Programación orientada a objetos

"En vez de un procesador de celdas de memoria…
tenemos un universo de objetos de buen comportamiento
que cortésmente solicitan a las demás llevar a cabo sus deseos"
-- Ingalls, 1981 (revista Byte)

Evolución

Años 60 **Simula**
Resolución de problemas de simulación
© Ole-Johan Dahl & Krysten Nygaard (Noruega)

Años 70 **Smalltalk**
Entorno de programación entendible por "novatos"
© Alan Kay (Xerox PARC, Palo Alto, California)

Años 80 **C++**
Extensión de C
© Bjarne Stroustroup (AT&T Bell Labs)

Años 90 **Java**
"Write once, run everywhere"
© Sun Microsystems

Conceptos básicos

- Todo es un objeto
- Los objetos se comunican entre sí pasándose mensajes
- Cada objeto tiene un estado
  (contiene su propia memoria [datos])
- Un objeto es un caso particular (instancia) de una clase
- Las clases definen el comportamiento de un conjunto de objetos

Resolución de problemas “con orientación a objetos”

Problema
Quiero enviar un paquete a un amigo que vive en otra ciudad

Opciones

a) Hacerlo todo yo mismo
   ~ Descomposición en subproblemas
     (programación estructurada)

b) Delegar en alguien para que lo haga (p.ej. Correos)
   ~ “Realizar un encargo”
     (programación orientada a objetos)

Solución orientada a objetos

- Se busca un objeto capaz de enviar un paquete
- Se le envía un mensaje con mi solicitud
- El objeto se hace responsable de satisfacer mi solicitud
- El objeto utiliza un algoritmo que yo no tengo por qué conocer

Consecuencias

- Un programa orientado a objetos se estructura como un conjunto de agentes que interactúan (programa como colección de objetos).
- Cada objeto proporciona un servicio que es utilizado por otros objetos (reutilización).
- La acción se inicia por la transmisión de un mensaje al objeto responsable de realizarla.
- Si el receptor acepta el mensaje, acepta la responsabilidad de llevar a cabo la acción solicitada.
- El receptor puede utilizar cualquier técnica que logre el objetivo deseado.

Clases y objetos

Clase

Implementación de un tipo de dato.

Una clase sirve tanto de *módulo* como de *tipo*

- **Tipo**: Descripción de un conjunto de objetos (equipados con ciertas operaciones).
- **Módulo**: Unidad de descomposición del software.

Objeto

Instancia de una clase:
Unidad atómica que encapsula estado y comportamiento.

- Un objeto puede caracterizar una entidad física (un teléfono, un interruptor, un cliente) o una entidad abstracta (un número, una fecha, una ecuación matemática).
- Todos los objetos son instancias de una clase: Los objetos se crean por instanciación de las clases.
- Todos los objetos de una misma clase (p.ej. coches) comparten ciertas características: sus atributos (tamaño, peso, color, potencia del motor…) y el comportamiento que exhiben (aceleran, frenan, curvan…).

Todo objeto tiene…

- **Identidad** (puede distinguirse de otros objetos)
- **Estado** (datos asociados a él)
- **Comportamiento** (puede hacer cosas)

Las diferentes instancias de cada clase difieren entre sí por los valores de los datos que encapsulan (sus atributos).

Dos objetos con los mismos valores en sus atributos pueden ser diferentes.

TODOS los objetos de una misma clase usan el mismo algoritmo como respuesta a mensajes similares.

El algoritmo empleado como respuesta a un mensaje (esto es, el método invocado) viene determinado por la clase del receptor.

Una clase es una descripción de un conjunto de objetos similares.

Al programar, definimos una clase para especificar cómo se comportan y mantienen su estado los objetos de esa clase.

A partir de la definición de la clase, se crean tantos objetos de esa clase como nos haga falta

Clases en Java

Cada clase en Java:

- Se define en un fichero independiente con extensión `.java`.
- Se carga en memoria cuando se necesita.

La máquina virtual Java determina en cada momento las clases necesarias para la aplicación y las carga en memoria.

El programa puede ampliarse dinámicamente (sin tener que recompilar): La aplicación no es un bloque monolítico de código.

Para definir una clase en Java se utiliza la palabra reservada `class`, seguida del nombre de la clase (un identificador):

```
public class MiClase
{
  …
}
```

NOTA: Es necesario que indiquemos el modificador de acceso `public` para que podamos usar nuestra clase "desde el exterior".

Los límites de la clase se marcan con llaves { … }

? Se debe sangrar el texto que aparece entre las llaves para que resulte más fácil delimitar el ámbito de los distintos elementos de nuestro programa. De esta forma se mejora la legibilidad de nuestro programa.

Acerca del nombre de las clases

El nombre de una clase debe ser un identificador válido en Java.

Por convención, los identificadores que se les asignan a las clases en Java comienzan con mayúscula.

Además, en Java, las clases públicas deben estar definidas en ficheros con extensión `.java` cuyo nombre coincida exactamente con el identificador asignado a la clase (¡ojo con las mayúsculas y las minúsculas, que en Java se consideran diferentes!)

Errores comunes

- Cuando el nombre de la clase no coincide con el nombre del fichero, el compilador nos da el siguiente error:

```
Public class MiClase must be defined
in a file called MiClase.java
```

  donde `MiClase` es el nombre de nuestra clase.

- Las llaves siempre deben ir en parejas: Si falta la llave de cierre } el compilador da un error:

```
'}' expected
```

  Lo mismo ocurre si se nos olvida abrir la llave ({):

```
'{' expected
```

? Es una buena costumbre cerrar inmediatamente una llave en cuanto se introduce en el texto del programa. Después se posiciona el cursor entre las dos llaves y se completa el texto del programa.

Uso del compilador `javac`

Al compilar un programa con `javac`,
el compilador nos muestra la lista de errores que se han detectado.

Para cada error, el compilador nos ofrece los siguientes datos:

- El nombre del fichero donde está el error.
- La línea en la que se ha detectado el error.
- Un mensaje descriptivo del tipo de error (en inglés).
- Una reproducción de la línea **donde se detectó** el error.
- La posición exacta donde se detectó el error en la línea.

Ejemplos

Al compilar el fichero `Error.java`, cuyo contenido es

```
public class 2Error
{
}
```

el compilador nos da los siguientes errores:

```
Error.java:1: malformed floating point literal
public class 2Error
             ^
Error.java:4: '{' expected

^
2 errors
```

El primer error se produce porque el identificador de la clase no es válido y el segundo error es consecuencia del anterior.

Si cambiamos el identificador de la clase por un identificador válido en Java, se produce un error si el nombre de la clase con el nombre del fichero:

```
Error.java:1: class Error3 is public,
should be declared in a file named Error3.java
public class Error3
       ^
1 error
```

El compilador también nos dará un error si se nos olvida el punto y coma del final de una sentencia o de una declaración, tal como sucede en el siguiente ejemplo:

```
public class Error
{
  int x
}
```

En este caso, el error se detecta en la línea siguiente a la declaración de la variable `x`:

```
Error.java:4: ';' expected
}
^
1 error
```

En la mayoría de los lenguajes de programación, Java incluido, el compilador ignora los espacios y líneas en blanco que introduzcamos en el texto del programa.

- Los espacios y líneas en blanco los usaremos nosotros para mejorar la legibilidad del texto del programa.

Cuando el compilador nos da un error, debemos comprobar la línea en la que se detecta el error para corregirlo. Si esa línea no contiene errores sintácticos, entonces debemos analizar las líneas que la preceden en el texto del programa.

Encapsulación de datos (variables) y operaciones (métodos)

Objeto = Identidad + Estado + Comportamiento

Identidad

La identidad de un objeto lo identifica unívocamente:

- Es independiente de su estado.
- No cambia durante la vida del objeto.

Estado

El estado de un objeto viene dado por los valores de sus atributos.

- Cada atributo toma un valor en un dominio concreto.
- El estado de un objeto evoluciona con el tiempo
- Los atributos de un objeto no deberían ser manipulables directamente por el resto de objetos del sistema (*ocultamiento de información*):
  - Se protegen los datos de accesos indebidos.
  - Se distingue entre interfaz e implementación.
  - Se facilita el mantenimiento del sistema.

Interfaz vs. implementación
Los objetos se comunican a través de interfaces bien definidas sin tener que conocer los detalles internos de implementación de los demás objetos.

Ejemplo: Un coche se puede conducir…
- Sin saber exactamente de qué partes consta el motor ni cómo funciona éste (implementación).
- Basta con saber manejar el volante, el acelerador y el freno (interfaz).

Comportamiento

Los métodos que definen el comportamiento de un objeto:

- Agrupan las competencias del objeto (responsabilidades)
- Describen sus acciones y reacciones.

  Las acciones realizadas por un objeto son consecuencia directa de un estímulo externo (un mensaje enviado desde otro objeto) y dependen del estado del objeto.

El interfaz de un objeto ha de establecer un contrato, un conjunto de condiciones precisas que gobiernan las relaciones entre una clase proveedora y sus clientes (*diseño por contrato*).

Estado y comportamiento están relacionados.

Ejemplo
Un avión no puede aterrizar (acción) si no está en vuelo (estado)

Representación gráfica de una clase (notación UML)

Una clase se representa con un rectángulo dividido en tres partes:

- El nombre de la clase
  (identifica la clase de forma unívoca)
- Sus atributos
  (datos asociados a los objetos de la clase)
- Sus operaciones
  (comportamiento de los objetos de esa clase)

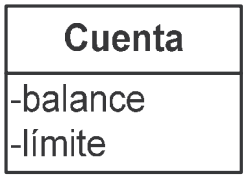
IMPORTANTE: Las clases se deben identificar con un nombre que, por lo general, pertenecerá al vocabulario utilizado habitualmente al hablar del problema que tratamos de resolver.

Representación de una clase:


```
public class Cuenta
{
  …
}
```

Declaración en Java de una clase.

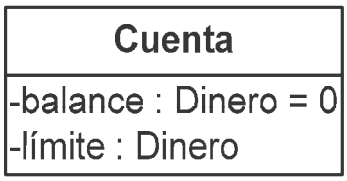
Representación de una clase con sus atributos:




```
public class Cuenta
{
  private double balance;  // Saldo
  private double limit;    // Límite
  …
}
```

Las variables de instancia
sirven para especificar en Java los atributos de la clase.

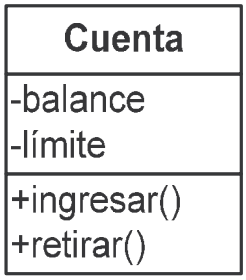
Representación del tipo y de los valores por defecto de los atributos:




```
public class Cuenta
{
  private double balance = 0;
  private double limit;
  …
}
```

NOTA: Aquí hemos decidido utilizar el tipo primitivo `double` para representar cantidades de dinero. En un programa en el que no se pudiesen permitir errores de redondeo en los cálculos, deberíamos utilizar otro tipo más adecuado (por ejemplo, `BigDecimal`).

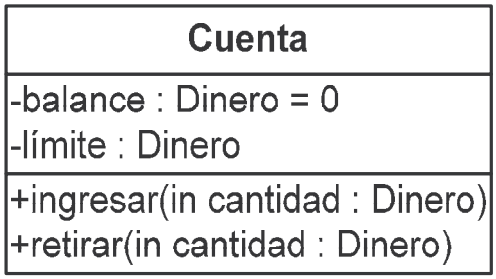
Representación de las operaciones de una clase:




```
public class Cuenta
{
  private double balance = 0;
  private double limit;

  public void ingresar (…) …
  public void retirar (…)  …
}
```

Los métodos implementan en Java las operaciones características de los objetos de una clase concreta.

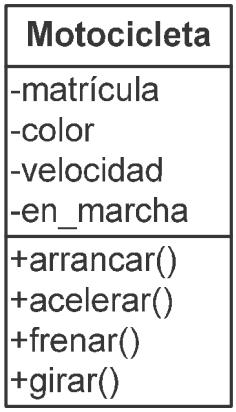
Especificación completa de la clase:




```
public class Cuenta
{
  private double balance = 0;
  private double limit;

  public void ingresar (double cantidad)
  {
     balance = balance + cantidad;
   }

  public void retirar (double cantidad)
  {
     balance = balance – cantidad;
  }
}
```

Clase de ejemplo

Representación gráfica en UML:



Definición en Java:
Fichero `Motocicleta.java`

```
public class Motocicleta
{
  // Atributos (variables de instancia)
  private String matricula;  // Placa de matrícula
  private String color;      // Color de la pintura
  private int velocidad;     // Velocidad actual (km/h)
  private boolean en_marcha; // ¿moto arrancada?

  // Operaciones (métodos)
  public void arrancar ()
  { … }
  public void acelerar ()
  { … }
  public void frenar ()
  { … }
  public void girar (float angulo)
  { … }
}
```

Uso de objetos

El operador .

El operador . (punto) en Java
nos permite acceder a los distintos miembros de una clase:

```
objeto.miembro
```

Cuando tenemos un objeto de un tipo determinado y queremos acceder a uno de sus miembros sólo tenemos que poner el identificador asociado al objeto (esto es, el identificador de una de las variables de nuestro programa) seguido por un punto y por el identificador que hace referencia a un miembro concreto de la clase a la que pertenece el objeto.

¿Cómo se comprueba el estado de un objeto?
Accediendo a las variables de instancia del objeto

```
objeto.atributo
```

Por ejemplo, `cuenta.balance` nos permitiría acceder
al valor numérico correspondiente al saldo de una cuenta,
siempre y cuando `cuenta` fuese una instancia de la clase `Cuenta`.

¿Cómo se le envía un mensaje a un objeto?
Invocando a uno de sus métodos.

```
objeto.método(lista de parámetros)
```

La llamada al método hace que el objeto realice la tarea especificada en la implementación del método, tal como esté definida en la definición de la clase a la que pertenece el objeto.

Ejemplos

```
cuenta.ingresar(150.00);
```

- Si `cuenta` es el identificador asociado a una variable de tipo `Cuenta`, se invoca al método `ingresar` definido en la clase `Cuenta` para depositar una cantidad de dinero en la cuenta.

