Lenguaje Java Avanzado

Sesión 1: Introducción al Lenguaje Java

Índice

- Introducción a Java
- Conceptos de POO
- Elementos de un programa Java
- Herencia, interfaces, polimorfismo
- Hilos
- Clases útiles

Java

- *Java* es un lenguaje OO creado por *Sun Microsystems* para poder funcionar en distintos tipos de procesadores y máquinas.
- Similar a C o C++, pero con algunas características propias (gestión de hilos, ejecución remota, etc)
- Independiente de la plataforma, gracias a la JVM (*Java Virtual Machine*), que interpreta los ficheros objeto
- Se dispone de antemano de la API (*Application Programming Interface*) de clases de Java.

Clases

- *Clases*: con la palabra *class* y el nombre de la clase

```
class MiClase
{
   ...
}
```

- Como nombre utilizaremos un sustantivo
- Puede estar formado por varias palabras
- Cada palabra comenzará con mayúscula, el resto se dejará en minúscula
  - Por ejemplo: `DataInputStream`
- Si la clase contiene un conjunto de métodos estáticos o constantes relacionadas pondremos el nombre en plural
  - Por ejemplo: `Resources`

Campos y variables

- *Campos y variables*: simples o complejos
- Utilizaremos sustantivos como nombres

```
Properties propiedades;
File ficheroEntrada;
int numVidas;
```

- Puede estar formado por varias palabras, con la primera en minúsculas y el resto comenzando por mayúsculas y el resto en minúsculas
  - Por ejemplo: `numVidas`
- En caso de tratarse de una colección de elementos, utilizaremos plural
  - Por ejemplo: `clientes`
- Para variables temporales podemos utilizar nombres cortos, como las iniciales de la clase a la que pertenezca, o un carácter correspondiente al tipo de dato

```
int i;
Vector v;
DataInputStream dis;
```

Constantes

- *Constantes*: Se declarán como *final* y *static*

```
final static String TITULO_MENU = “Menu”;
final static int ANCHO_VENTANA = 640;
final static double PI = 3.1416;
```

- El nombre puede contener varias palabras
- Las palabras se separan con ‘_’
- Todo el nombre estará en mayúsculas
  - Por ejemplo: `MAX_MENSAJES`

Métodos

- *Métodos*: con el tipo devuelto, nombre y parámetros

```
void imprimir(String mensaje)
{
   ...// Codigo del método
}
Vector insertarVector(Object elemento, int posicion)
{
   ...// Codigo del método
}
```

- Los nombres de los métodos serán verbo
- Puede estar formados por varias palabras, con la primera en minúsculas y el resto comenzando por mayúsculas y el resto en minúsculas
  - Por ejemplo: `imprimirDatos`

Constructores

- *Constructores*: se llaman igual que la clase, y se ejecutan con el operador *new* para reservar memoria

```
MiClase()
{
   ...// Codigo del constructor
}
MiClase(int valorA, Vector valorV)
{
   ...// Codigo del otro constructor
}
```

- No hace falta destructor, de eso se encarga el *garbage collector*
- Constructor superclase: `super(…)`

Paquetes

- *Paquetes*: organizan las clases en una jerarquía de paquetes y subpaquetes
- Para indicar que una clase pertenece a un paquete o subpaquete se utiliza la palabra *package* al principio de la clase

```
package paquete1.subpaquete1;
class MiClase {

```

- Para utilizar clases de un paquete en otro, se colocan al principio sentencias *import* con los paquetes necesarios:

```
package otropaquete;
import paquete1.subpaquete1.MiClase;
import java.util.*;
class MiOtraClase {

```

Paquetes

- Si no utilizamos sentencias *import*, deberemos escribir el nombre completo de cada clase del paquete no importado (incluyendo subpaquetes)

```
class MiOtraClase {
   paquete1.subpaquete1.MiClase a = ...;    // Sin import
   MiClase a = ...;                 // Con import
```

