Por defecto todo método puede sobreescribirse si no se declara como de tipo **`final`**
  - Desde un método se puede llamar a otros métodos de la superclase usando la misma referencia de antes: **`super`**

Ejemplo de herencia

```
public class Empleado {
  protected String _nombre;
  protected String _DNI;
  protected float _sueldoHora;
  protected int _horasTrabajadas;
  public Empleado(String nombre, String DNI, float sueldoHora){
    _nombre = nombre;
    _DNI = DNI;
    _sueldoHora = sueldoHora;
  }
  public float pagar(){
    return _sueldoHora * _horasTrabajadas;
  }
}
public class Director extends Empleado {
  private float _primas;
  …
  public Director(String nombre, String DNI, float sueldoHora, float primas){
    super(nombre, DNI, sueldoHora);
    _primas = primas;
  }
 public float pagar(){
    float debido = super.pagar();
    return debido + _primas;
  }
}
```

Clases abstractas

- Deben declararse como abstractas aquellas clases que tengan al menos un método abstracto
  - Un método abstracto es aquél que se deja sin implementar
  - Tanto el método como la clase se marcan con la palabra reservada **`abstract`**
- No se pueden crear ejemplares (objetos) de clases abstractas
- La subclase de una clase abstracta debe sobreescribir todos los métodos abstractos o ser declarada a su vez como abstracta

```
public abstract class Figura {
  private int _x, _y;
  private int _color;
  public abstract void dibuja();
  public void setColor(int c) {…};
}


public class Circulo extends Figura {
  private int _radio;
  public void dibuja() {…};
}
```

Interfaces

- Equivalentes a las *clases virtuales puras* de C++
  - Todos los métodos están "declarados" pero no implementados
  - Una interfaz (**`interface`**) puede heredar de otra usando **`extends`** (pudiendo haber herencia múltiple entre interfaces)
- Una clase puede implementar (**`implements`**) una o varias interfaces, debiendo entonces implementar todos sus métodos

```
interface NombreInterfaz {
        // Declaración de constantes …

        // Declaración de métodos
        public tipoDevuelto nombreMetodo1(listaArgumentos)
}

class NombreClase implements NombreInterfaz1,NombreInterfaz2 {
  // Declaración de atributos y métodos de la clase …

  // Implementación de los métodos de las interfaces
  public tipoDevuelto nombreMetodo1(listaArgumentos) {…};
}
```

La clase Object

- Por defecto todas las clases en Java heredan de la clase **`java.lang.Object`**
  - **`Object`** es la raíz de toda la jerarquía de clases
  - Todos los objetos pueden por tanto tratarse como de tipo **`Object`**
- **`Object`** define un conjunto de métodos muy útiles
  - **`public boolean equals(Object o)`**
    - Compara el objeto con otro
    - Por defecto se comparan directamente las referencias (es la *identidad*)
  - **`public String toString()`**
    - Representa el objeto en forma de cadena de texto
    - Por defecto se compone así la cadena:
      `getClass().getName() + '@' + Integer.toHexString(hashCode())`
  - **`protected Object clone()`**
    - Realiza una copia del objeto
    - Por defecto hace copias superficiales y no "profundas" (copiar *recursivamente* los atributos que sean a su vez objetos)

Comparación entre objetos

- El operador == no vale
  - Sólo compara referencias (siempre es la *identidad*)
- Para poder comparar, la clase debe proporcionar el método **`boolean equals(Object o)`** y:
  1. Comprobar si el parámetro es distinto de **`null`**
  2. Comprobar si el parámetro es el mismo que **`this`**
  3. Comprobar si el parámetro es de la misma clase
  4. Comprobar la igualdad atributo a atributo

Ejemplo de comparación

```
public class A {
  private int _n;
  private B _b;

  public boolean equals (Object o) {
    if (o == null)
      return false;
    if (o == this)
      return true;
    if (!(o instanceof A))
      return false;
    A a = (A)o;
    return (this._n == a._n) &&
           (this._b.equals(a._b));
  }
}
```

