

Unidad 1: Revisión del paradigma de orientación a objetos

Metodología de la Programación - EUITIO

Algunos conceptos: revisión rápida

- ◆ Revisaremos "por encima"
 - ◆ Abstracción
 - ◆ Encapsulamiento
 - ◆ Ocultación
 - ◆ Constructores y destructores

Abstracción

- ◆ Consiste en la **generalización** conceptual de los **atributos y comportamiento** de un determinado **conjunto de objetos**.
- ◆ Abstraer los **métodos y los datos comunes** a un **conjunto de objetos** y almacenarlos en una clase.

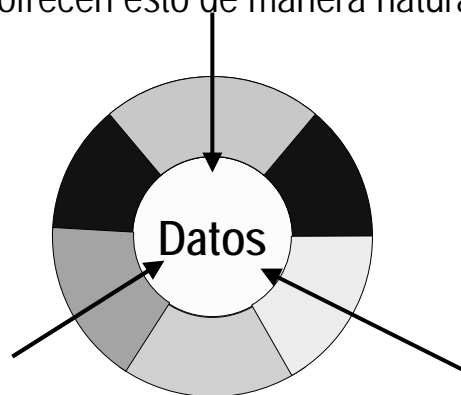
15-02-2007

© A. Cernuda del Río y C. Luengo Díez

3

Encapsulamiento o encapsulación

- ◆ No se accede directamente a los datos, sino siempre a través de métodos de acceso
- ◆ Los objetos ofrecen esto de manera natural



15-02-2007

© A. Cernuda del Río y C. Luengo Díez

4

Ocultación de información (I)

- ◆ Muy relacionada con el encapsulamiento
- ◆ Restricción del acceso a ciertos datos y métodos (privados)
- ◆ Desde fuera de un objeto, muchos detalles de la implementación interna son *invisibles*
- ◆ Ventajas:
 - ◆ Se evitan usos imprevistos de esos métodos o datos → errores
 - ◆ Se pueden cambiar los detalles de implementación sin *estropear* colaboraciones
 - ◆ Así, los cambios y errores tienen un efecto mucho más limitado (vencer la complejidad)

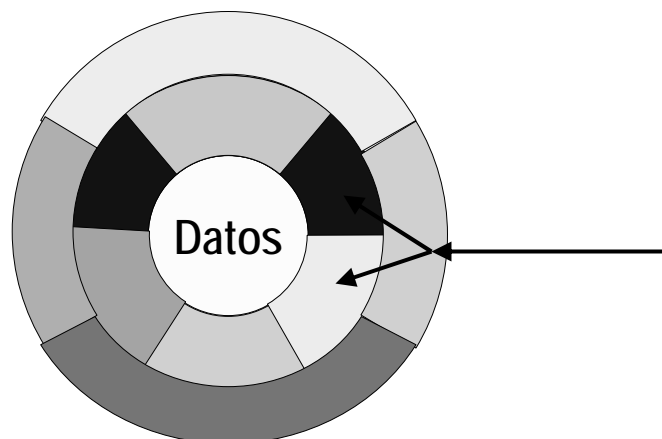
15-02-2007

© A. Cernuda del Río y C. Luengo Díez

5

Ocultación de información (II)

