Curso de Java
POO: Programación orientada a objetos

Luis Guerra Velasco

Curso INEM 02830. Programación en Java

Marzo 2010

Índice

Índice

Paradigma

- Un paradigma es una forma de afrontar la construcción de código software
  - No hay paradigmas mejores ni peores
  - Todos tienen sus ventajas e inconvenientes
- Hay distintos paradigmas:
  - POO, Estructurado, Funcional, Lógico, etc

Características de la POO

- Facilidad de diseño y relación con el mundo real (UML)
- Reusabilidad y facilidad de mantenimiento
- Sistemas más complejos
  - Abstracción
  - Trabajo en equipo
- Del lenguaje máquina hacia el mundo real
- Resuelve problemas complicados. No está pensado para tareas sencillas

UML

- UML (Unified Modeling Language): Lenguaje unificado de modelos
- "Mapa" del código. No sirve para desarrollar, sino para describir
- Se utilizan diferentes diagramas. 13 en UML 2.0

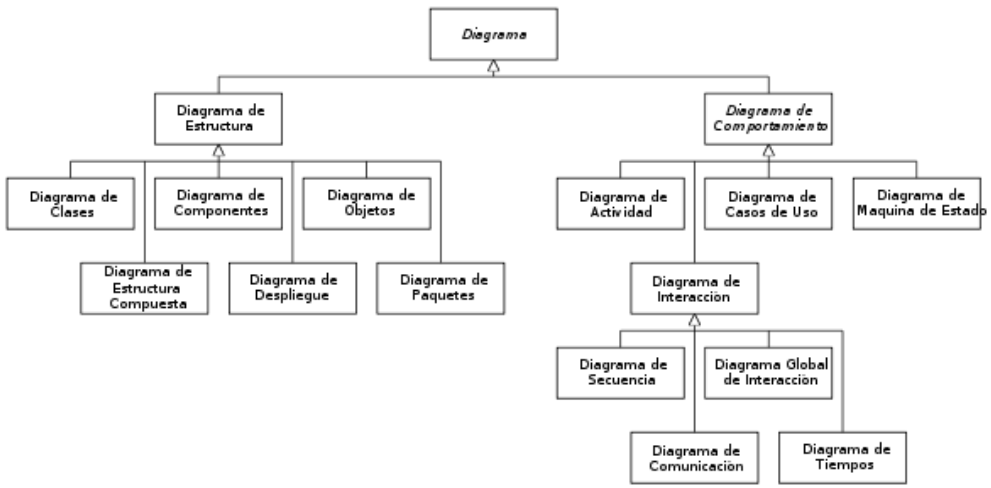


Figura: Jerarquía de diagramas UML 2.0

Elementos de la POO

- Los elementos principales son:
  - clases: Especificación de un conjunto de elementos
  - objetos: Elemento autónomo y con una funcionalidad concreta. Instancias concretas de una clase

Elementos de la POO

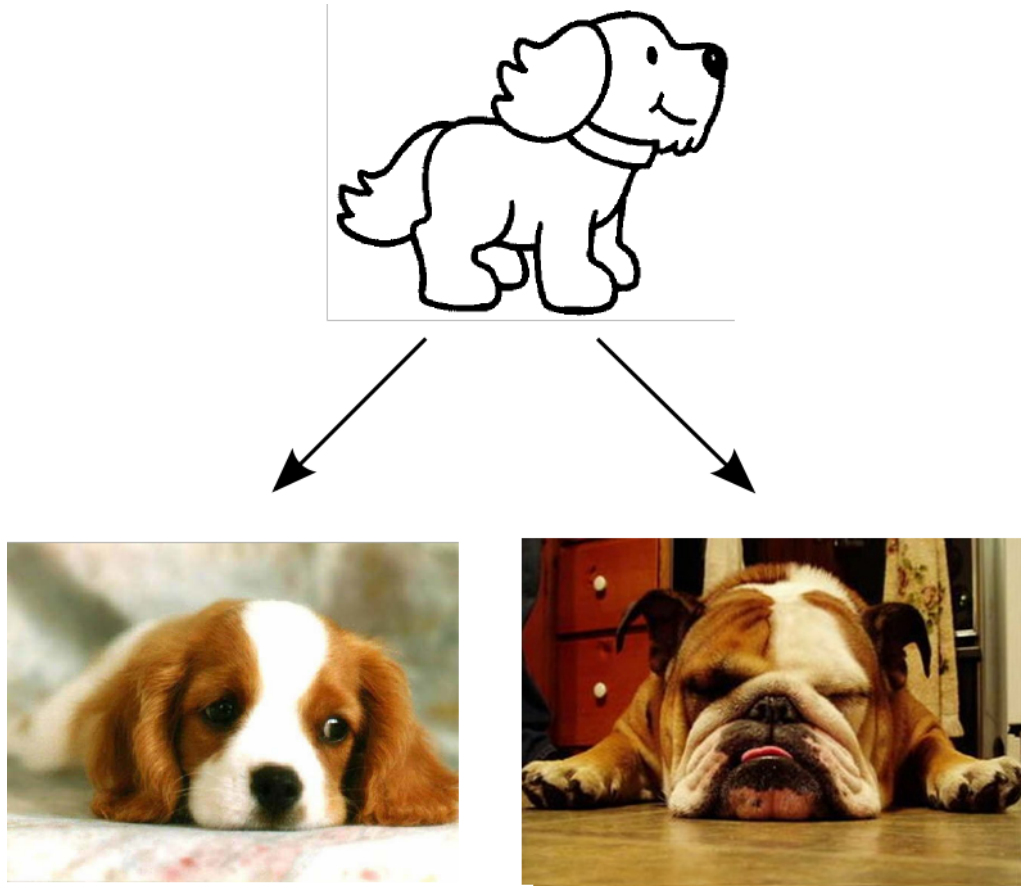
- Los elementos principales son:
  - clases: Especificación de un conjunto de elementos
  - objetos: Elemento autónomo y con una funcionalidad concreta. Instancias concretas de una clase
- También se basa en otros conceptos: herencia, polimorfismo y encapsulamiento