Copia de objetos

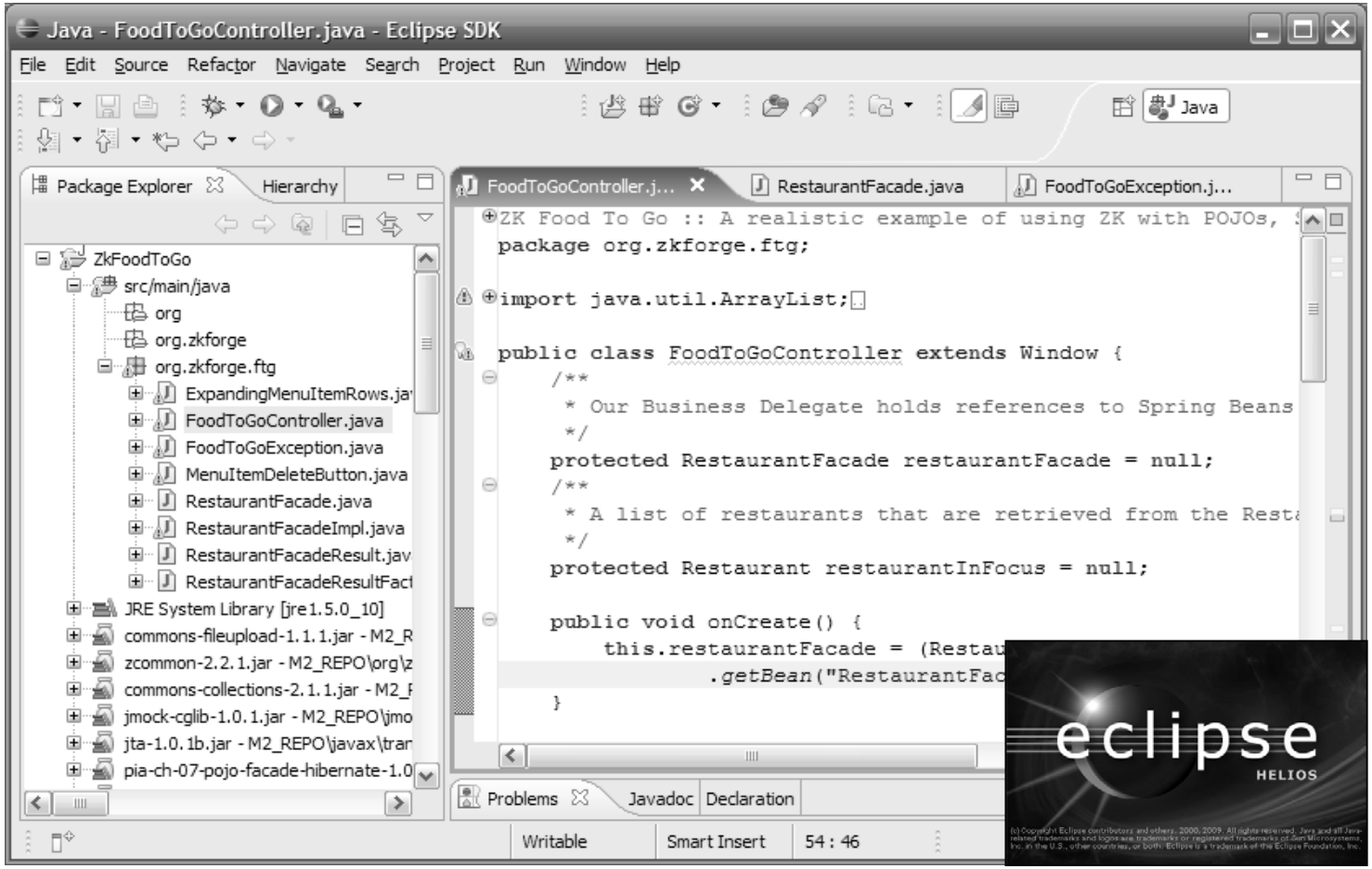
- El operador == no vale
  - Sólo copia referencias
- Para poder ser copiada la clase debe implementar el interfaz **`Cloneable`**, implementando el método **`Object clone ()`**
  - Internamente la copia no se hace con **`new`**, sino invocando al método **`clone`** de la superclase
    - Esta puede generar una excepción que hay que capturar (e ignorar si se produce, habitualmente)
- La implementación de **`clone`** de **`Object`** hace una *copia bit a bit del objeto* y la devuelve
  - Si queremos hacer una *copia profunda*, nuestras implementaciones de **`clone`** deberán copiar todos los atributos que sean de tipo objeto (salvo si son constantes o inmutables de una copia a otra por alguna razón)

Ejemplo de copia

```
public class A implements Cloneable {
  private int _n;
  private B _b;

  public Object clone() {
    Object ret;
    try {
      ret = super.clone();
      A copia = (A)ret;
      copia._b = (B)this._b.clone();
    }
    catch (CloneNotSupportedException e) {
        …
    }
  }
}
```

Eclipse



http://www.eclipse.org

Eclipse IDE for Java Developers

- **Eclipse** es una sofisticada tecnología para desarrollar Entornos de Desarrollo Integrado (IDEs) multiplataforma
  - Comenzó como un proyecto de IBM, cuyo código fue liberado en 2001 y totalmente independizado en 2003 gracias a la fundación Eclipse
- **Eclipse IDE for Java Developers** es una distribución concreta de Eclipse para desarrollar aplicaciones Java
  - Hay otras distribuciones con distintas versiones cada una, pero nosotros usaremos esta en su versión 3.6 (alias "Helios", 2010) http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/heliossr1
  - Facilita el trabajo del programador, integrando muchas herramientas (JavaDoc, JUnit, Ant…) y ofreciendo:
    - Resaltado de sintaxis
    - Compilación en tiempo de edición (para detectar y corregir errores)
    - Asistentes (*wizards*) para crear proyectos, clases, pruebas, etc.
    - Refactorización del código

Críticas, dudas, sugerencias…

Federico Peinado
www.federicopeinado.es