- ◆ Ejemplo de ocultación: métodos privados



15-02-2007

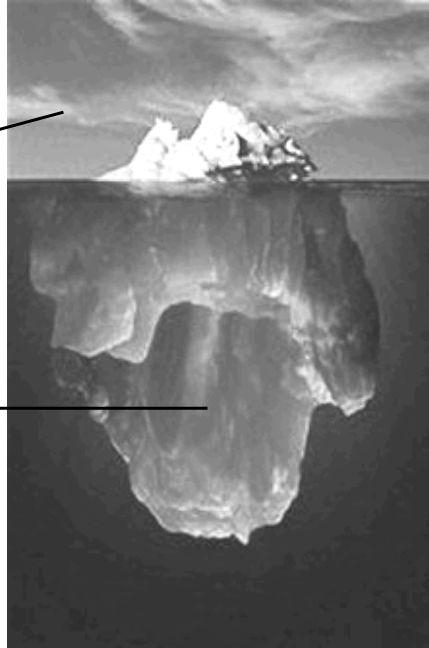
© A. Cernuda del Río y C. Luengo Díez

6

Encapsulamiento y ocultación

Parte pública

Parte privada



15-02-2007

© A. Cernuda del Río y C. Luengo Díez

7

Niveles de Ocultación en Java

- ◆ En Java existen cuatro niveles de ocultación para los miembros de una clase:
 - ◆ **public**, **private**, **protected** y *en blanco*
 - ◆ **private**: Se accede desde cualquier miembro de la misma clase
 - ◆ **protected**: Se accede desde cualquier miembro de la misma clase, clase derivada o miembro de clase del mismo *package*.
 - ◆ **public**: Se accede desde cualquier miembro de la misma clase, clase derivada, clase del mismo *package* o clase de otro *package*.
 - ◆ **en blanco**: Se puede acceder desde cualquier parte del *package* al que pertenezca la clase; es la ocultación por omisión (no tiene palabra reservada)

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

8

Comportamiento (I)

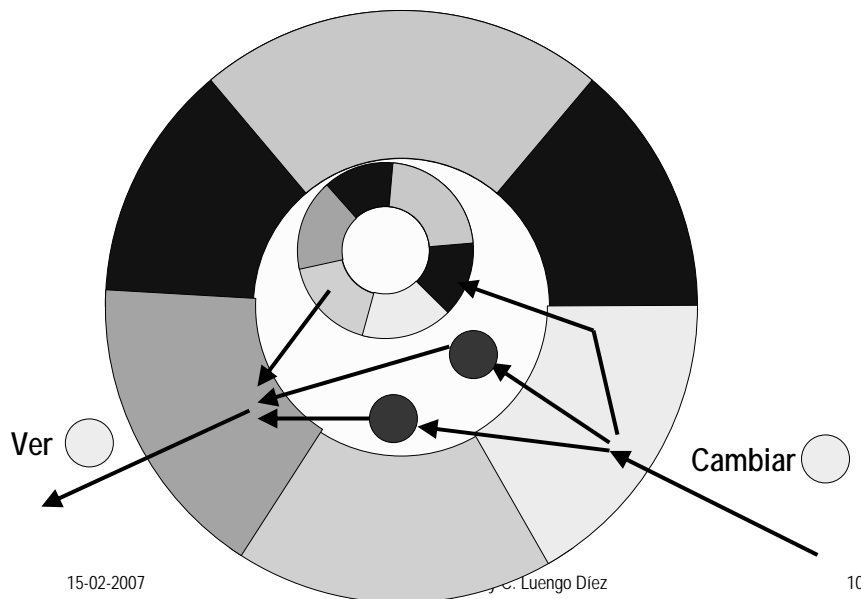
- ◆ Los datos que hay en un objeto son, en principio, *inertes*
- ◆ Representan el *estado* del objeto
- ◆ Pero los métodos permiten implementar un *comportamiento*
- ◆ Ejemplo: al modificar un dato, indirectamente se modifica otro

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

9

Comportamiento (II)



Constructores

- ◆ Hay ciertas acciones que conviene realizar al crear un objeto
 - ◆ Inicialización
 - ◆ Puesta de datos miembro a cero
 - ◆ Creación e inicialización de objetos agregados
- ◆ Se podría hacer con métodos específicos para ello...
- ◆ ...pero eso favorece las omisiones y los errores
- ◆ Por eso existe un "método" que se invoca de manera "automática" al crear el objeto, y está asociado a esa creación: el **constructor**
- ◆ Puede (suele) tener parámetros

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

11

Destructores

- ◆ Análogo al constructor, pero para cuando se destruye el objeto (liberar recursos, destruir ordenadamente los objetos agregados, etc.)
- ◆ Pero...
 - ◆ Tienen sentido en lenguajes en que la destrucción sea explícita (muy importantes en C++)
 - ◆ En Java... recolector de basura
 - ◆ No existe el concepto de destructor. Está el método **finalize()** pero a efectos prácticos no resulta demasiado útil

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

12

Después de la revisión rápida...