- La implementación del método `ingresar` se encarga de actualizar el saldo de la cuenta, sin que nosotros nos tengamos que preocupar de cómo se realiza esta operación.

```
System.out.println("Mensaje");
```

- `System` es el nombre de una clase incluida en la biblioteca de clases estándar de Java.

- `System.out` es un miembro de la clase `System` que hace referencia al objeto que representa la salida estándar de una aplicación Java.

- `println()` es un método definido en la clase a la que pertenece el objeto `System.out`.

- La implementación del método `println()` se encarga de mostrar el mensaje que le pasamos como parámetro y hace avanzar el cursor hasta la siguiente línea (como si pulsásemos la tecla ↵).

- `System.out.println()` es una llamada a un método, el método `println()` del objeto `System.out`

- La línea completa forma una sentencia (terminada con un punto y coma) que delega en el método `println()` para que éste se encargue de mostrar una línea en pantalla.

Creación de objetos

Antes de poder usar un objeto hemos de crearlo…

El operador `new`

El operador `new` nos permite crear objetos en Java.

```
Tipo identificador = new Tipo();
```

Si escribimos un programa como el siguiente:

```
public class Ingreso
{
  public static void main (String args[])
  {
    Cuenta cuenta;                    // Error

    cuenta.ingresar(100.00);
  }
}
```

El compilador nos da el siguiente error:

```
Ingreso.java:7:
variable cuenta might not have been initialized
    cuenta.ingresar(100.00);
    ^
```

Hemos declarado una variable que, inicialmente, no tiene ningún valor. Antes de utilizarla, deberíamos haberla inicializado (con un objeto del tipo adecuado):

```
Cuenta cuenta = new Cuenta();
```

Observaciones

- Se suele crear una clase aparte, que únicamente contenga un método `main`, como punto de entrada de la aplicación.
- En la implementación del método `main` se crean los objetos que sean necesarios y se les envían mensajes para indicarles lo que deseamos que hagan.

Constructores

Cuando utilizamos el operador `new` acompañado del nombre de una clase, se crea un objeto del tipo especificado (una instancia de la clase cuyo nombre aparece al lado de `new`).

Al crear un objeto de una clase concreta, se invoca a un método especial de esa clase, denominado **constructor**, que es el que se encarga de inicializar el estado del objeto.

Constructor por defecto

Por defecto, Java crea automáticamente un constructor sin parámetros para cualquier clase que definamos.

```
public class Cuenta
{
  // Constante
  public static final double LIMITE_NORMAL = 300.00;

  // Variables de instancia
  private double balance = 0;
  private double limit = LIMITE_NORMAL;

  // Métodos
  ...
}
```

Al crear un objeto de tipo `Cuenta` con `new Cuenta()`, se llama al constructor por defecto de la clase `Cuenta`, con lo cual se crea un objeto de tipo `Cuenta` cuyo estado inicial será el indicado en la inicialización de las variables de instancia `balance` y `limit`.

Constructores definidos por el usuario

Los lenguajes de programación nos permiten definir constructores para especificar cómo ha de inicializarse un objeto al crearlo.

> El nombre del constructor ha de coincidir con el nombre de la clase.

Podemos definir un constructor
para inicializar las variables de instancia de una clase,
en vez de hacerlo en la propia declaración de las variables de instancia:

```
public class Cuenta
{
  // Constante
  public static final double LIMITE_NORMAL = 300.00;

  // Variables de instancia
  private double balance;
  private double limit;

  // Constructor sin parámetros
  public Cuenta ()
  {
    this.balance = 0;
    this.limit   = LIMITE_NORMAL;
  }

  // Métodos
  ...
}
```

> La palabra reservada `this` hace referencia al objeto
> que realiza la operación cuya implementación especifica el método.

También podemos definir constructores con parámetros:

```
public class Cuenta
{
  // Constante
  public static final double LIMITE_NORMAL = 300.00;

  // Variables de instancia
  private double balance;
  private double limit;

  // Constructor con parámetros
  public Cuenta (double limit)
  {
    this.balance = 0;
    this.limit = limit;
  }

  // Métodos
  ...
}
```

Ahora, para crear un objeto de tipo Cuenta, hemos de especificar los parámetros adecuados para su constructor:

```
Cuenta cuenta = new Cuenta(Cuenta.LIMITE_NORMAL);
```

O bien

```
Cuenta cuentaVIP = new Cuenta(6000.00);
```

La cuenta VIP tendrá un límite de 6000€ en vez del límite normal.

> En cuanto definimos un constructor, ya no podemos utilizar el constructor por defecto de la clase (el constructor sin parámetros que Java crea automáticamente si no especificamos ninguno).

Si sólo está definido el constructor anterior

```
public Cuenta (double limit)
```

al crear un objeto con

```
Cuenta cuenta = new Cuenta();
```

el compilador de Java nos da el siguiente error:

```
CuentaTest.java:5: cannot find symbol
symbol  : constructor Cuenta()
location: class Cuenta
    Cuenta cuenta = new Cuenta();
                    ^
1 error
```

porque no existe ningún constructor
sin parámetros definido para la clase `Cuenta`.

> Para facilitarnos la creación de objetos,
> Java nos permite definir varios constructores para una misma clase
> (siempre y cuando tengan parámetros diferentes).

De forma que podemos incluir los dos constructores
en la implementación de la clase `Cuenta` y escribir lo siguiente:

```
public class CuentaTest
{
  public static void main (String args[])
  {
    Cuenta cuentaNormal = new Cuenta();
    Cuenta cuentaVIP    = new Cuenta(6000.00);
    …
  }
}
```

Referencias

Cualquier tipo que definamos en Java con una clase
es un tipo no primitivo.

Cuando declaramos una variable de un tipo primitivo en Java, estamos reservando espacio en memoria para almacenar un **valor** del tipo correspondiente.

Sin embargo,
cuando declaramos una variable de un tipo no primitivo en Java,
lo único que hacemos es reservar una zona en memoria
donde se almacenará una **referencia** a un objeto del tipo especificado
(y no el objeto en sí, de ahí la necesidad de utilizar el operador `new`).

Inicialización por defecto de los objetos en Java

Cuando se crea un objeto con el operador `new`, por defecto:

- Las variables de instancia de tipo numérico (`byte, short, int, long, float y double`) se inicializan a 0.
- Las variables de instancia de tipo `char` se inicializan a `'\0'`.
- Las variables de instancia de tipo `boolean` se inicializan a `false`.
- Las variables de instancia de cualquier tipo no primitivo se inicializan a `null` (una palabra reservada del lenguaje que indica que la referencia no apunta a ninguna parte).

IMPORTANTE

Para acceder a un miembro de un objeto
(leer el valor de una variable de instancia o invocar un método)
hemos de tener una referencia a un objeto distinta de `null`

Uso de sentencias de asignación

Cuando usamos datos de tipos primitivos,
las sentencias de asignación copian valores:

```
// Intercambio de los valores de dos variables

int x = 100;
int y = 200;
int tmp;

tmp = y;   // La variable temporal
           // almacena el valor inicial de y

y = x;     // Se almacena en y el valor de x (100)

x = tmp;   // Se almacena en x
           // el valor original de y (200)
```

Cuando usamos objetos (datos de tipos no primitivos),
las sentencias de asignación copian referencias:

```
Cuenta miCuenta = new Cuenta();
Cuenta tmp;

tmp = miCuenta;         // Copia la referencia

tmp.ingresar(150);      // Se hace un ingreso en
                        //   ¡¡¡ miCuenta !!!
```

MUY IMPORTANTE

En una sentencia de asignación,
no se crea una copia del dato representado por un objeto
(salvo que trabajemos con tipos primitivos)

Relaciones entre clases: Diagramas de clases UML

Las relaciones existentes entre las distintas clases
nos indican cómo se comunican los objetos de esas clases entre sí:

Los mensajes "navegan"
por las relaciones existentes entre las distintas clases.

Existen distintos tipos de relaciones:

- **Asociación** (conexión entre clases)
- **Dependencia** (relación de uso)
- **Generalización/especialización** (relaciones de herencia)

Asociación

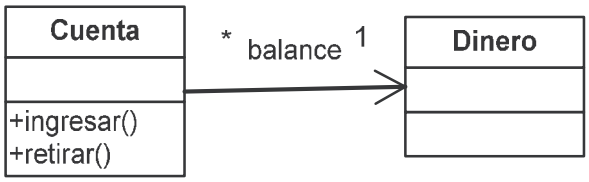
Una asociación es una relación estructural que describe una conexión entre objetos.

Gráficamente, se muestra como una línea continua que une las clases relacionadas entre sí.

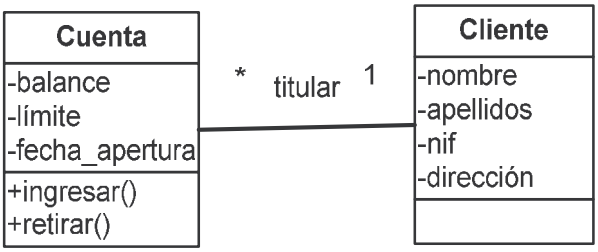
Navegación de las asociaciones

Aunque las asociaciones suelen ser bidireccionales (se pueden recorrer en ambos sentidos), en ocasiones es deseable hacerlas unidireccionales (restringir su navegación en un único sentido).

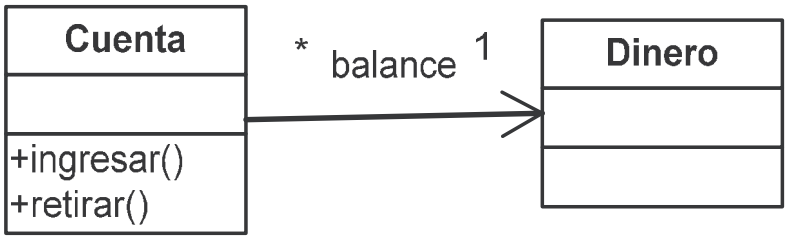
Gráficamente, cuando la asociación es unidireccional, la línea termina en una punta de flecha que indica el sentido de la asociación:



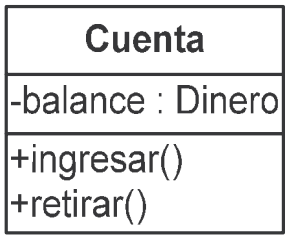
Asociación unidireccional



Asociación bidireccional

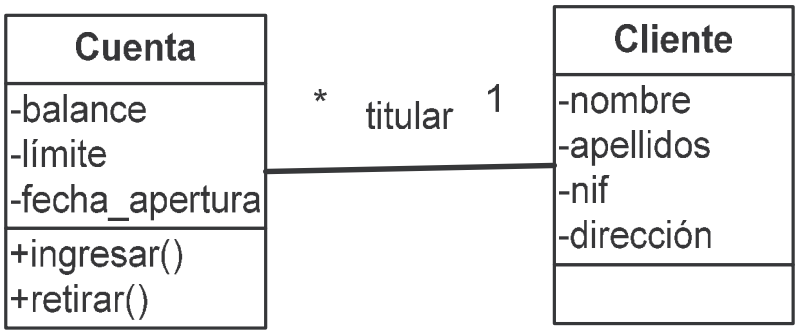


equivale a

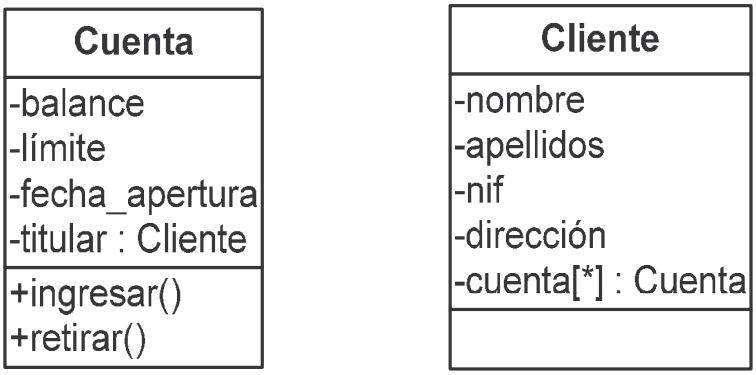



```
class Cuenta
{
  private Dinero balance;

  public void ingresar (Dinero cantidad)
  {
    balance += cantidad;
  }

  public void retirar (Dinero cantidad)
  {
    balance -= cantidad;
  }

  public Dinero getSaldo ()
  {
    return balance;
  }
}
```

Hemos supuesto que `Dinero` es un tipo de dato con el que se pueden hacer operaciones aritméticas y hemos añadido un método adicional que nos permite comprobar el saldo de una cuenta.



viene a ser lo mismo que



con la salvedad de
que el enlace bidireccional hemos de mantenerlo nosotros

```
public class Cuenta
{
  …
  private Cliente titular;
  …
}

public class Cliente
{
  …
  private Cuenta cuenta[];
  …
}
```

Un cliente puede tener varias cuentas, por lo que en la clase cliente hemos de mantener un conjunto de cuentas (un vector en este caso).

Multiplicidad de las asociaciones

La multiplicidad de una asociación determina
cuántos objetos de cada tipo intervienen en la relación:

> El número de instancias de una clase que se relacionan
> con UNA instancia de la otra clase.

- Cada asociación tiene dos multiplicidades
  (una para cada extremo de la relación).

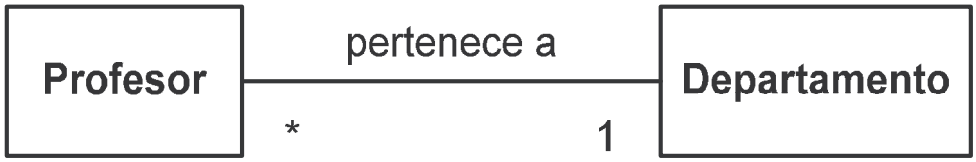
- Para especificar la multiplicidad de una asociación hay que indicar la multiplicidad mínima y la multiplicidad máxima (mínima..máxima)

Multiplicidad	Significado
`1`	Uno y sólo uno
`0..1`	Cero o uno
`N..M`	Desde N hasta M
`*`	Cero o varios
`0..*`	Cero o varios
`1..*`	Uno o varios (al menos uno)

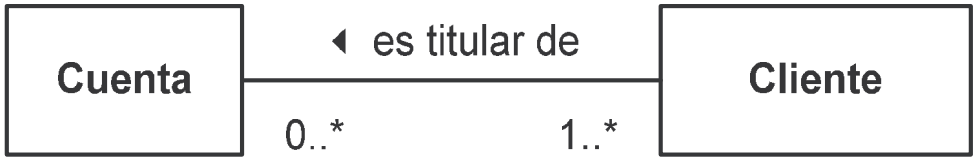
- Cuando la multiplicidad mínima es 0, la relación es opcional.