- Los paquetes se estructuran en directorios en el disco duro, siguiendo la misma jerarquía de paquetes y subpaquetes

```
./paquete1/subpaquete1/MiClase.java
```

Paquetes

- Siempre se deben incluir las clases creadas en un paquete
  - Si no se especifica un nombre de paquete la clase pertenecerá a un paquete “sin nombre”
  - No podemos importar clases de paquetes “sin nombre”, las clases creadas de esta forma no serán accesibles desde otros paquetes
  - Sólo utilizaremos paquetes “sin nombre” para hacer una prueba rápida, nunca en otro caso

Convenciones de paquetes

- El nombre de un paquete deberá constar de una serie de palabras simples siempre en minúsculas
  - Se recomienda usar el nombre de nuestra DNS al revés
    `especialistajee.org` → `org.especialistajee.prueba`
- Colocar las clases interdependientes, o que suelan usarse juntas, en un mismo paquete
- Separar clases volátiles y estables en paquetes diferentes
- Hacer que un paquete sólo dependa de paquetes más estables que él
- Si creamos una nueva versión de un paquete, daremos el mismo nombre a la nueva versión sólo si es compatible con la anterior

Tipo enumerado

```
enum EstadoCivil {soltero, casado, divorciado};

EstadoCivil ec = EstadoCivil.casado;
ec = EstadoCivil.soltero;

switch(ec) {
  case soltero:
    System.out.println("Es soltero"); break;
  case casado:
    System.out.println("Es casado"); break;
  case divorciado:
    System.out.println("Es divorciado"); break;
}
```

Otras características

- Imports estáticos

```
import static java.lang.Math;
…
double raiz = sqrt(1252.2);
```

- Argumentos variables

```
public void miFunc(String param, int... args) {
    for(int i: args) { … }
}
```

- Anotaciones (metainformación)
  - P.ej., `@deprecated`

Convenciones generales

- Indentar el código uniformemente
- Limitar la anchura de las líneas de código (para impresión)
- Utilizar líneas en blanco para separar bloques de código
- Utilizar espacios para separar ciertos elementos en una línea
- Documentación:
  - Utilizar `/* ... */` para esconder código sin borrarlo
  - Utilizar `// ...` para detalles de la implementación
  - Utilizar javadoc para describir la interfaz de programación

Modificadores de acceso

- Las clases y sus elementos admiten unos modificadores de acceso:
  - *privado*: el elemento es accesible sólo desde la clase en que se encuentra
  - *protegido*: el elemento es accesible desde la propia clase, desde sus subclases, y desde clases del mismo paquete
  - *público*: el elemento es accesible desde cualquier clase
  - *paquete*: el elemento es accesible desde la propia clase, o desde clases del mismo paquete.

Modificadores de acceso

- *private* se utiliza para elementos PRIVADOS
- *protected* se utiliza para elementos PROTEGIDOS
- *public* se utiliza para elementos PUBLICOS
- No se especifica nada para elementos PAQUETE

```
public class MiClase {
   private int n;
   protected void metodo() { ... }
```

- Todo fichero Java debe tener una y solo una clase pública, llamada igual que el fichero (más otras clases internas que pueda tener)

Modificadores de acceso

	La misma clase	Cualquier clase del mismo paquete	Subclase de otro paquete	Cualquier clase de otro paquete
public	sí	sí	sí	sí
protected	sí	sí	sí	
default	sí	sí		
private	sí			

Otros modificadores

- *abstract*: para definir clases y métodos abstractos
- *static*: para definir elementos compartidos por todos los objetos que se creen de la misma clase
  - miembros que no pertenecen al objeto en si, sino a la clase
  - dentro de un método estático sólo podemos utilizar elementos estáticos, o elementos que hayamos creado dentro del propio método
- *final*: para definir elementos no modificables ni heredables

```
public abstract class MiClase {
   public static final int n = 20;
   public abstract void metodo();
   ...
```