Perros. Objetos y Clases

Perros. Objetos y Clases

Perros. Objetos y Clases

Perros. Objetos y Clases

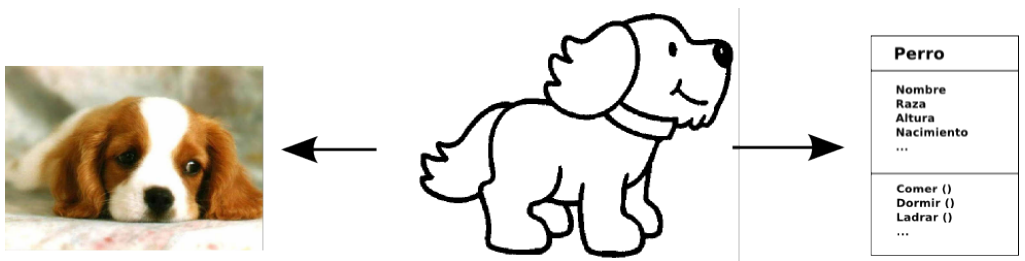


Resumen de objetos y clases

- Objetos
  - Elementos con comportamiento y estado. Métodos y atributos concretos
  - Instancias de clase
  - Interactuan por medio de mensajes
- Clases
  - Plantillas para definir elementos (objetos)
  - Pueden estar directamente relacionadas unas con otras

Resumen de objetos y clases

- Objetos
  - Elementos con comportamiento y estado. Métodos y atributos concretos
  - Instancias de clase
  - Interactuan por medio de mensajes
- Clases
  - Plantillas para definir elementos (objetos)
  - Pueden estar directamente relacionadas unas con otras



Ejemplo de plantilla

```
public class NombreClase {

    /**
     * Atributos:
     * tipo nombreAtributo1;
     * tipo nombreAtributo2;
     */

    /**
     * Metodos:
     * salida nombreMetodo1 (parametros){ contenido del
         metodo};
     * salida nombreMetodo2 (parametros){ contenido del
         metodo};
     */

}
```

Ejemplo de clase Perro

Ejemplo de clase Perro

```
public class Perro {

    String nombre;
    String raza;
    float altura;

    void perro(){}
    void comer(){}
    void dormir(){}
    void ladrar(){}
}
```

Encapsulación

Puede (y suele) haber distintos niveles de visibilidad:

- public: se puede acceder desde cualquier lugar
- private: sólo se puede acceder desde la propia clase
- protected: sólo se puede acceder desde la propia clase o desde una clase que herede de ella

De esta forma se controla qué cosas son modificables y cómo se pueden modificar El estado suele ser privado, y se suele modificar a través del comportamiento

Ejemplo de clase Perro con visibilidad

Ejemplo de clase Perro con visibilidad

```
public class Perro {

    private String nombre;
    private String raza;
    private float altura;

    public void perro(){}
    public void comer(){}
    public void dormir(){}
    public void ladrar(){}
}
```

Operadores

Operador "."