- Una multiplicidad mínima mayor o igual que 1 establece una relación obligatoria.

Todo departamento tiene un director.
Un profesor puede dirigir un departamento.



Todo profesor pertenece a un departamento.
A un departamento pueden pertenecer varios profesores.

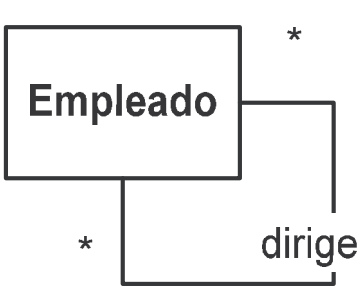


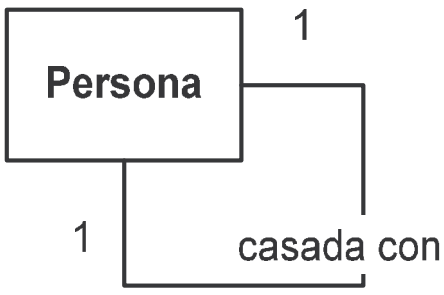
Relación opcional
Un cliente puede o no ser titular de una cuenta

Relación obligatoria
Una cuenta ha de tener un titular como mínimo

Relaciones involutivas

Cuando la misma clase aparece en los dos extremos de la asociación.





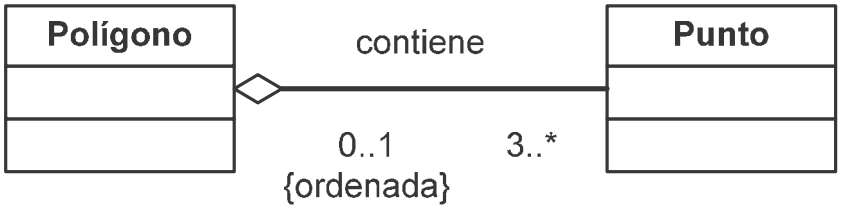
Agregación y composición

Casos particulares de asociaciones:
Relación entre un todo y sus partes

Gráficamente,
se muestran como asociaciones con un rombo en uno de los extremos.

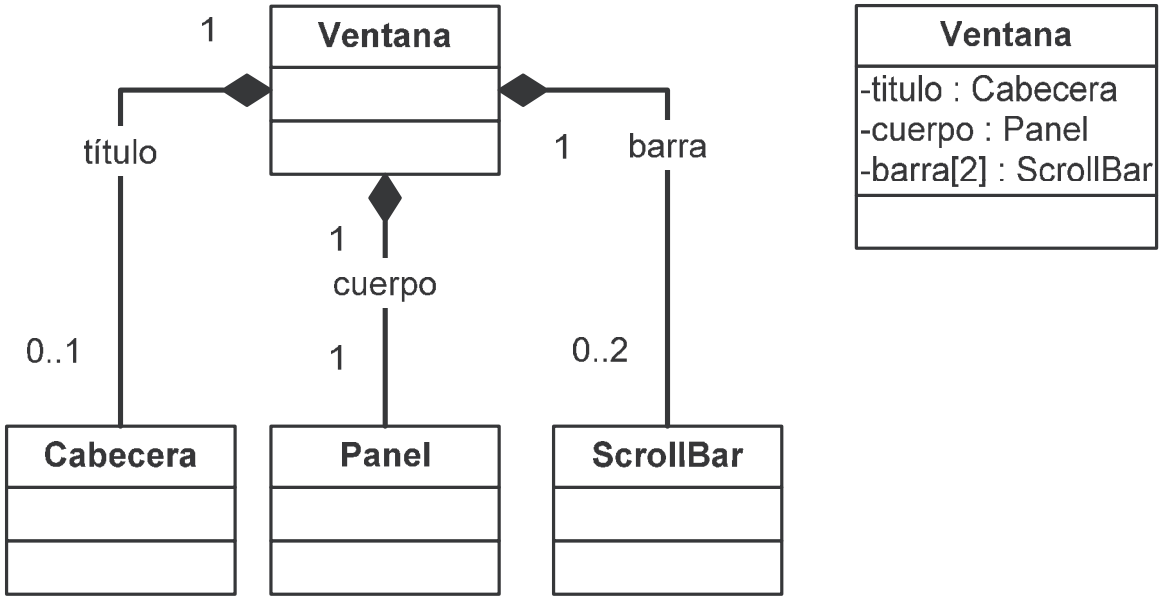
Agregación

Las partes pueden formar parte de distintos agregados.



Composición

Agregación disjunta y estricta:
Las partes sólo existen asociadas al compuesto
(sólo se accede a ellas a través del compuesto)



Dependencia

Relación (más débil que una asociación) que muestra la relación entre un cliente y el proveedor de un servicio usado por el cliente.

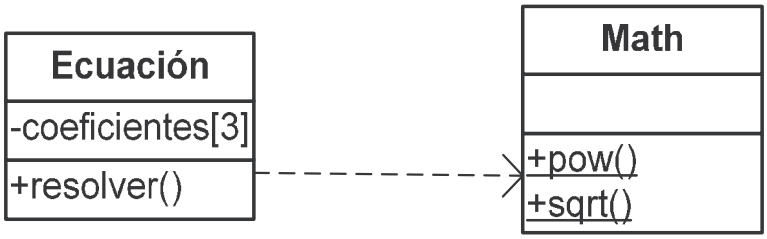
- Cliente es el objeto que solicita un servicio.

- Servidor es el objeto que provee el servicio solicitado.

Gráficamente, la dependencia se muestra como una línea discontinua con una punta de flecha que apunta del cliente al proveedor.

Ejemplo

Resolución de una ecuación de segundo grado



$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Para resolver una ecuación de segundo grado hemos de recurrir a la función `sqrt` de la clase `Math` para calcular una raíz cuadrada.

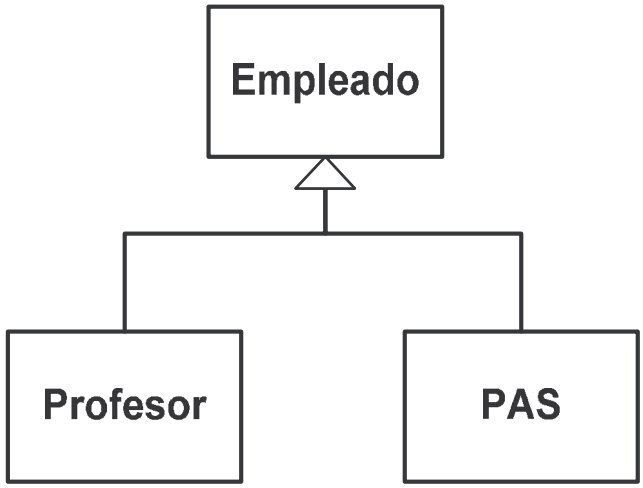
NOTA:

La clase `Math` es una clase "degenerada" que no tiene estado. Es, simplemente, una colección de funciones de cálculo matemático.

Herencia (generalización y especialización)

La relación entre una superclase y sus subclases

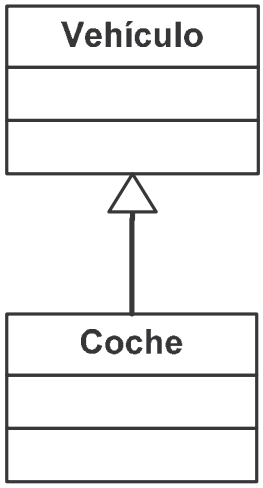
Objetos de distintas clases pueden tener atributos similares y exhibir comportamientos parecidos (p.ej. animales, mamíferos…).




```
public class Empleado
{
   ...
}

public class Profesor extends Empleado
{
   ...
}

public class PAS extends Empleado
{
   ...
}
```

La noción de clase está próxima a la de conjunto:

Generalización y especialización
expresan relaciones de inclusión entre conjuntos.



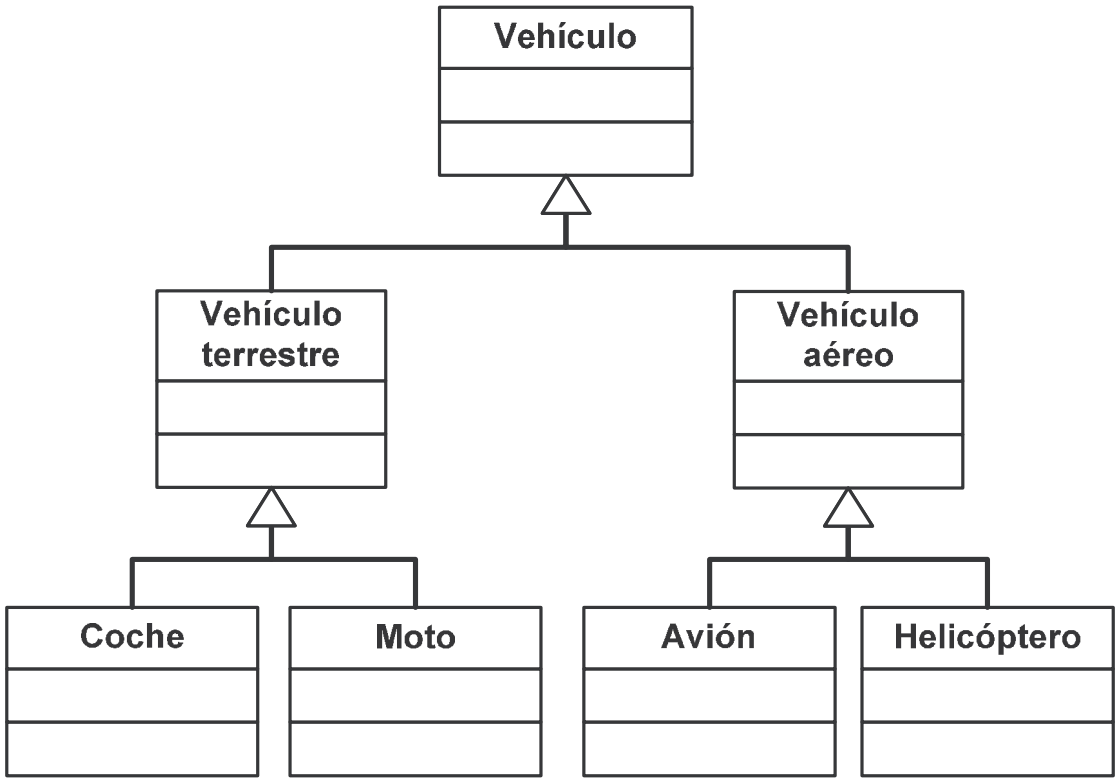
Instancias: **coches ⊂ vehículos**

- Todo coche es un vehículo.
- Algunos vehículos son coches.

Propiedades: **propiedades(coches) ⊃ propiedades(vehículos)**

- Un coche tiene todas las propiedades de un vehículo.
- Algunas propiedades del coche no las tienen todos los vehículos.

Jerarquías de clases



Las clases se organizan
en una estructura jerárquica formando una taxonomía.

- El comportamiento de una categoría más general es aplicable a una categoría particular.

- Las subclases heredan características de las clases de las que se derivan y añaden características específicas que las diferencian.

En el diagrama de clases,
los atributos, métodos y relaciones de una clase se muestran en el nivel más alto de la jerarquía en el que son aplicables.

Visibilidad de los miembros de una clase

Se pueden establecer distintos niveles de encapsulación
para los miembros de una clase (atributos y operaciones)
en función de desde dónde queremos que se pueda acceder a ellos:

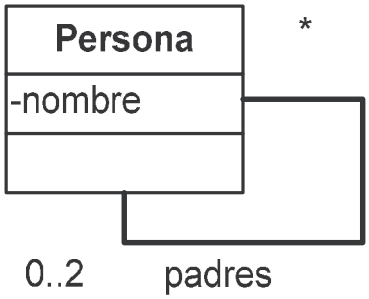
Visibilidad	Significado	Java	UML
Pública	Se puede acceder al miembro de la clase desde cualquier lugar.	`public`	+
Protegida	Sólo se puede acceder al miembro de la clase desde la propia clase o desde una clase que herede de ella.	`protected`	#
Privada	Sólo se puede acceder al miembro de la clase desde la propia clase.	`private`	-

Para encapsular por completo el estado de un objeto,
todos sus atributos se declaran como variables de instancia privadas
(usando el modificador de acceso `private`).

A un objeto siempre se accede a través de sus métodos públicos
(su **interfaz**).

Para usar el objeto no es necesario conocer
qué algoritmos utilizan sus métodos
ni qué tipos de datos se emplean para mantener su estado
(su **implementación**).