Otros modificadores

- *volatile* y *synchronized*: para elementos a los que no se puede acceder al mismo tiempo desde distintos hilos de ejecución
  - *volatile* no proporciona atomicidad pero es más eficiente

```
volatile int contador;
contador++; //puede causar problemas, son 3 operaciones diferentes
```

  - *synchronized* se usa sobre bloques de código y métodos

```
synchronized(this){
    contador++;
}
```

Otros modificadores

- *native*: para métodos que están escritos en otro lenguaje, por ejemplo en C++, utilizando JNI (Java Native Interface)
- *transient*: para atributos que no forman parte de la persistencia de objeto, para evitar que se serialicen
- *strictfp*: evitar que se utilice toda la precisión de punto flotante que proporcione la arquitectura. Usar el estándar del IEEE para float y double. No es aconsejable a menos que sea necesario.

Herencia y polimorfismo

- **Herencia**: definir una clase a partir de otra existente
  - La nueva clase “hereda” todos los campos y métodos de la clase a partir de la que se crea, y aparte puede tener los suyos propios
  - *Ejemplo*: a partir de una clase *Animal* podemos definir otras más concretas como *Pato, Elefante…*
- **Polimorfismo**: si tenemos un método en cualquier clase que sea *dibuja (Animal a)*, podemos pasarle como parámetro tanto un objeto *Animal* como cualquier subtipo que herede directa o indirectamente de él (*Elefante, Pato…*)

Clases abstractas e interfaces

- Una *clase abstracta* es una clase que deja algunos métodos sin código, para que los rellenen las subclases que hereden de ella

```
public abstract class MiClase {
   public abstract void metodo1();
   public void metodo2() {
       ...
   }
}
```

- Un *interfaz* es un elemento que sólo define la cabecera de sus métodos, para que las clases que implementen dicha interfaz rellenen el código según sus necesidades.

```
public interface Runnable {
   public void run();
}
```

- Asignaremos un nombre a los interfaces de forma similar a las clases, pudiendo ser en este caso adjetivos o sustantivos.

Herencia e interfaces

- Herencia
  - Definimos una clase a partir de otra que ya existe
  - Utilizamos la palabra *extends* para decir que una clase hereda de otra (Pato *hereda de* Animal):

```
class Pato extends Animal
```

  - Relación “es”: Un pato *ES* un animal
- Interfaces
  - Utilizamos la palabra *implements* para decir que una clase implementa los métodos de una interfaz

```
class MiHilo implements Runnable {
   public void run() {
       ... // Codigo del método
   }
```

  - Relación “actúa como”: MiHilo *ACTÚA COMO* ejecutable

Polimorfismo

- Si una variable es del tipo de la superclase, podemos asignarle también un objeto de la clase hija

```
Animal a = new Pato();
```

- Si una variable es del tipo de una interfaz implementada por nuestra clase, podemos asignarle también un objeto de esta clase

```
Runnable r = new MiHilo();
```

- Sólo se puede heredar de una clase, pero se pueden implementar múltiples interfaces:

```
class Pato extends Animal implements Runnable, ActionListener
```

Punteros *this* y *super*

- *this* se utiliza para hacer referencia a los elementos de la propia clase:

```
class MiClase {
   int i;
   MiClase(int i) {
       this.i = i;    // i de la clase = i del parámetro
```

- *super* se utiliza para llamar al mismo método en la superclase:

```
class MiClase extends OtraClase{
   MiClase(int i) {
       super(i);      // Constructor de OtraClase(...)
```

Object

- Clase base de todas las demás
  - Todas las clases heredan en última instancia de ella

- Es importante saber las dependencias (herencias, interfaces, etc) de una clase para saber las diferentes formas de instanciarla o referenciarla (polimorfismo)

Ejemplo de polimorfismo

- Por ejemplo, si tenemos:

```
public class MiClase extends Thread implements List{

```

- Podremos referenciar un objeto *MiClase* de estas formas:

```
MiClase mc = new MiClase();
Thread t = new MiClase();
List l = new MiClase();
Object o = new MiClase();
```