Acceso a miembros de una clase:

```
perro1.nombre = "Pancho";
perro1.ladrar();
```

Operadores

Operador "."

Acceso a miembros de una clase:

```
perro1.nombre = "Pancho";
perro1.ladrar();
```

Si soy la clase Dueño...

Operadores

Operador "."

Acceso a miembros de una clase:

```
perro1.nombre = "Pancho";
perro1.ladrar();
```

Si soy la clase Dueño... ¡Cuidado con la visibilidad!

Operadores

Operador "."

Acceso a miembros de una clase:

```
perro1.nombre = "Pancho";
perro1.ladrar();
```

Si soy la clase Dueño... ¡Cuidado con la visibilidad!

Operador "this"

Acceso a atributos dentro de la propia clase:

```
this.nombre = "Pancho";
```

Métodos

- Definen el comportamiento de los objetos de una clase
- La implementación se suele ocultar al exterior de la clase

Estructura

- Cabecera: *modificadores - tipoSalida - nombre - (parámetros)*
- Cuerpo: Se define la implementación

Métodos habituales

Constructor

- Sirve para inicializar un objeto al crearlo
- Existe sobrecarga (distintos parámetros) (para cualquier método)
- Coincide con el nombre de la clase y no devuelve nada por definición

Métodos habituales

Constructor

- Sirve para inicializar un objeto al crearlo
- Existe sobrecarga (distintos parámetros) (para cualquier método)
- Coincide con el nombre de la clase y no devuelve nada por definición

Get & Set

- Sirven para obtener o para modificar los atributos de una clase

Ejemplos de métodos típicos

```
public class Perro {
    private String nombre;
    ...
    public void Perro(String nombre){
        this.nombre = nombre;
    }
    public String getNombre(){
        return this.nombre;
    }
    public void setNombre(String nombre){
        this.nombre = nombre
    }
}
```

Métodos típicos

Destructor

- No es tan típico, no se suele usar (se hace de manera automática)
- Se tiene que sobreescribir
- No devuelve nada por definición

Métodos típicos

Destructor

- No es tan típico, no se suele usar (se hace de manera automática)
- Se tiene que sobreescribir
- No devuelve nada por definición

finalize

```
protected void finalize()
```

Instanciación de objetos

Antes de poder usar un objeto tenemos que crearlo:

Nuevo operador

Tipo identificador = **new** *Tipo()*

Ejemplo con Perro

```
Perro miPancho = new Perro("Pancho")
```

Instanciación de objetos

Antes de poder usar un objeto tenemos que crearlo:

Nuevo operador

Tipo identificador = **new** *Tipo()*

Ejemplo con Perro

```
Perro miPancho = new Perro("Pancho")

Perro miPancho = new Perro("Pancho", "Cocker", 40)
```

Uso de objetos

- Una vez tenemos el objeto instanciado...
- ...podemos modificar su estado usando los métodos

```
Perro miPancho = new Perro("Pancho", "Cocker", 40)
```

Uso de objetos

- Una vez tenemos el objeto instanciado...
- ...podemos modificar su estado usando los métodos

```
Perro miPancho = new Perro("Pancho", "Cocker", 40)

miPancho.setNombre("Toby")
miPancho.dormir()
```

Tipos de variables

- De instancia: Definida para las instancias de una clase. Una copia por objeto

Tipos de variables

- De instancia: Definida para las instancias de una clase. Una copia por objeto
- De clase: Definida para la clase. Una copia por clase

static

```
private static int perros = 0;
```

Tipos de variables

- De instancia: Definida para las instancias de una clase. Una copia por objeto
- De clase: Definida para la clase. Una copia por clase

static

```
private static int perros = 0;
```

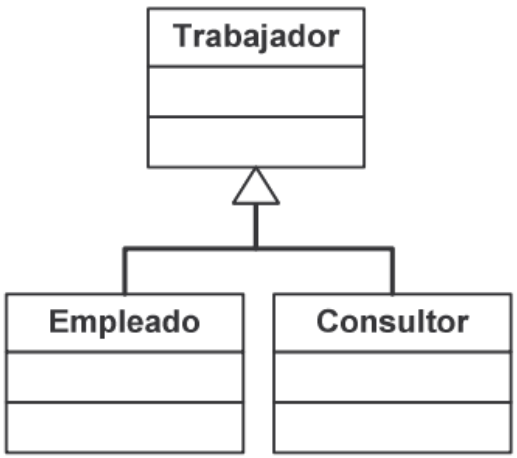
- Local: Definida dentro del cuerpo de un método, ámbito restringido