Diseño incorrecto




```
public class Persona
{
    public String nombre;
    public Persona padre;
    public Persona madre;
    public ArrayList hijos = new ArrayList();
}
```

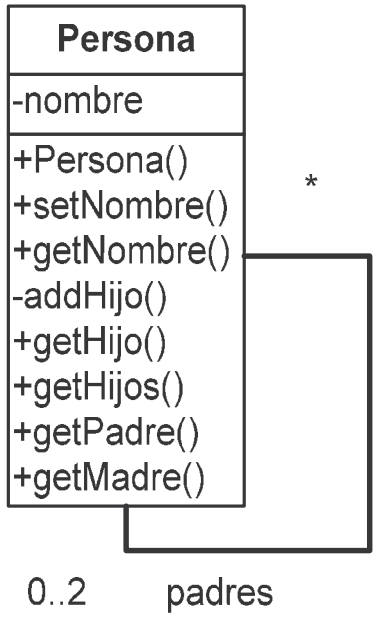
Uso correcto de la clase:

```
Persona juan = new Persona();
Persona carlos = new Persona();
Persona silvia = new Persona();

juan.nombre = "Juan";
carlos.nombre = "Carlos";
silvia.nombre = "Silvia";

juan.padre = carlos;
juan.madre = silvia;
carlos.hijos.add(juan);
silvia.hijos.add(juan);
```

Uso incorrecto de la clase
(pese a ser válido tal como está implementada):

```
juan.padre = carlos;
juan.madre = carlos;
silvia.hijos.add(juan);
juan.hijos.add(juan);
```

Diseño correcto




```
import java.util.ArrayList;

public class Persona
{
  // Variables de instancia privadas
  private String nombre;
  private Persona padre;
  private Persona madre;
  private ArrayList hijos = new ArrayList();

  // Constructores públicos
  public Persona (String nombre)
  {
    this.nombre = nombre;
  }

  public Persona
        (String nombre, Persona padre, Persona madre)
  {
    this.nombre = nombre;
    this.padre = padre;
    this.madre = madre;

    padre.addHijo(this);
    madre.addHijo(this);
  }
```

```
  // Método privado

  private void addHijo (Persona hijo)
  {
    hijos.add(hijo);
  }

  // Métodos públicos
  // p.ej. Acceso a las variables de instancia

  public void setNombre (String nombre)
  {
    this.nombre = nombre;
  }

  public String getNombre ()
  {
    return nombre;
  }
  …
}
```

Con esta implementación, desde el exterior de la clase **no** se pueden modificar las relaciones existentes entre padres e hijos, por lo que estas siempre se mantendrán correctamente si implementamos bien la clase `Persona`.

Ejemplo

```
Persona carlos = new Persona("Carlos");
Persona silvia = new Persona("Silvia");
Persona juan = new Persona("Juan",carlos,silvia);
```

Operación permitida (a través de un método público):

```
juan.setNombre("Antonio");  // Cambio de nombre
```

Operación no permitida (error de compilación):

```
addHijo(Persona) has private access in Persona
juan.addHijo(carlos);
     ^
```

UML

El Lenguaje Unificado de Modelado

Grady Booch, Jim Rumbaugh e Ivar Jacobson

El lenguaje UML es un estándar OMG diseñado para visualizar, especificar, construir y documentar software orientado a objetos.

Un modelo es una simplificación de la realidad.

El modelado es esencial en la construcción de software para…

- Comunicar la estructura de un sistema complejo
- Especificar el comportamiento deseado del sistema
- **Comprender mejor** lo que estamos construyendo
- Descubrir oportunidades de simplificación y reutilización

Un modelo proporciona “los planos” de un sistema y puede ser más o menos detallado, en función de los elementos que sean relevantes en cada momento.

> El modelo ha de capturar “lo esencial”.

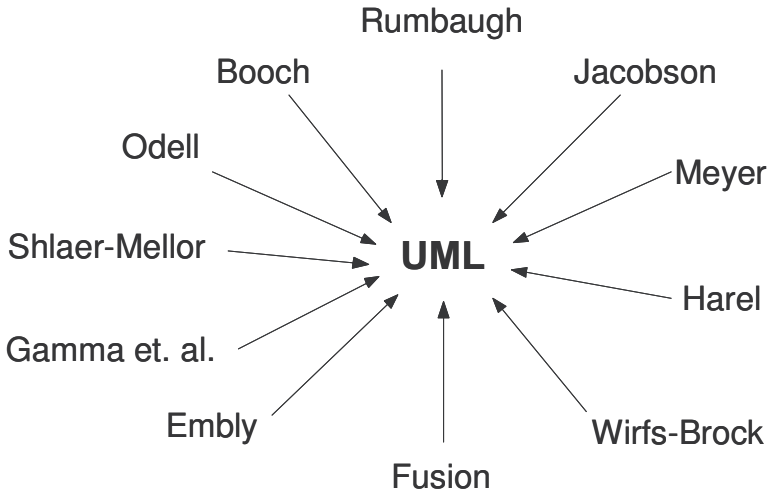
Todo sistema puede describirse desde distintos puntos de vista:

- Modelos estructurales (organización del sistema)
- Modelos de comportamiento (dinámica del sistema)

UML estandariza 9 tipos de diagramas para representar gráficamente un sistema desde distintos puntos de vista.

Ventaja principal de UML

Unifica distintas notaciones previas.

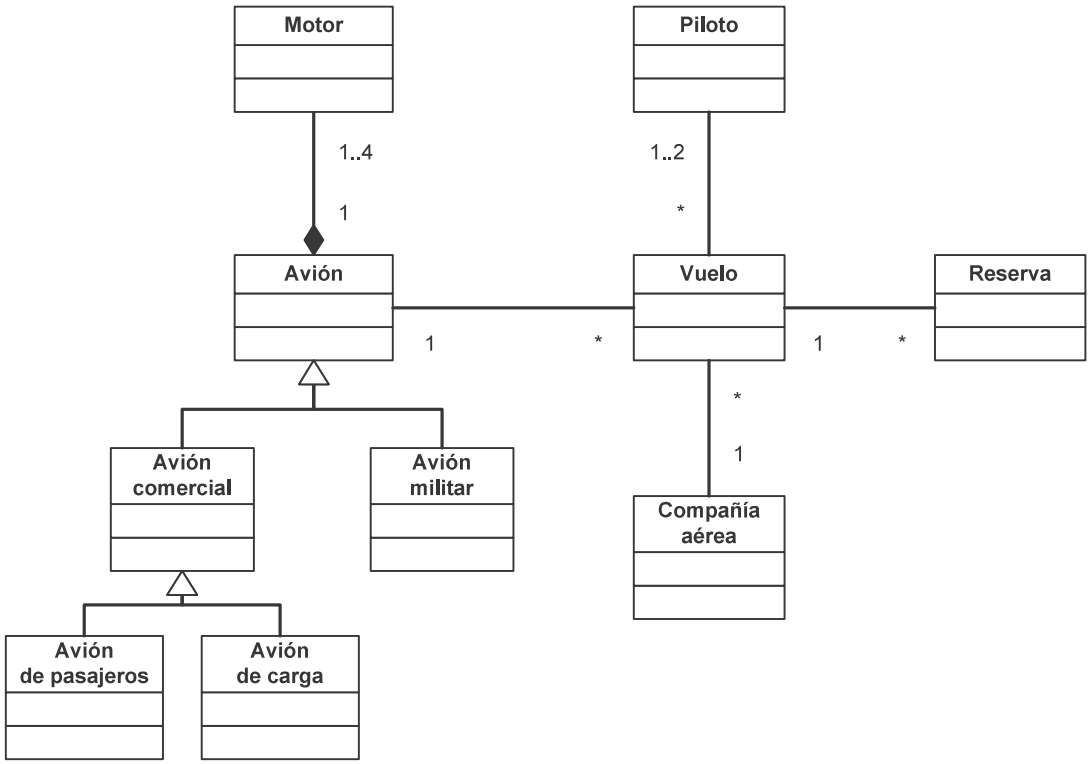


Inconvenientes de UML

- Falta de integración con otras técnicas (p.ej. diseño de interfaces de usuario)
- UML es excesivamente complejo (y no está del todo libre de ambigüedades): “el 80% de los problemas puede modelarse usando alrededor del 20% de UML”

Diagramas de clases

Muestran un conjunto de clases y sus relaciones



Los diagramas de clases proporcionan una perspectiva estática del sistema (representan su diseño estructural).

Notación

Atributos

[visibilidad] nombre [multiplicidad] [**:** tipo [= valor_por_defecto]]

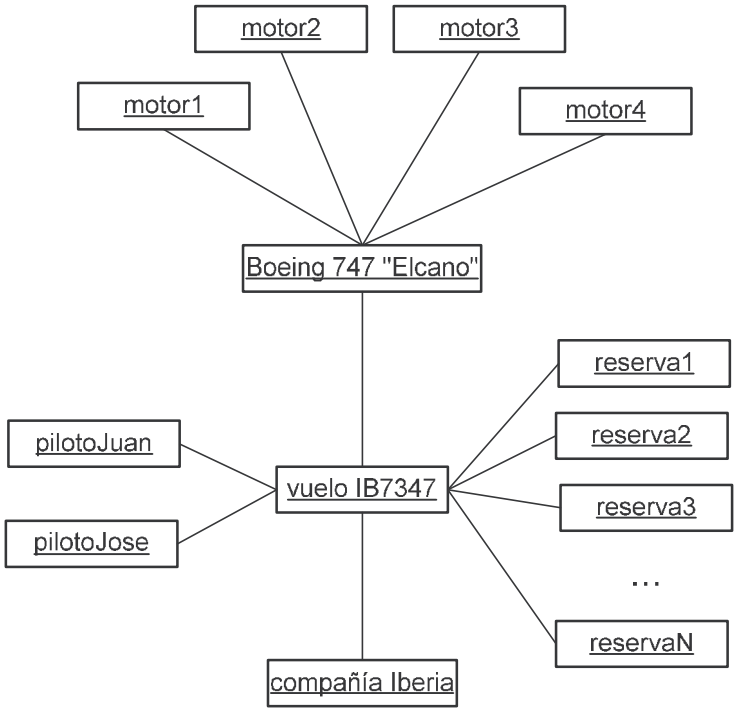
Operaciones

[visibilidad] nombre ([[**in**|**out**] parámetro **:** tipo [**,** …]])[**:**tipo_devuelto]

- Los corchetes indican partes opcionales.
- Visibilidad: privada (-), protegida (#) o pública (+)
- Multiplicidad entre corchetes (p.ej. [2], [0..2], [*], [3..*])
- Parámetros de entrada (in) o de salida (out).

Diagramas de objetos

Muestran un conjunto de objetos y sus relaciones
(una situación concreta en un momento determinado).



Los diagramas de objetos representan instantáneas de instancias
de los elementos que aparecen en los diagramas de clases

Un diagrama de objetos expresa la parte estática de una interacción.

Para ver los aspectos dinámicos de la interacción
se utilizan los diagramas de interacción
(diagramas de secuencia y diagramas de comunicación/colaboración)

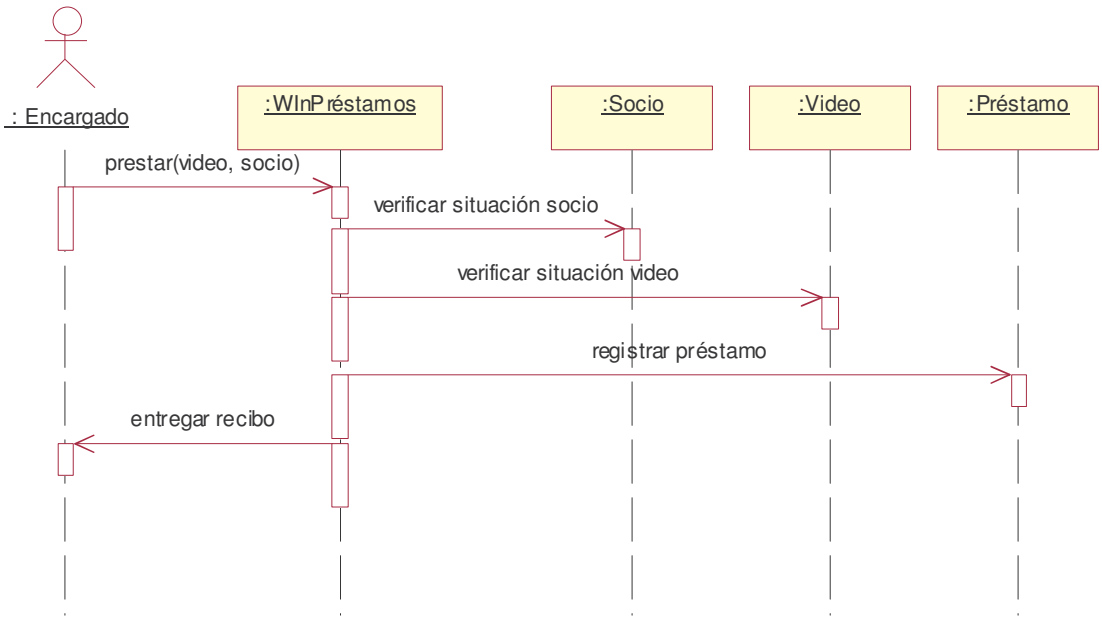
NOTA:
Los identificadores subrayados indican que se trata de objetos.

Diagramas de interacción

Muestran una interacción concreta: un conjunto de objetos y sus relaciones, junto con los mensajes que se envían entre ellos.

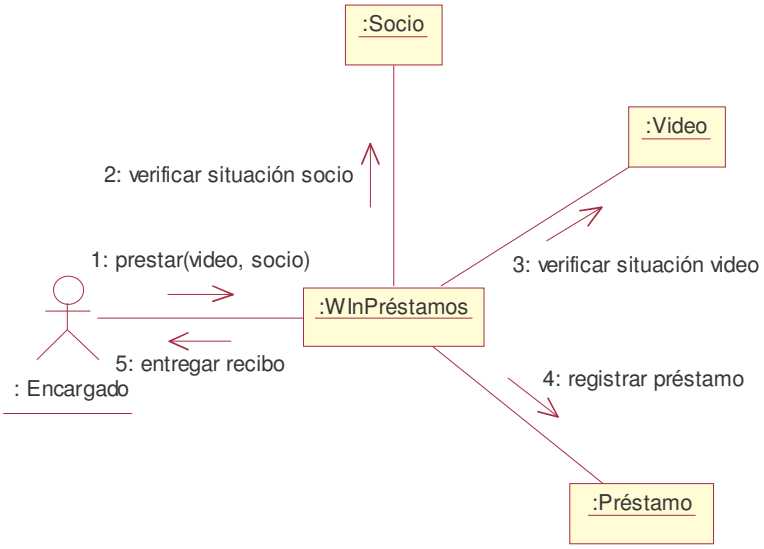
Diagramas de secuencia

Resaltan la ordenación temporal de los mensajes que se intercambian.



Diagramas de comunicación (UML 2.0)
= Diagramas de colaboración (UML 1.x)

Resaltan la organización estructural
de los objetosque intercambian mensajes.



Los diagramas de secuencia y de comunicación son isomorfos:

- Un diagrama de secuencia se puede transformar mecánicamente en un diagrama de comunicación.
- Un diagrama de comunicación se puede transformar automáticamente en un diagrama de secuencia.