Hilos

- Cada hilo es un flujo de ejecución independiente
- Tiene su propio contador de programa
- Todos acceden al mismo espacio de memoria
- Necesidad de sincronizar cuando se accede concurrentemente a los recursos
- Existen estructuras de datos sincronizadas (ej, Vector) y sin sincronizar (ej, ArrayList)

Creación de Hilos

- Se pueden crear de dos formas:
  - Heredando de Thread
    - Problema: No hay herencia múltiple en Java
  - Implementando Runnable
- Debemos crear sólo los hilos necesarios
  - Dar respuesta a más de un evento simultáneamente
  - Permitir que la aplicación responda mientras está ocupada
  - Aprovechar máquinas con varios procesadores

Heredar de Thread

- Heredar de Thread y sobrecargar run( )

```
public class MiHilo extends Thread {
    public void run() {
        // Codigo de la tarea a ejecutar en el hilo
    }
}
```

- Instanciar el hilo

```
Thread t = new Thread(new MiHilo());
t.start();
```

Implementar Runnable

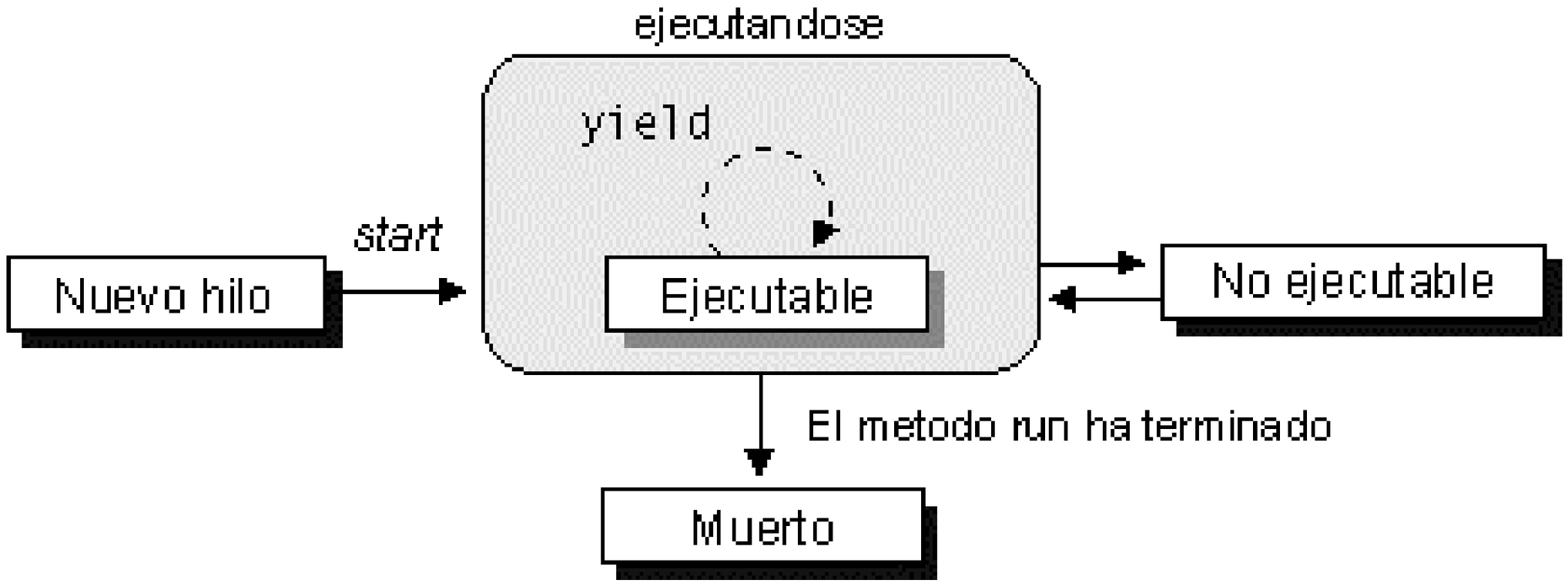
- Implementar Runnable

```
public class MiHilo implements Runnable {
    public void run() {
        // Codigo de la tarea a ejecutar en el hilo
    }
}
```