- ◆ Hecha esta revisión rápida, nos ocuparemos de la aplicación de los elementos básicos del modelo de objetos

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

13

¿Por qué usamos los conceptos de objetos?

- ◆ Un modelo es "una forma de entender la realidad"
- ◆ El modelo de objetos permite "ver la realidad" de manera que:
 - ◆ Esté simplificada
 - ◆ Usemos los mismos términos para definirla
 - ◆ Los programas "se parezcan" a la realidad
 - ◆ Tengamos alguna idea para empezar el programa
- ◆ Es más útil en programas grandes

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

14

Ideas para identificar clases

- ◆ Buscar sustantivos en el texto
- ◆ ¿Podemos pensar en objetos concretos de esta clase?
- ◆ Parámetros o valores de retorno (de métodos) de cierto tipo que hasta ahora no existía

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

15

Ideas para identificar atributos

- ◆ Atributo: **característica** de un objeto de cierta clase, que puede tomar distintos valores
- ◆ **Cualidad**, algo de lo cual podemos preguntar el valor
- ◆ Algo que da idea del estado de un objeto
- ◆ Algo que permite distinguir un objeto de otro

- ◆ **Atributos estáticos**: Tienen el mismo valor para todos los objetos de una clase

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

16

Ideas para identificar métodos

- ◆ Acciones que "se le pueden pedir" a un objeto de una clase (cosas que puede hacer)
- ◆ Acciones que se realizan (hay que buscar a qué clase se le asignan; "quién es el responsable de hacer esto", "a quién se le puede pedir que haga esto")
- ◆ Métodos estáticos: Tienen el mismo efecto independientemente del objeto
 - ◆ Se ejecutan sin necesidad de un objeto (directamente sobre la clase)
 - ◆ No pueden acceder a ningún atributo, excepto a los estáticos

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

17

Ideas para identificar parámetros

- ◆ Los "materiales" necesarios para realizar una tarea (en un método)
- ◆ La información que hay que dar al responsable de la tarea para que la haga
- ◆ OJO: No hay parámetro si esa información "ya se la sabe" (si son datos miembro o agregados de la clase)

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

18

Ideas para identificar el tipo de retorno de métodos

- ◆ Resultado de una operación (de cara al exterior de la clase)
- ◆ Si el único resultado es una alteración del estado de la clase, no hace falta tipo de retorno

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

19

Apliquémoslo

- ◆ Ejercicios

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

20

Vistos los elementos básicos...

- ◆ A continuación estudiaremos
 - ◆ Herencia
 - ◆ Métodos virtuales
 - ◆ Redefinición

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

21

Herencia

- ◆ Definición
 - ◆ Una clase B deriva de A si B incorpora todas las características de A, además de otras propias
 - ◆ Llamaremos **clase base** a A y **derivada** a B
- ◆ La herencia refleja los procesos mentales de generalización y especialización
 - ◆ Proceso de clasificación
 - ◆ Estructura jerárquica

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

22

Herencia

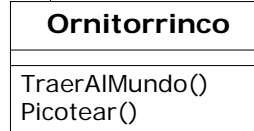
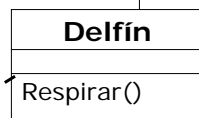
- ◆ La clase derivada tiene los mismos métodos que la clase base
- ◆ En la clase derivada, pueden redefinirse los métodos *virtuales* de la clase base
 - ◆ **Redefinir**: proporcionar en una clase derivada una nueva versión de un método virtual de la clase base
 - ◆ **Método virtual**: el que puede redefinirse en una clase derivada
- ◆ Implementación en Java: palabra reservada **extends**
- ◆ Métodos virtuales: todos (excepto los **final** y **static**)

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

23

Herencia



- Tiene Edad.
- Tiene Peso.
- Tiene los métodos Mamar() y TraerAlMundo().
- Tiene su propia versión de Respirar().