Diagramas de secuencia

Muestran la secuencia de mensajes entre objetos durante un escenario concreto (paso de mensajes).

- En la parte superior aparecen los objetos que intervienen.
- La dimensión temporal se indica verticalmente (el tiempo transcurre hacia abajo).
- Las líneas verticales indican el período de vida de cada objeto.
- El paso de mensajes se indica con flechas horizontales u oblicuas (cando existe demora entre el envío y la atención del mensaje).
- La realización de una acción se indica con rectángulos sobre las líneas de actividad del objeto que realiza la acción.

Diagramas de comunicación/colaboración

La distribución de los objetos en el diagrama permite observar adecuadamente la interacción de un objeto con respecto de los demás

- La perspectiva estática del sistema viene dada por las relaciones existentes entre los objetos (igual que en un diagrama de objetos).
- La vista dinámica de la interacción viene indicada por el envío de mensajes a través de los enlaces existentes entre los objetos.

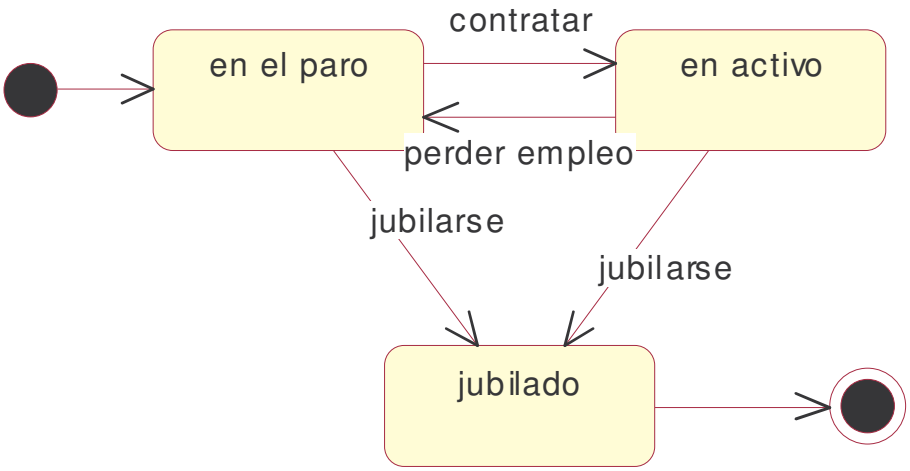
NOTA: Los mensajes se numeran para ilustrar el orden en que se emiten.

Otros diagramas UML para representar aspectos dinámicos del sistema

- **Diagramas de casos de uso**
  (actores y casos de uso del sistema)

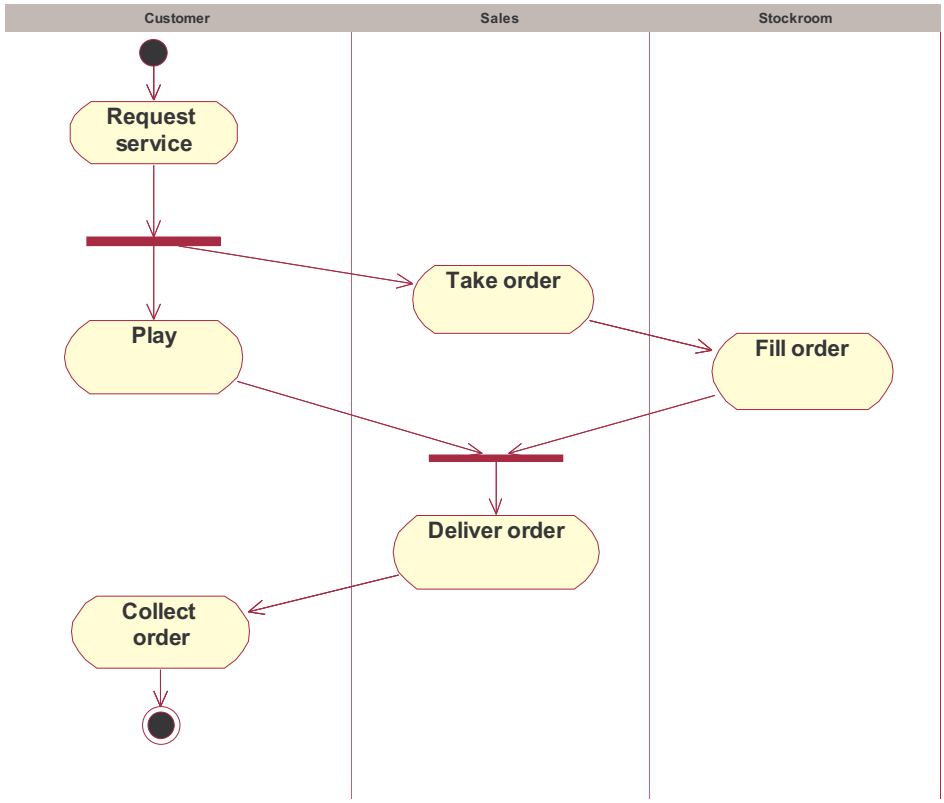
Los diagramas de uso se suelen utilizar en el modelado del sistema desde el punto de vista de sus usuarios para representar las acciones que realiza cada tipo de usuario.

- **Diagramas de estados**
  (estados y transiciones entre estados),



Los diagramas de estados son especialmente importantes para describir el comportamiento de un sistema reactivo (cuyo comportamiento está dirigido por eventos).

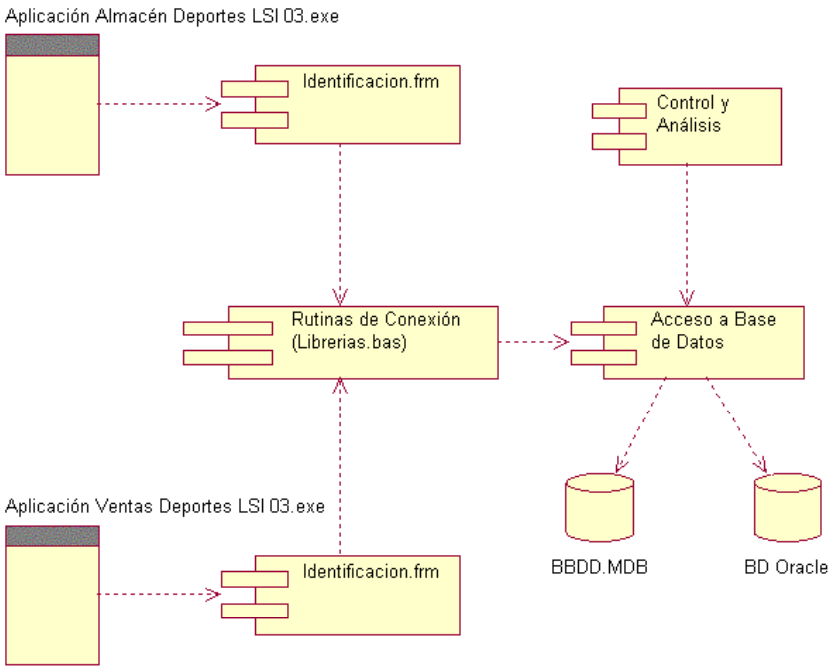
- **Diagramas de actividades**
  (flujo de control en el sistema)



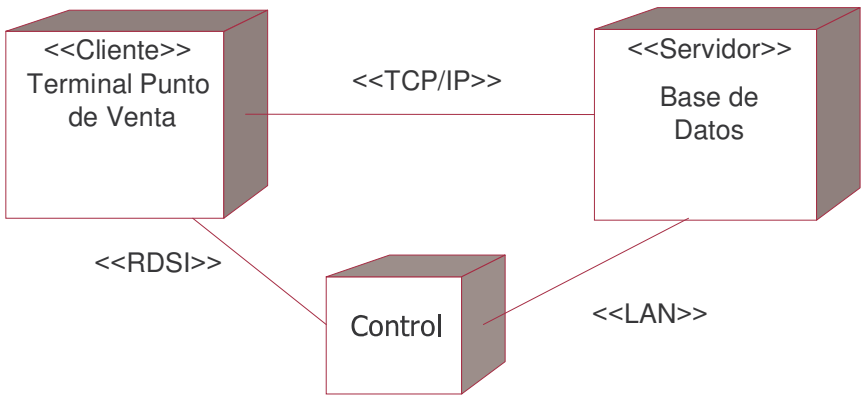
Los diagramas de actividades muestran el orden en el que se van realizando tareas dentro de un sistema (el flujo de control de las actividades).

Diagramas UML
para representar aspectos físicos del sistema

- **Diagramas de componentes**
  (componentes y dependencias entre ellos)
  Organización lógica de la implementación de un sistema



- **Diagramas de despliegue**
  (nodos de procesamiento y componentes)
  Configuración del sistema en tiempo de ejecución



Referencias

Páginas web

http://www.uml.org/
Página oficial de UML, uno de los estándares promovidos por el OMG.

http://www.cetus-links.org/oo_uml.html
Colección de enlaces relacionados con UML.

http://www.agilemodeling.com/essays/umlDiagrams.htm
Información práctica acerca de todos los diagramas UML 2

http://www.ootips.org/
Ideas clave en programación orientada a objetos.

Libros

Martin Fowler: *"UML Distilled:*
A Brief Guide to the Standard Object Modeling Language"
3rd edition. Addison-Wesley, 2004. ISBN 0321193687

Grady Booch et al.:
"Object-Oriented Analysis and Design with Applications"
3rd edition. Addison-Wesley, 2004. ISBN 020189551X

Craig Larman: *"Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process"*
2nd edition. Prentice-Hall, 2001. ISBN 0130925691

Robert C. Martin:
"Agile Software Development: Principles, Patterns, and Practices"
Prentice-Hall, 2003. ISBN 0135974445

…

Programación orientada a objetos
Relación de ejercicios

1. Proponga tres ejemplos de objetos del mundo real:

   - Para cada uno de ellos, determine la clase a la que pertenecen.
   - Asóciele a cada clase un identificador descriptivo adecuado.
   - Enumere varios atributos y operaciones para cada una de las clases.
   - Represente gráficamente las clases utilizando la notación UML.
   - A partir de los diagramas UML, escriba el código necesario para definir las clases utilizando el lenguaje de programación Java.

2. Rellene los huecos en las siguientes afirmaciones:

   a. Los objetos encapsulan _____ y _____.

   b. Los objetos se comunican entre sí pasándose _____.

   c. Para comunicarse con un objeto concreto, no es necesario conocer su _____, basta con saber cuál es su _____.

   d. Pueden existir varios tipos de relaciones entre clases: _____, _____ y _____.

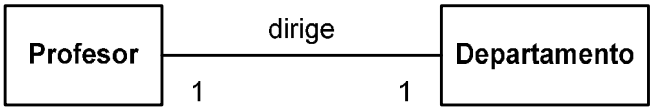
   e. Los lenguajes de programación orientada a objetos utilizan relaciones de _____ para derivar nuevas clases a partir de clases base.

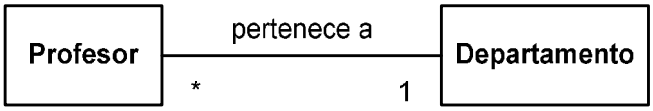
   f. _____ define una notación gráfica estándar para representar diseños orientados a objetos.

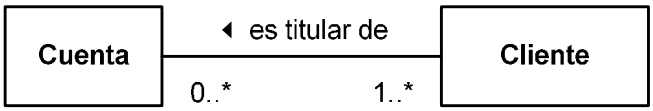
   g. Las clases se definen en Java en ficheros de texto con la extensión _____

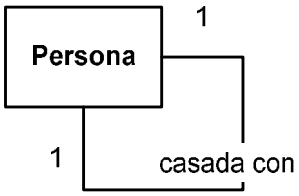
   h. El compilador de Java genera ficheros con extensión _____ al compilar un fichero de código fuente escrito en Java.

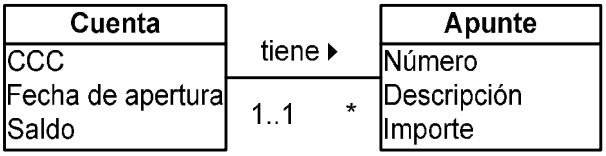
3. Definir adecuadamente las clases en Java que se derivan de los siguientes diagramas de clases UML:

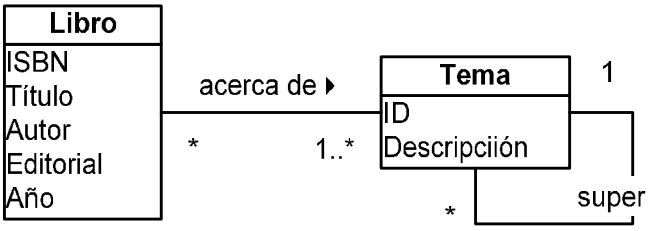


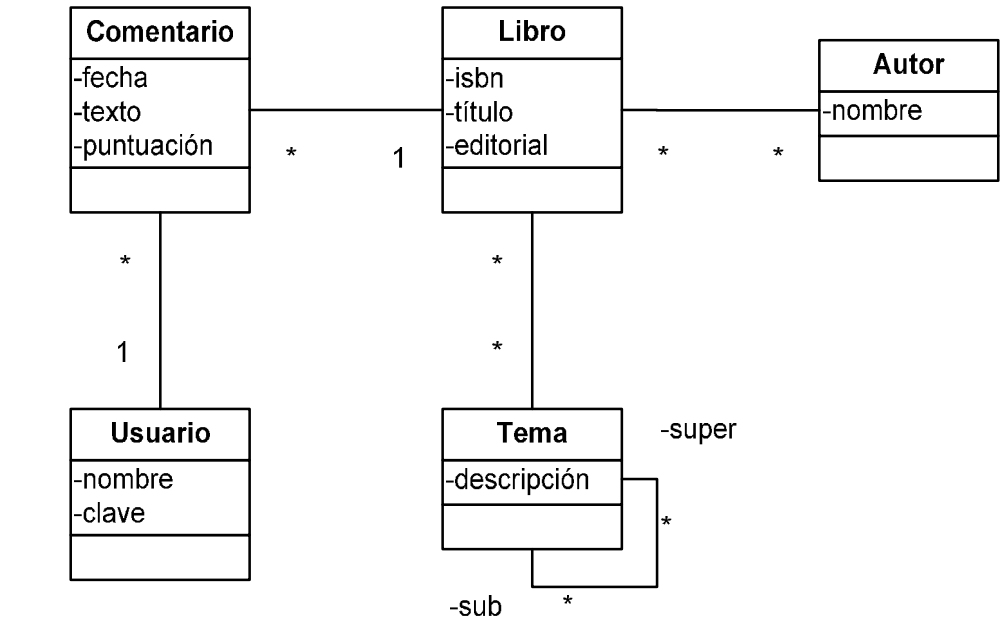


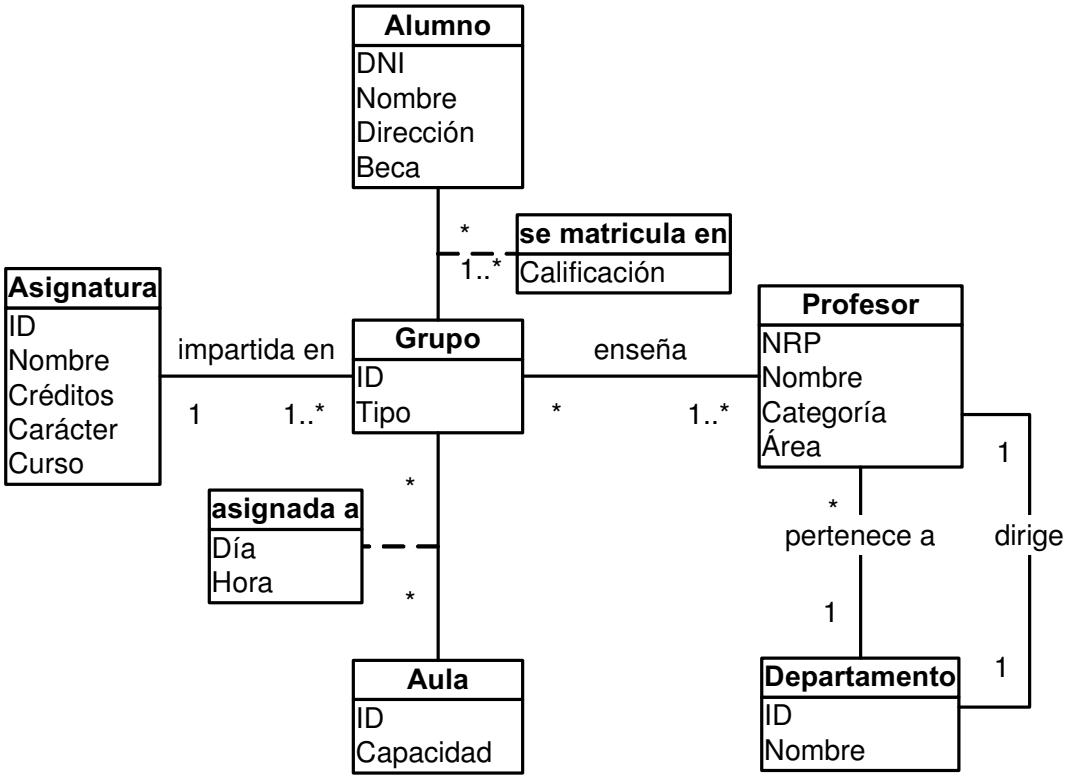






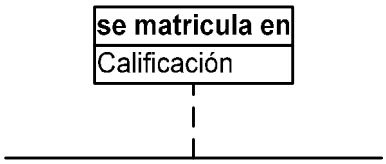



Comentario
-fecha
-texto
-puntuación
Libro
-isbn
-título
-editorial
Autor
-nombre
Usuario
-nombre
-clave
Tema
-descripción
-super
-sub
*
1


Alumno
DNI
Nombre
Dirección
Beca
se matricula en
Calificación
1..*
Asignatura
ID
Nombre
Créditos
Carácter
Curso
impartida en
1
1..*
Grupo
ID
Tipo
enseña
*
1..*
Profesor
NRP
Nombre
Categoría
Área
asignada a
Día
Hora
pertenece a
dirige
Aula
ID
Capacidad
Departamento
ID
Nombre

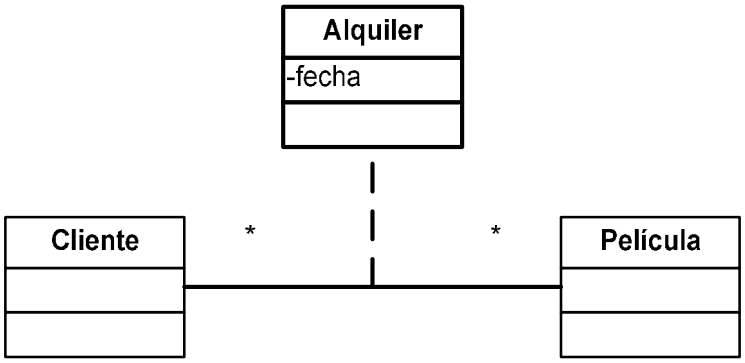
Nota: CLASES ASOCIACIÓN

Las clases asociación (como "se matricula en") se emplean para indicar que la asociación existente entre dos clases tiene atributos propios:



En realidad, las clases asociación de un diagrama de clases UML son clases convencionales cuyo único papel consiste en relacionar objetos de otras clases (no tienen comportamiento propio)

Ejemplo



La fecha del alquiler no es un atributo del cliente ni de la película,
es algo específico del hecho de alquilar la película.

```
class Cliente
…
class Pelicula
…

class Alquiler
{
  private Cliente  cliente;
  private Pelicula peli;
  private DateTime fecha;

  public Alquiler
         (Cliente cliente, Pelicula peli, DateTime fecha)
  {
    this.cliente = cliente;
    this.peli = peli;
    this.fecha = fecha;
  }
  …
}
```

Modularización

Uso de subprogramas

Razones válidas para crear un subprograma
Pasos para escribir un subprograma
Acerca del nombre de un subprograma

Métodos

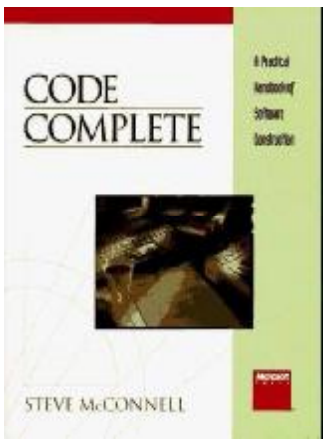
Definición de los métodos: cabecera, cuerpo y signatura
Uso de los métodos
Paso de parámetros
Devolución de resultados (sentencia `return`)
Constructores (la palabra reservada `this`)
Métodos estáticos

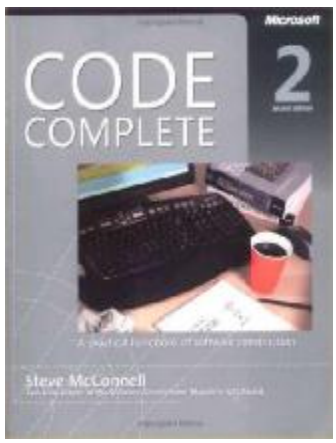
Ámbito de las variables

Cohesión y acoplamiento

Bibliografía

Steve McConnell: *"Code Complete"*.
Microsoft Press, 2004 [2ª edición] ISBN 0735619670.
Microsoft Press, 1994 [1ª edición] ISBN 1556154844.





Uso de subprogramas

Los lenguajes de programación permiten descomponer un programa complejo en distintos subprogramas:

- **Funciones y procedimientos**
  en lenguajes de programación estructurada

- **Métodos**
  en lenguajes de programación orientada a objetos

Razones válidas para crear un subprograma

Reducir la complejidad del programa ("divide y vencerás").

Eliminar código duplicado.

Mejorar la legibilidad del código.

Limitar los efectos de los cambios (aislar aspectos concretos).

Ocultar detalles de implementación
(*ocultación de información*)
p.ej. algoritmos complejos

Promover la reutilización de código
p.ej. componentes reutilizables y familias de productos

Facilitar la adaptación del código a nuevas necesidades
p.ej. interfaces de usuario

Mejorar la portabilidad del código.

Pasos para escribir un subprograma

1. Definir el problema que el subprograma ha de resolver.

2. Darle un nombre no ambiguo al subprograma.

3. Decidir cómo se puede probar el funcionamiento del subprograma (de esta forma, desde el comienzo se piensa en cómo se utilizará el subprograma, lo que tiende a mejorar el diseño de un interfaz adecuado para el subprograma).

4. Escribir la declaración del subprograma:
   - La cabecera de la función en lenguajes estructurados.
   - La cabecera del método en lenguajes orientados a objetos.

5. Buscar el algoritmo más adecuado para resolver el problema.

6. Escribir los pasos principales del algoritmo como comentarios en el texto del programa.

7. Rellenar el código correspondiente a cada comentario.

8. Revisar mentalmente cada fragmento de código.

9. Repetir los pasos anteriores hasta quedar completamente satisfecho.

El nombre de un subprograma

Al crear un subprograma hemos de darle un nombre:

- Cuando el subprograma no devuelve ningún valor (procedimientos y métodos `void`):

  El nombre del subprograma suele estar formado por un verbo seguido, opcionalmente, del nombre de un objeto.

  Ejemplos:
  ```
  ingresar
  realizarTransferencia
  abonarImpuestos
  …
  ```

  El subprograma se encargará de realizar una operación independiente con respecto al resto del programa.

- Cuando el subprograma devuelve un valor (funciones y métodos):

  El nombre del subprograma suele ser una descripción del valor devuelto por la función o el método.

  Ejemplos:
  ```
  saldoActual
  saldoMedio
  …
  ```

  El subprograma se usará habitualmente para obtener un valor que emplearemos dentro de una expresión.

Observaciones

El nombre debe describir todo lo que hace el subprograma.

Se deben evitar nombres genéricos que no dicen nada (vg. `calcular`)

Se debe ser consistente en el uso de convenciones.

Métodos

Los métodos definen
el comportamiento de los objetos de una clase dada
(lo que podemos hacer con los objetos de esa clase)

Los métodos exponen la interfaz de una clase.

Un método define la secuencia de sentencias
que se ejecuta para llevar a cabo una operación:

La implementación de la clase se oculta del exterior.

Los métodos…

Nos dicen cómo hemos de usar los objetos de una clase.

Nos permiten cambiar la implementación de una clase sin tener que modificar su interfaz (esto es, sin tener que modificar el código que utiliza objetos de la clase cuya implementación cambiamos)

Ejemplo:
Utilizar un algoritmo más eficiente
para resolver un problema concreto
sin tener que tocar el código del resto del programa.

Definición de métodos

Sintaxis en Java

modificadores tipo nombre (*parámetros*)
{
 cuerpo
}

La estructura de un método se divide en:

- **Cabecera** (determina su interfaz)

  modificadores tipo nombre (*parámetros*)

- **Cuerpo** (define su implementación)

```
{
  // Declaraciones de variables
  ...
  // Sentencias ejecutables
  ...
  // Devolución de un valor (opcional)
  ...
}
```

En el cuerpo del método se implementa el algoritmo necesario para realizar la tarea de la que el método es responsable.

El cuerpo de un método se puede interpretar como una caja negra que contiene su implementación:

El método oculta los detalles de implementación.

Cuando utilizamos un método, sólo nos interesa su interfaz.

Ejemplo
El punto de entrada a una aplicación escrita en Java

```
public static void main (String[] args)
{
  ...
}
```

- Como todo en Java, ha de ser un miembro de una clase (esto es, estar definido en el interior de una clase).

- El modificador de acceso `public` indica que se puede acceder a este miembro de la clase desde el exterior de la clase.

- El modificador `static` indica que se trata de un método de clase (un método común para todos los objetos de la clase).

- La palabra reservada `void` indica que, en este caso el método `main` no devuelve ningún valor.

  En general, no obstante, los métodos son capaces de devolver un valor al terminar su ejecución.

- Los paréntesis nos indican que se trata de un método: Lo que aparece entre paréntesis son los parámetros del método (en este caso, un vector de cadenas de caracteres, que se representan en Java con objetos de tipo `String`).

- El cuerpo del método va delimitado por llaves { }.

CONVENCIÓN
El texto correspondiente al código que se ejecuta al invocar un método se sangra con respecto a la posición de las llaves que delimitan el cuerpo del método.

La cabecera de un método

La cabecera de un método determina su interfaz

- **Modificadores de acceso** (p.ej. `public` o `private`)
  Determinan desde dónde se puede utilizar el método.

- **Tipo devuelto** (cualquier tipo primitivo, no primitivo o `void`)
  Indica de que tipo es la salida del método, el resultado que se obtiene tras llamar al método desde el exterior.

  NOTA:
  `void` se emplea cuando el método no devuelve ningún valor.

- **Nombre del método**
  Identificador válido en Java

  CONVENCIÓN:
  En Java, los nombres de métodos comienzan con minúscula.

- **Parámetros formales**
  Entradas que necesita el método para realizar la tarea de la que es responsable.

  MÉTODOS SIN PARÁMETROS:
  Cuando un método no tiene entradas, hay que poner `()`

El cuerpo de un método

El cuerpo de un método define su implementación:

NB: Como cualquier bloque de código en Java,
el cuerpo de un método ha de estar delimitado por llaves { }

La signatura de un método

El nombre de un método, los tipos de sus parámetros y el orden de los mismos definen la signatura de un método.

- Los modificadores y el tipo del valor devuelto por un método **no** forman parte de la signatura del método.

Sobrecarga

Lenguajes como Java permiten que existan distintos métodos con el mismo nombre siempre y cuando su signatura no sea idéntica (algo que se conoce con el nombre de *sobrecarga*)

Ejemplo

```
System.out.prinln(…);
    - System.out.println()
    - System.out.println(boolean)
    - System.out.println(char)
    - System.out.println(char[])
    - System.out.println(double)
    - System.out.println(float)
    - System.out.println(int)
    - System.out.println(long)
    - System.out.println(Object)
    - System.out.println(String)
```

No es válido definir dos métodos con el mismo nombre
que difieran únicamente por el tipo del valor que devuelven.