- Instanciar el hilo

```
Thread t = new Thread(new MiHilo());
t.start();
```

Ciclo de vida de los hilos



- El hilo será no ejecutable cuando:
- Se encuentre durmiendo (llamando a `sleep`)
- Se encuentre bloqueado (con `wait`)
- Se encuentre bloqueado en una petición de E/S

Scheduler

- El *scheduler* decide qué hilo ejecutable ocupa el procesador en cada instante
- Se sacará un hilo del procesador cuando:
  - Se fuerce la salida (llamando a `yield`)
  - Un hilo de mayor prioridad se haga ejecutable
  - Se agote el *quantum* del hilo
- Establecemos la prioridad con

```
t.setPriority(prioridad);
```

  - La prioridad es un valor entero entre `Thread.MIN_PRIORITY y Thread.MAX_PRIORITY`

Concurrencia y sección crítica

- Cuando varios hilos acceden a un mismo recurso pueden producirse problemas de concurrencia
- *Sección crítica*: Trozo del código que puede producir problemas de concurrencia
- Debemos sincronizar el acceso a estos recursos
  - Este código no debe ser ejecutado por más de un hilo simultáneamente
- Todo objeto Java (`Object`) tiene una variable cerrojo que se utiliza para indicar si ya hay un hilo en la sección crítica
  - Los bloques de código `synchronized` utilizarán este cerrojo para evitar que los ejecute más de un hilo

Métodos sincronizados

- Sincronzar un método o una sección de código

```
public synchronized void seccion_critica() {
    // Codigo
}
```

- Se utiliza el cerrojo del objeto en el que se definen
  - Se podrán ejecutar por un sólo hilo en un instante dado.

- Deberemos utilizar la sincronización sólo cuando sea necesario, ya que reduce la eficiencia
- No sincronizar métodos que contienen un gran número de operaciones que no necesitan sincronización
  - Reorganizar en varios métodos
- No sincronizar clases que proporcionen datos fundamentales
  - Dejar que el usuario decida cuando sincronizarlas en sus propias clases

Bloqueo de hilos

- Si el hilo va a esperar a que suceda un evento (por ejemplo, terminar una E/S), hay que bloquearlo para que no ocupe el procesador:

```
wait();
```

- Cuando suceda el evento debemos desbloquearlo desde otro hilo con:

```
notify();
```

- Ambos métodos deben ser invocados desde métodos sincronizados

- Dependencia de hilos: podemos esperar a que un hilo haya acabado de ejecutarse para poder continuar otro hilo
- Para ello bloquearemos el hilo actual que debe esperar a otro hilo `t` con:

```
t.join();
```

Object: objetos diferentes

- También es importante distinguir entre entidades independientes y referencias:

```
MiClase mc1 = new MiClase();
MiClase mc2 = mc1;
// Es distinto a:
MiClase mc2 = (MiClase)(mc1.clone());
```

- El método *clone* de cada objeto sirve para obtener una copia en memoria de un objeto con los mismos datos, pero con su propio espacio
  - No realiza una copia en profundidad
  - Si queremos hacer copias de los objetos que tenga como campos debe sobrescribir este método

Object: comparar objetos

- Cuando queremos comparar dos objetos entre sí (por ejemplo, de la clase *MiClase*), no se hace así: `if (mc1 == mc2)`
- Sino con su método *equals*: `if (mc1.equals(mc2))`

- Deberemos redefinir este método en las clases donde lo vayamos a usar, para asegurarnos de que los objetos se comparan bien
  - Notar que la clase *String*, es un subtipo de *Object* por lo que para comparar cadenas...:

```
if (cadena == “Hola”) ...       // NO
if (cadena.equals(“Hola”)) ... // SI
```