Índice

Herencia

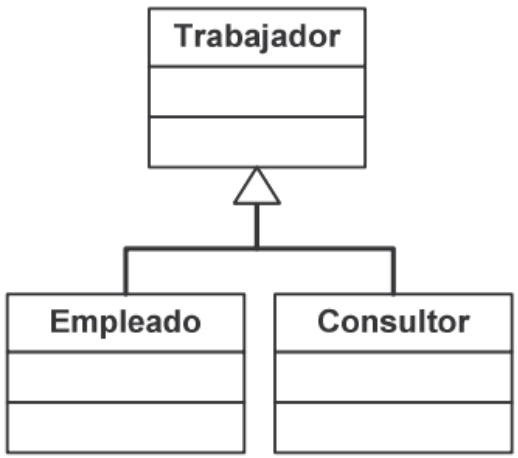
- Es un mecanismo que sirve para reutilizar clases
- Se utiliza cuando existen clases que comparten muchas de sus características
- Se extiende la funcionalidad de clases más genéricas
- Se introducen los conceptos de superclase y subclase

Superclase y Subclase



- Trabajador es la superclase
- Empleado y consultor son subclases

Superclase y subclase



- Los objetos de las subclases pueden poseer atributos y métodos propios (no existentes en la superclase)
- Y además, heredan los atributos y métodos de la superclase

Clase Object

- Realmente todas las clases que creemos son subclases...

Clase Object

- Realmente todas las clases que creemos son subclases...
- ...de la clase Object
- Esta clase tiene métodos habituales que están disponibles para cualquier clase que creemos:
  - clone
  - equals
  - toString
  - ...

Ejemplo de superclase

```
public class Mamifero {

    private String origen;
    private int patas;
    private String nombre;
    ...

    public Mamifero(String nombre, String origen, int
        patas){
        this.nombre = nombre;
        this.origen = origen;
        this.patas = patas
    }
    ...
}
```

Ejemplo de subclase

```
public class Perro extends Mamifero {
    private String raza;
    ...

    public Perro(String nombre, String origen){
        super(nombre, origen, 4)
    }
    ...
}
```

Palabras reservadas

extends

Sirve para indicar la superclase de la cual se hereda

super

Sirve para acceder a miembros de la superclase desde la subclase

- super = Llamada al constructor
- super.metodoquesea = Llamada al métodoquesea

Recordatorio

- Los miembros protected son accesibles desde los miembros de su subclase
- Un objeto de una subclase, también es objeto de la superclase. Al revés no
- Java no permite herencia múltiple

Redefinición de métodos

- Se puede modificar localmente el comportamiento de los métodos heredados
- De esta manera, objetos de diferentes tipos pueden responder de forma diferente a la misma llamada
  - Este es el concepto clave del polimorfismo
- Permite programar de manera más general

Ejemplo

- ¿Se mueve igual un pez, un pájaro o una rana?

Ejemplo

- ¿Se mueve igual un pez, un pájaro o una rana?
- En cambio, todos son animales y por tanto podrían heredar el método mover de dicha clase

Ejemplo

- ¿Se mueve igual un pez, un pájaro o una rana?
- En cambio, todos son animales y por tanto podrían heredar el método mover de dicha clase
- Gracias al polimorfismo, cada objeto concreto realizará la operación mover como corresponda

Apuntes

- No se debe confundir el polimorfismo con la sobrecarga
- En caso de querer no permitir la redefinición de métodos o incluso la creación de subclases:

final

```
public final void mover()
public final class Perro extends Mamifero
```

Clases Abstractas

- Es una clase para la cual nunca se creará una instancia de objetos
- Sirve sólo como superclase, y por tanto, para definir subclases

Clases Abstractas

- Es una clase para la cual nunca se creará una instancia de objetos
- Sirve sólo como superclase, y por tanto, para definir subclases
- Cuando uno de los métodos no tiene implementación, estamos ante una clase abstracta

abstract

```
public abstract class Figura
public abstract double area();
```

Ejemplo abstract

- Una figura es una superclase típica...¿Por qué?