- Tiene Edad.
- Tiene Peso.
- Tiene Mamar() y Respirar().
- Tiene su versión de TraerAlMundo().
- Tiene un método Picotear().

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

24

Herencia

◆ Principio de sustitución de Liskov

Una clase base debe poder sustituirse por una clase derivada

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

25

Clases derivadas: Constructores

- ◆ Por defecto desde un constructor de una clase derivada se llama al constructor sin argumentos de la clase base.
- ◆ Si se desea llamar a otro constructor de la clase base se utiliza la palabra reservada **super**
- ◆ Para mantener el encapsulamiento, una clase derivada debe inicializar sus variables específicas en el constructor, y dejar al constructor de la clase base inicializar las suyas.

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

26

Clases abstractas

- ◆ Método virtual puro: el que no tiene implementación
- ◆ Se marca con la palabra **abstract**
- ◆ Clase abstracta: tiene al menos un método virtual puro
- ◆ Se marca con la palabra **abstract**
- ◆ De una clase abstracta no se pueden crear objetos
- ◆ Pero Sí se pueden declarar referencias

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

27

Polimorfismo

- ◆ Definición
 - ◆ Utilización de objetos de diversas clases como si todos ellos fuesen de la misma
- ◆ Ventajas
 - ◆ Un programa puede crecer fácilmente:
 - ◆ Se añaden clases derivadas
 - ◆ El programa puede usar las nuevas clases sin cambiar nada
 - ◆ Se aprovecha la herencia para simplificar los programas (véase a continuación...)

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

28

Polimorfismo

◆ Ejemplo:

```
Vehiculo v;  
...  
if (v.VerTipo() == "Bicicleta")  
{  
  ...  
}  
else  
  if (v.VerTipo() == "Coche")  
  {  
    ...  
  }  
  else  
    if (v.VerTipo() == "Patinete")  
    {  
      ...  
    }
```

MAL ASUNTO

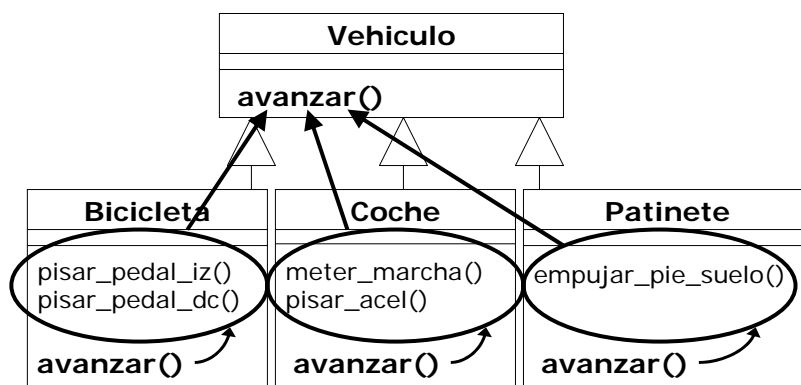
Accidente, s.
Acontecimiento inevitable debido a la acción de leyes naturales inmutables.
(Ambrose Bierce – Diccionario del Diablo)

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

29

Polimorfismo



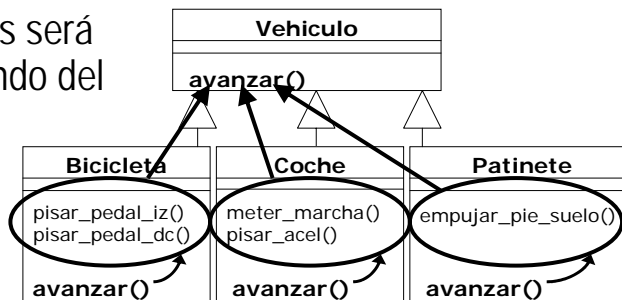
15-02-2007

© A. Cernuda del Río y C. Luengo Díez

30

Polimorfismo

- ◆ Si tenemos un algoritmo `conducir`, es mejor que se limite a invocar a `avanzar`
- ◆ Si invoca a `meter_marcha`, `pisar_ace1`, etc... en el algoritmo "aparcarse" tendremos que volver a hacer lo mismo
- ◆ Y lo que hagamos será distinto dependiendo del vehículo concreto