Object: representar en cadenas

- Muchas veces queremos imprimir un objeto como cadena. Por ejemplo, si es un punto geométrico, sacar su coordenada X, una coma, y su coordenada Y
- La clase *Object* proporciona un método *toString* para definir cómo queremos que se imprima un objeto. Podremos redefinirlo a nuestro gusto

```
public class Punto2D {
    ...
    public String toString()
    {
        return “(“ + x + ”,” + y + ”)”;
    }
}
...
Punto2D p = ...;
System.out.println(p);  // Imprimirá (x, y) del punto
```

Properties

- Esta clase es un tipo de tabla *hash* que almacena una serie de propiedades, cada una con un valor asociado
- Además, permite cargarlas o guardarlas en algún dispositivo (fichero)
- Algunos métodos interesantes:

```
Object setProperty(Object clave, Object valor)
Object getProperty(Object clave)
Object getProperty(Object clave, Object default)

void load(InputStream entrada)
void store(OutputStream salida, String cabecera)
```

System

- Ofrece métodos y campos útiles del sistema, como el ya conocido *System.out.println*
- Otros métodos interesantes de esta clase (todos estáticos):

```
void exit(int estado)
void gc()
long currentTimeMillis()
void arrayCopy(Object fuente, int pos_fuente,
               Object destino, int pos_destino,
               int numElementos)
```

Otras clases

- La clase *Math* proporciona una serie de métodos (estáticos) útiles para diferentes operaciones matemáticas (logaritmos, potencias, exponenciales, máximos, mínimos, etc)
- Otras clases útiles son la clase *Calendar* (para trabajar con fechas y horas), la clase *Currency* (para monedas), y la clase *Locale* (para situarnos en las características de fecha, hora y moneda de una región del mundo)

Transfer Objects

- Encapsulan datos con los que normalmente se trabaja de forma conjunta
- Nos permiten transferir estos datos entre las diferentes capas de la aplicación

```
public class Usuario {
  private String login;
  private String password;
  private boolean administrador;
}
```

Getters y Setters

- Es buena práctica de programación declarar todos los campos de las clases privados
- Para acceder a ellos utilizaremos métodos
  - *Getters* para obtener el valor del campo
  - *Setters* para modificar el valor del campo
- Estos métodos tendrán prefijo `get` y `set` respectivamente, seguido del nombre del campo al que acceden, pero comenzando por mayúscula
  - Por ejemplo: `getLogin()`, `setLogin(String login)`
- El *getter* para campos booleanos tendrá prefijo `is` en lugar de `get`
  - Por ejemplo: `isAdministrador()`

BeanUtils

- Utilidades de la biblioteca commons-beanutils de Apache.
- BeanUtils.copyProperties(objDestino, objOrigen)
- Copia los campos comunes entre los dos objetos
- Los reconoce usando la API de Reflection
- La identificación está basada en los nombres de los getters y los setters y en su tipo de dato.
- Ejemplo: **int** origen.get**NombreCampo**( ), e void destino.set**NombreCampo**(**int** n).

BeanUtils

- Ejemplo: proyección de un punto3D en un punto2D.
- En lugar de copiar todos los campos uno a uno:

```
punto2D.setX(punto3D.getX());
punto2D.setY(punto3D.getY());
punto2D.setDescripcion(punto3D.getDescripcion());
```

- usamos copyProperties:

```
BeanUtils.copyProperties(punto2D, punto3D);
```

```
public class Punto2D {
 private int x;
 private int y;
 private String descripcion;

 public String getDescripcion() {
    return descripcion;
 }
 public void setDescripcion(String
descripcion) {
    this.descripcion = descripcion;
 }
 public int getX() {
    return x;
 }
 public void setX(int x) {
    this.x = x;
 }
 public int getY() {
    return y;
 }
 public void setY(int y) {
    this.y = y;
 }
}
```

```
public class Punto3D {
 private int x;
 private int y;
 private int z;
 private String descripcion;
 /* ...y los getters y setters para los cuatro
campos */
}
```

¿Preguntas...?