Ejemplo abstract

- Una figura es una superclase típica...¿Por qué?
- No se puede calcular el área de una fígura

Ejemplo abstract

- Una figura es una superclase típica...¿Por qué?
- No se puede calcular el área de una fígura
- Es necesario saber qué figura es la subclase (círculo, cuadrado...)
- Y redefinir el método para calcular el área dependiendo de la figura concreta

Clases Interfaz

- Una interfaz es una clase completamente abstracta
- No contiene nada de implementación ni encapsula datos
- Los atributos sólo pueden ser constantes y deben inicializarse
- Siempre public, static y finalize
- La ventaja es que indica el qué pero no el cómo

Palabras reservadas

interface

```
public interface Figura
```

implements

```
public class Circulo implements Figura
```

Interfaz

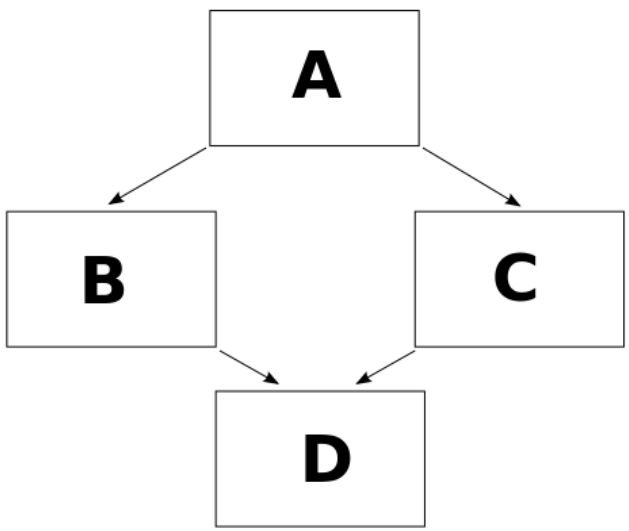
- Toda clase que implemente una interfaz debe proporcionar una definición a sus métodos
- Si alguno de los métodos no obtiene una "conducta", estaremos creando una clase abstracta
- Si no se hace correctamente se obtendrán errores de compilación

Herencia múltiple

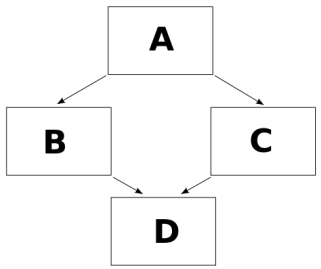
- Ya hemos dicho que no existe herencia múltiple...
- ...pero se pueden usar interfaces para ello

Herencia múltiple

- Ya hemos dicho que no existe herencia múltiple...
- ...pero se pueden usar interfaces para ello



Herencia múltiple



- **A**, **B** y **C** deben ser interfaces
- B y C son subclases de A
- D extiende tanto a B como a C (y por tanto a A)

Índice

Ficheros

- El código de una clase pública se encontrará en un fichero .java
- El nombre del fichero coincidirá con el nombre de la clase

Ficheros

- El código de una clase pública se encontrará en un fichero .java
- El nombre del fichero coincidirá con el nombre de la clase

Excepción

En un fichero podría haber varias clases, si sólo una de ellas es public y el resto clases auxiliares

Paquetes

- Las clases se agrupan en paquetes
- El paquete al que pertenece una clase se indica al comienzo del fichero:

Paquetes

- Las clases se agrupan en paquetes
- El paquete al que pertenece una clase se indica al comienzo del fichero:

Package

```
package ejemplo;
public class ...{
```

Paquetes

- Las clases se agrupan en paquetes
- El paquete al que pertenece una clase se indica al comienzo del fichero:

Package

```
package ejemplo;
public class ...{
```

- Una clase sólo puede pertenecer a un paquete

Paquetes

- Se pueden organizar los paquetes de forma jerárquica

Jerarquia

```
package ejemplo.objetos;
```

- Para usar una clase que está en distinto paquete:
  - Se puede importar la clase entera
  - Se pueden realizar llamadas utilizando: nombrepaquete.loquesea

Importar clases

- Se puede importar un conjunto de clases
- O una clase concreta

import

```
import ejemplo.*;
import ejemplo.Concreto;
```

Accesos directos

- No es recomendable usarlo
- Se utilizaría si vamos a acceder a algo muy concreto de otro paquete

Acceso directo

```
ejemplo.Concreto ejem = new ejemplo.
    Concreto();
```

Librerias externas

- Se pueden utilizar librerias (clases compiladas) externas a nuestro código
- Por defecto, la libreria básica de JDK está cargada
  - Por eso podemos utilizar directamente cosas como Math.PI
- Cualquier paquete compilado se puede guardar como un .jar...
- ...y, por lo tanto, usarlo en otro proyecto

.JAR

- Las librerias son ficheros .JAR (**J**ava **AR**chive)
- Al compilar se guardan en la carpeta "dist"
- Para usarlos, los añadimos a nuestro proyecto:
  - Propiedades - Librerias - Añadir JAR