De todas formas, no conviene abusar demasiado de esta prestación del lenguaje, porque resulta propensa a errores
(en ocasiones, creeremos estar llamando a una "versión"
de un método cuando la que se ejecuta en realidad es otra).

NOTA: En la creación de *constructores*
sí es importante disponer de esta característica.

Uso de métodos

Para enviarle un mensaje a un objeto,
invocamos (llamamos a) uno de sus métodos:

> La llamada a un método de un objeto
> le indica al objeto que delegamos en él
> para que realice una operación de la que es reponsable.

A partir de una referencia a un objeto,
podermos llamar a uno de sus métodos con el **operador .**

Tras el nombre del método, entre paréntesis,
se han de indicar sus parámetros (si es que los tiene).

El método podemos usarlo cuantas veces queramos.

MUY IMPORTANTE: De esta forma, evitamos la existencia de código duplicado en nuestros programas.

Ejemplo

```
Cuenta cuenta = new Cuenta();

cuenta.mostrarMovimientos();
```

Obviamente, el objeto debe existir antes de que podamos invocar uno de sus métodos. Si no fuese así, en la ejecución del programa obtendríamos el siguiente error:

```
java.lang.NullPointerException
    at ...
```

al no apuntar la referencia a ningún objeto (`null` en Java).

Ejemplo de ejecución paso a paso
Cuando se invoca un método, el ordenador pasa a ejecutar las sentencias definidas en el cuerpo del método:

```
public class Mensajes
{
  public static void main (String[] args)
  {
    mostrarMensaje("Bienvenida");
    // ...
    mostrarMensaje("Despedida");
  }

  private static void mostrarMensaje (String mensaje)
  {
    System.out.println("*** " + mensaje + " ***");
  }
}
```

Al ejecutar el programa (con `java Mensajes`):

1. Comienza la ejecución de la aplicación, con la primera sentencia especificada en el cuerpo del método `main`.
2. Desde `main`, se invoca `mostrarMensaje` con *"Bienvenida"* como parámetro.
3. El método `mostrarMensaje` muestra el mensaje de bienvenida decorado y termina su ejecución.
4. Se vuelve al punto donde estábamos en `main` y se continúa la ejecución de este método.
5. Justo antes de terminar, volvemos a llamar a `mostrarMensaje` para mostrar un mensaje de despedida.
6. `mostrarMensaje` se vuelve a ejecutar, esta vez con *"Despedida"* como parámetro, por lo que esta vez se muestra en pantalla un mensaje decorado de despedida.
7. Se termina la ejecución de `mostrarMensaje` y se vuelve al método desde donde se hizo la llamada (`main`).
8. Se termina la ejecución del método `main` y finaliza la ejecución de nuestra aplicación.

Los parámetros de un subprograma

Un método puede tener parámetros:

A través de los parámetros se especifican los datos de entrada que requiere el método para realizar su tarea.

Los parámetros definidos en la cabecera del método se denominan **parámetros formales**.

Para cada parámetro, hemos de especificar tanto su tipo como un identificador que nos permita acceder a su valor actual en la implementación del método.

Cuando un método tiene varios parámetros, los distintos parámetros se separan por comas en la cabecera del método.

En la definición de un método,
la lista de parámetros formales de un método establece:

- Cuántos parámetros tiene el método
- El tipo de los valores que se usarán como parámetros
- El orden en el que han de especificarse los parámetros

En la invocación de un método,
se han de especificar los valores concretos para los parámetros.

Los valores que se utilizan como parámetros
al invocar un método se denominan
parámetros actuales (o "argumentos").

Cuando se efectúa la llamada a un método, los valores indicados como parámetros actuales se asignan a sus parámetros formales.

En la implementación del método, podemos utilizar entonces los parámetros del método como si fuesen variables normales (y de esta forma acceder a los valores concretos con los que se realiza cada llamada al método).

Obviamente, el número y tipo de los parámetros indicados en la llamada al método ha de coincidir con el número y tipo de los parámetros especificados en la definición del método.

En Java, todos los parámetros se pasan por valor:

> Al llamar a un método,
> el método utiliza una copia local de los parámetros
> (que contiene los valores con los cuales fue invocado).

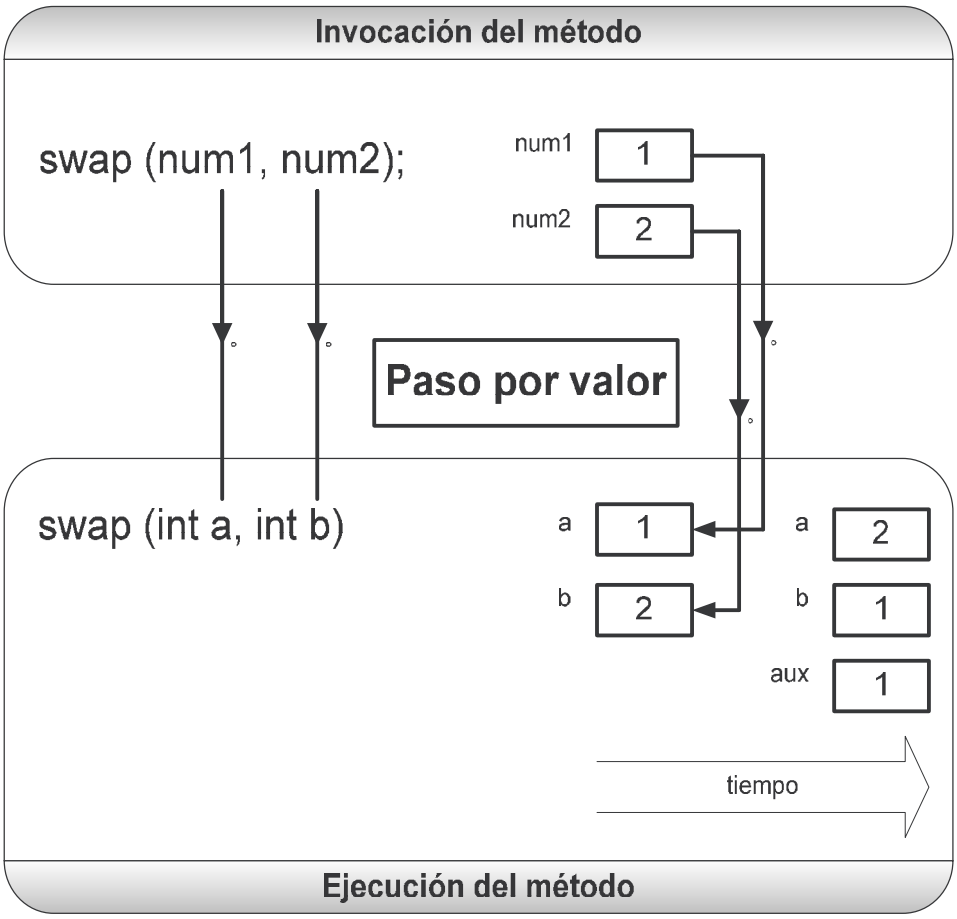
¡OJO! Como las variables de tipos no primitivos son, en realidad, referencias a objetos, lo que se copia en este caso es la referencia al objeto (no el objeto en sí).

Como consecuencia, podemos modificar el estado de un objeto recibido como parámetro si invocamos métodos de ese objeto que modifiquen su estado.

La referencia al objeto no cambia, pero sí su estado.

Ejemplo: Intercambio incorrecto de valores

```
public void swap (int a, int b)   // Definición
{
  int aux;
  aux = b;
  a = b;
  b = aux;
}
...
swap(a,b);                        // Invocación
```



- Los valores de `num1` y `num2` se copian en `a` y `b`.
- La ejecución del método `swap` no afecta ni a `num1` ni a `num2`.

`swap` no intercambia los valores de las variables porque el intercambio se hace sobre las **copias locales** de los parámetros de las que dispone el método, no sobre las variables originales.

Convenciones

- Si varios métodos utilizan los mismos parámetros, éstos han de ponerse siempre en el mismo orden.

  De esta forma, resulta más fácil de recordar la forma correcta de usar un método.

- No es aconsejable utilizar los parámetros de una rutina como si fuesen variables locales de la rutina.

  En otras palabras, los parámetros no los utilizaremos para almacenar resultados parciales.

- Se han de documentar claramente las suposiciones que se hagan acerca de los valores permitidos para los distintos parámetros de un método.

  Esta información debería figurar en la documentación del código realizada con la herramienta `javadoc`.

- Sólo se deben incluir los parámetros que realmente necesite el método para efectuar su labor.

  Si un dato no es necesario para realizar un cálculo, no tiene sentido que tengamos que pasárselo al método.

- Las dependencias existentes entre distintos métodos han de hacerse explícitas mediante el uso de parámetros.

  Si evitamos la existencia de variables globales (datos compartidos entre distintos módulos), el código resultante será más fácil de entender.

Devolución de resultados: La sentencia **`return`**

Cuando un método devuelve un resultado, la implementación del método debe terminar con una sentencia `return`:

```
return expresión;
```

Como es lógico, el tipo de la *expresión* debe coincidir con el tipo del valor devuelto por el método, tal como éste se haya definido en la cabecera del método.

Ejemplo

```
public static float media (float n1, float n2)
{
  return (n1+n2)/2;
}

public static void main (String[] args)
{
  float resultado = media (1,2);

  System.out.println("Media = " + resultado);
}
```

El compilador de Java comprueba que exista una sentencia `return` al final de un método que deba devolver un valor.
Si no es así, nos dará el error

```
Missing return statement
```

El compilador también detecta si hay algo después de la sentencia `return` (un error porque la sentencia return finaliza la ejecución de un método y nunca se ejecuta lo que haya después):

```
Unreachable statement
```

Ejemplo
Figuras geométricas

```
// Title:       Geometry
// Version:     0.0
// Copyright:   2004
// Author:      Fernando Berzal
// E-mail:      berzal@acm.org

public class Point
{
  // Variables de instancia

  private double x;
  private double y;

  // Constructor

  public Point (double x, double y)
  {
    this.x = x;
    this.y = y;
  }

  // Métodos

  public double distance (Point p)
  {
    double dx = this.x - p.x;
    double dy = this.y - p.y;

    return Math.sqrt(dx*dx+dy*dy);
  }

  public String toString ()
  {
    return "(" + x + "," + y + ")";
  }
}
```

NB: La interfaz de la clase habría que documentarlo añadiendo los correspondientes comentarios `javadoc`.

```
public class Circle
{
  private Point  centro;
  private double radio;

  // Constructor

  public Circle (Point centro, double radio)
  {
    this.centro = centro;
    this.radio  = radio;
  }

  // Métodos

  public double area ()
  {
    return Math.PI*radio*radio;
  }

  public boolean isInside (Point p)
  {
     return ( centro.distance(p) < radio );
  }

  public String toString ()
  {
     return "Círculo con radio " + radio
          + " y centro en " + centro;
  }
}
```

Ejemplo de uso

```
public static void main(String[] args)
{
  Circle fig    = new Circle( new Point(0,0), 10);
  Point  dentro = new Point (3,3);
  Point  fuera  = new Point (10,10);

  System.out.println(figura);
  System.out.println(dentro+"? "+fig.isInside(dentro));
  System.out.println(fuera +"? "+fig.isInside(fuera) );
}
```

Constructores. La palabra reservada `this`

Los constructores son métodos especiales que sirven para inicializar el estado de un objeto cuando lo creamos con el operador `new`

- Su nombre ha de coincidir coincide con el nombre de la clase.
- Por definición, no devuelven nada.

```
public class Point
{
  // Variables de instancia
  private double x;
  private double y;

  // Constructor
  public Point (double x, double y)
  {
    this.x = x;
    this.y = y;
  }

  // Acceso a las coordenadas

  public double getX ()
  {
    return x;
  }

  public double getY ()
  {
    return y;
  }
}
```

NOTA:
La palabra reservada `this` permite acceder al objeto sobre el que se ejecuta el método.

La clave de implementar la clase `Point` de esta forma
(y no dar acceso directo a las variables de instancia)
es que podemos cambiar la implementación de `Point`
para usar coordenadas polares
y la implementación de las clases que trabajan con puntos
(p.ej. `Circle`) no hay que tocarla:

```
public class Point
{
  // Variables de instancia

  private double r;
  private double theta;

  // Constructor

  public Point (double x, double y)
  {
    r     = Math.sqrt(x*x+y*y);
    theta = Math.atan2(y,x);
  }

  // Acceso a las coordenadas

  public double getX ()
  {
    return r*Math.cos(theta);
  }

  public double getY ()
  {
    return return r*Math.sin(theta);
  }
}
```

Gracias a la encapsulación,
podemos crear componentes reutilizables
cuya evolución no afectará al resto del sistema.

Podemos definir varios constructores para poder inicializar un objeto de distintas formas (siempre y cuando los constructores tengan signaturas diferentes);

```
public class Contacto
{
  private String nombre;
  private String email;

  public Contacto (String nombre)
  {
    this.nombre = nombre;
  }

  public Contacto (String nombre, String email)
  {
    this.nombre = nombre;
    this.email = email;
  }

}
```

Ejemplo de uso:

```
public class ContactoTest
{
  public static void main(String[] args)
  {
    Contacto nico = new Contacto ("Nicolás");

    Contacto juan = new Contacto ("Juan",
                                  "juan@acm.org");
    ...
  }
}
```

Constructor de copia
Un constructor que recibe como parámetro
un objeto de la misma clase que la del constructor

Otro ejemplo de uso de la palabra reservada `this`
consiste en llamar a un constructor desde otro de los constructores
(algo que, de hacerse, siempre ha de ponerse
al comienzo de la implementación del constructor)

```
public class Contacto
{
  private String nombre;
  private String email;

  public Contacto (String nombre)
  {
    this(nombre, "");
  }

  public Contacto (String nombre, String email)
  {
    this.nombre = nombre;
    this.email = email;
  }

  // Constructor de copia

  public Contacto (Contacto otro)
  {
    this (otro.nombre, otro.email);
  }
}
```

RECORDATORIO: **Encapsulación**
Se consigue con los modificadores de acceso `public` y `private`.