15-02-2007

© A. Cernuda del Río y C. Luengo Díez

31

Polimorfismo: síntomas

- ◆ Síntomas de que se necesita usar polimorfismo:
 - ◆ Hacemos una tarea similar (de manera ligeramente distinta) en todas las subclases, repitiendo código
 - ◆ Hacemos algo de manera diferente dependiendo de la clase específica de un objeto
 - ◆ Necesitamos preguntar la clase de un objeto (no nos basta saber la clase base)

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

32

Interfaces (I)

- ◆ Conjunto de operaciones que se pueden invocar sobre un objeto
 - ◆ Sus signaturas (nombres, parámetros y tipos de retorno)
- ◆ Interfaz: "cara" que el objeto presenta al exterior
 - ◆ Ejemplo: volante, pedales, palanca de cambio
- ◆ La interfaz se asemeja a un "contrato"
 - ◆ Si un objeto me ofrece un conjunto de operaciones "estándar", me da igual cuál sea su tipo (clase); "podremos entendernos"

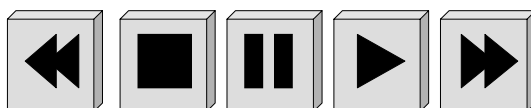
15-02-2007

© A. Cernuda del Río y C. Luengo Díez

33

Interfaces (II)

- ◆ Ejemplo de interfaz



```
public interface ChismeReproductor
{
    void play();
    void pause();
    void stop();
    void fast_forward();
    void rewind();
}
```

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

34

Interfaces (III)

- ◆ Para operar con esa interfaz...
 - ◆ Sólo necesito que el objeto que tengo entre manos la implemente
 - ◆ `class video implements ChismeReproductor { ... }` → Puedo operar con el vídeo (recuérdese: "contrato")
 - ◆ Me da igual de qué clase sea ese objeto
 - ◆ ¿Aparato de vídeo o de audio? ¿DVD o VHS o VideoCD? ¿CD o cassette o MiniDisc o tocadiscos?
 - ◆ Las posibles clases ni siquiera tienen que heredar de una común
 - ◆ El PC tiene esa interfaz... y no "es un"... ¿qué?

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

35

Interfaces (IV)

- ◆ Recuérdese la herencia: si B hereda de A,

B ES-UN A

- ◆ En las interfaces, si B implementa A,

B EJERCE-DE A

- ◆ En Java, una clase sólo puede "ser" una cosa, pero puede "ejercer de" muchas

```
class nombre implements interfaz1, ..., interfazn
```

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

36

Ejemplos- Interfaces

```
public interface Parlanchin {
    void habla();
}

class Reloj implements Parlanchin{
    public void habla(){
        System.out.println("¡Cucu, cucu, ..!");
    }
}

class Perro extends Animal implements
    Parlanchin{
    public void habla(){
        System.out.println("¡Guau!");
    }
}
}
```

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

37

Ejemplos- Interfaces

```
public interface InstrumentoMusical {
    void tocar();
    void afinar();
    String tipoInstrumento();
}

class InstrumentoViento extends Instrumento
    implements InstrumentoMusical {
    void tocar() { . . . };
    void afinar() { . . . };
    String tipoInstrumento() {...}
}

class Gaita extends InstrumentoViento {
    String tipoInstrumento() {
        return "Gaita";
    }
}
}
```

15-02-2007

© A. Cernuda del Río y C. Luengo Díez

38

Interfaces (V)

- ◆ En Java, existen elementos del lenguaje de programación para usar las interfaces
- ◆ En otros lenguajes, no
 - ◆ En C++, para hacer lo mismo, se suele recurrir a clases base abstractas
 - ◆ Eso nos obliga a heredar de ellas si queremos implementar ese "contrato" o interfaz
 - ◆ Puesto que no hay interfaces, en C++ es imprescindible la herencia múltiple para poder hacer esto