- Las variables de instancia de una clase suelen definirse como privadas (con la palabra reservada `private`).

- Los métodos públicos muestran la interfaz de la clase.

- Pueden existir métodos no públicos que realicen tareas auxiliares manteniendo la separación entre interfaz e implementación.

Métodos estáticos

Los métodos estáticos pertenecen a la clase
(no están asociados a un objeto particular de la clase)

Ya hemos visto algunos ejemplos:

```
Math.pow(x,y)

public static void main (String[] args) …
```

Para invocar un método estático,
usamos directamente el nombre de la clase (p.ej. `Math`)

No tenemos que instanciar antes un objeto de la clase.

`main` es un método estático
de forma que la máquina virtual Java
puede invocarlo sin tener que crear antes un objeto.

Como los métodos estáticos no están asociados a objetos concretos,
no pueden acceder a las variables de instancia de un objeto
(las cuales pertenecen a objetos particulares).

Variables estáticas = Variables de clase

Las clases también pueden tener variables que se suelen emplear para representar constantes y variables globales a las cuales se pueda acceder desde cualquier parte de la aplicación (aunque esto último no es muy recomendable).

Ejemplos `System.out`

Colores predefinidos:
`Color.black`, `Color.red`...

Como es lógico,
los métodos estáticos sólo pueden acceder a variables estáticas.

```
public class Mensajes
{
  private String mensaje = "Hola";  // ¡Error!

  public static void main (String[] args)
  {
    mostrarMensaje(mensaje);
  }

  private static void mostrarMensaje (String mensaje)
  {
    System.out.println("*** " + mensaje + " ***");
  }
}
```

El programa anterior funcionaría correctamente
si hubiésemos declarado `mensaje` como una variable estática:

```
  private static String mensaje = "Hola";
```

> NOTA: No es aconsejable declarar variables estáticas
> salvo para definir constantes, como sucede en la clase `Math`.

Métodos estáticos y variables estáticas en la clase `Math`

- Constantes matemáticas: `Math.PI`, `Math.E`
- Métodos estáticos: Funciones trigonométricas (`sin, cos, tan, acos, asin, atan`), logaritmos y exponenciales (`exp, log, pow, sqrt`), funciones de redondeo (`ceil, floor, round` [`== Math.floor(x+0.5)`], `rint` [al entero más cercano, se coge el par]), máximo y mínimo (`max, min`), valor absoluto (`abs`), generación de número pseudoaleatorios (`random`)…

Ámbito de las variables

Una **variable de instancia**
es una variable definida para las instancias de una clase
(cada objeto tiene su propia copia de la variable de instancia).

Una **variable estática**
es una variable definida para la clase
(compartida entre todas las instancias de una clase).

Una **variable local**
es una variable definida dentro del cuerpo de un método.

El ámbito de una variable es la parte del programa
en la que podemos hacer referencia a la variable

- El ámbito de una variable de instancia abarca todos los métodos no estáticos de una clase:
  - Cuando es privada, todos los métodos pueden acceder al valor almacenado en la variable de instancia.
  - Cuando es pública, se puede acceder a ella desde cualquier lugar en el que se disponga de una referencia a un objeto de la clase.

- El ámbito de una variable estática:
  - Si es privada, cubre todos los métodos estáticos de la clase en que está definida.
  - Si es pública, abarca todos los métodos estáticos de todas las clases que formen parte de la aplicación.

- El ámbito de una variable local comienza en su declaración y termina donde termina el bloque de código ({}) que contiene la declaración.

Uso de variables locales

En Java, las declaraciones se pueden poner en cualquier parte del código de un método, no necesariamente al principio:

```
void method ()
{
  int i=0;              // Declara e inicializa i

  while (i<10) {        // i está definido aquí
    int j=0;            // Declara j
    ...                 // i y j definidos
  }                     // j ya no está definido

  System.out.print(i); // i todavía está definido
}                      // i deja de estar definido
```

De todas formas, nosotros declararemos siempre todas las variables locales al comienzo del cuerpo del método.

- Las variables locales han de declararse antes de utilizarse.

- Se pueden declarar variables locales con el mismo nombre en diferentes bloques de código.

  Incluso se podrían declarar en bloques de código no anidados dentro de un mismo método, aunque no es recomendable hacerlo.

IMPORTANTE

Las variables de instancia se inicializan automáticamente al crear un objeto (a `0` o `null`), mientras que las variables locales de un método tenemos que inicializarlas nosotros antes de usarlas.

Cohesión y acoplamiento

Cohesión
Medida del grado de identificación
de un módulo con una función concreta.

Cohesión aceptable (fuerte)

COHESIÓN FUNCIONAL (un módulo realiza una única acción).

COHESIÓN SECUENCIAL (un módulo contiene acciones que han de realizarse en un orden particular sobre unos datos concretos).

COHESIÓN DE COMUNICACIÓN (un módulo contiene un conjunto de operaciones que se realizan sobre los mismos datos).

COHESIÓN TEMPORAL (las operaciones se incluyen en un módulo porque han de realizarse al mismo tiempo; p.ej. inicialización).

Cohesión inaceptable (débil)

COHESIÓN PROCEDURAL (un módulo contiene operaciones que se realizan en un orden concreto aunque sean independientes).

COHESIÓN LÓGICA (cuando un módulo contiene operaciones cuya ejecución depende de un parámetro: el flujo de control del módulo es lo único que une a las operaciones que lo forman).

COHESIÓN COINCIDENTAL (cuando las operaciones de un módulo no guardan ninguna relación observable entre ellas).

Hay que procurar evitar situaciones de cohesión procedural, lógica o coincidental

Acoplamiento

Medida de la interacción
de los módulos que constituyen un programa.

Niveles de acoplamiento (de mejor a peor):

ACOPLAMIENTO DE DATOS (acoplamiento normal): Todo lo que comparten dos módulos se especifica en la lista de parámetros del módulo invocado.

ACOPLAMIENTO DE CONTROL: Cuando un módulo pasa datos que le indican a otro qué hacer (el primer módulo tiene que conocer detalles internos del segundo).

ACOPLAMIENTO EXTERNO: Cuando dos módulos utilizan los mismos datos globales o dispositivos de E/S (p.ej. ficheros).

Si los datos son de sólo lectura, el acoplamiento se puede considerar aceptable. No obstante, en general, este tipo de acoplamiento no es deseable porque la conexión existente entre los módulos no es visible (de forma explícita).

ACOPLAMIENTO PATOLÓGICO: Cuando un módulo utiliza el código de otro o altera sus datos locales ("acoplamiento de contenido").

- Los lenguajes estructurados incluyen reglas para el ámbito de las variables que impiden este tipo de acoplamiento.
- Los lenguajes orientados a objetos incluyen modificadores de visibilidad para evitar este tipo de acoplamiento.

Objetivo final
Reducir al máximo el acoplamiento entre módulos
y aumentar la cohesión interna de los módulos.

Ejemplo

Clase `Hipoteca`

```
public class Hipoteca
{
  double importe;  // en euros
  double interes;  // en porcentaje (anual)
  int    tiempo;   // en años

  // Constructor
  public Hipoteca (…) …

  // getters & setters

  public double getImporte () {
    return importe;
  }

  public void setCantidad (double euros) {
    importe = euros;
  }

  public double getInteres () {
    return interes;
  }

  public void setInteres (double tipoAnual) {
    interes = tipoAnual;
  }

  public int getTiempo () {
    return tiempo;
  }

  public void setTiempo (int years)
  {
    this.tiempo = years;
  }

  public double getCuota ()
  {
    double interesMensual = interes/(12*100);
    double cuota = (cantidad*interesMensual)
      / (1.0-1.0/Math.pow(1+interesMensual,tiempo*12));

    return Math.round(cuota*100)/100.0;
  }
}
```

Convenciones get y set

Muchas clases suelen incluir métodos públicos que permiten acceder y modificar las variables de instancia desde el exterior de la clase.

Por convención, los métodos se denominan:

- `getX` para obtener la variable de instancia `X`, y

- `setX` para establecer el valor de la variable de instancia `X`.

No es obligatorio, pero resulta conveniente, especialmente si queremos desarrollar componentes reutilizables (denominados JavaBeans) y facilitar su uso posterior.

```
import javax.swing.JOptionPane;

public class CuotaHipotecaria
{
  public static void main (String args[])
  {
    double   cantidad; // en euros
    double   interes;  // en porcentaje (anual)
    int      tiempo;   // en años
    Hipoteca hipoteca; // Hipoteca

    // Entrada de datos
    ...
    hipoteca = new Hipoteca(cantidad,interes,tiempo);

    // Resultado
    JOptionPane.showMessageDialog (null,
       "Cuota mensual = "+hipoteca.getCuota()+"€");

    System.exit(0);
  }
}
```

Modularización
Relación de ejercicios

1. Diseñe una clase `Cuenta` que represente una cuenta bancaria y permita realizar operaciones como ingresar y retirar una cantidad de dinero, así como realizar una transferencia de una cuenta a otra.

    a. Represente gráficamente la clase utilizando la notación UML

    b. Defina la clase utilizando la sintaxis de Java, definiendo las variables de instancia y métodos que crea necesarios.

    c. Implemente cada uno de los métodos de la clase. Los métodos deben actualizar el estado de las variables de instancia y mostrar un mensaje en el que se indique que la operación se ha realizado con éxito.

    d. Cree un programa en Java (en una clase llamada `CuentaTest`) que cree un par de objetos de tipo `Cuenta` y realice operaciones con ellos. El programa debe comprobar que todos los métodos de la clase `Cuenta` funcionan correctamente.

2. Diseñe una clase `Factura` que represente la venta de un producto en una tienda. La clase debe incluir información relativa al producto vendido (código, descripción y precio), datos acerca del cliente que compra el producto (nombre, apellidos, dirección, DNI) y el número de unidades compradas. Los métodos de la clase han de permitir obtener el importe total de la compra (suponiendo un porcentaje de IVA constante) y generar un informe con los datos de la factura (el "ticket" correspondiente a la venta), además de poder acceder y modificar los distintos datos recogidos en la factura.

    a. Represente gráficamente en UML la clase resultante.

    b. Implemente en Java la clase tal como esté representada en el diagrama.

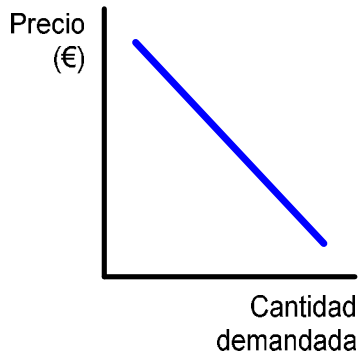
    c. Cree un programa (`FacturaTest`) que compruebe el correcto funcionamiento de la implementación realizada.

    d. Idee la forma de descomponer la clase `Factura` en varias clases de forma que la implementación resultante sea más cohesiva y las clases estén débilmente acopladas. Represente su diseño en UML e impleméntelo en Java teniendo en cuenta las relaciones existentes entre las distintas clases.

        PISTA:
        La factura mezcla varios datos de productos con datos relativos a clientes

3. CASO PRÁCTICO: *Los precios de los teléfonos móviles*
   Una empresa de telecomunicaciones nos ha encargado estudiar cuál sería la estrategia más adecuada para fijar los precios de los nuevos teléfonos móviles UMTS:

   a. En primer lugar, estudiamos cuál será la demanda de los nuevos productos, para lo cual creamos un gráfico como el siguiente:



   La gráfica muestra cómo la demanda varía en función del precio al que se venda cada terminal. Cuanto más alto sea el precio, menor será el número de personas dispuestas a pagarlo. Cuanto más bajo sea el precio, mayor será el número de personas que lo compren, aunque la empresa ingresará menos dinero por cada teléfono móvil.

   Para simplificar, suponemos que la curva de la demanda es una línea recta y creamos una clase `Demanda` que nos permitirá representar la demanda de un producto bajo diferentes circunstancias.

   Dicha clase ha de incluir métodos que nos digan cuál será la cantidad demandada a un precio determinado y qué precio hemos de fijar para conseguir vender una cantidad determinada de productos (esto es, a qué precio podemos ofrecer el producto para asegurarnos una cantidad demandada).

   NOTA: Como siempre, una vez que tengamos la clase, crearemos otra clase auxiliar que nos permita comprobar su correcto funcionamiento.

   b. A continuación, pasamos a analizar el coste que supone para nosotros producir teléfonos móviles UMTS. El coste vendrá dado por una inversión fija (en la planta que hemos de construir para fabricar los móviles) más un coste marginal por unidad (que tenderá a cero cuantos más móviles fabriquemos. El coste total vendrá dado por:

$$coste_{total} = coste_{inicial} + unidades * coste_{marginal}$$

   Decidimos crear otra clase, `Costes`, para representar el coste de producción de un producto. Esta clase incluirá un método que nos dirá cuánto nos cuesta fabricar un número determinado de unidades.

c. Finalmente, tenemos que calcular cuáles serán los ingresos que obtendremos al vender teléfonos móviles:

$$ingresos = precio * unidades$$

Decidimos crear otra clase, `Ingresos`, para representar el dinero que obtendremos al vender teléfonos móviles. Los ingresos, obviamente, dependen de la demanda y del precio que decidamos establecer. La clase deberá ofrecer un método que nos dé los ingresos totales obtenidos a un precio determinado.

d. Ahora se nos plantea el problema de ver cuál es el precio más ventajoso para la empresa en función de la demanda y de los costes que ha de afrontar. Este precio "ideal" lo podemos calcular de distintas formas:

- A partir de los ingresos y gastos totales, buscamos cuál es el valor tal que la diferencia *ingresos-costes* es máxima.

- Definimos el **ingreso marginal** como los ingresos adicionales que nos supone vender una nueva unidad de nuestro producto (bajando el precio de venta). Esto es, el ingreso marginal vendrá definido por la función:

  $$ingreso_{marginal}(x) = ingreso_{total}(x) - ingreso_{total}(x-1)$$

  En este caso, el precio ideal será aquel para el que el ingreso marginal obtenido por la venta de una unidad sea igual al coste marginal de producir esa unidad. Si fuese mayor, podríamos vender más unidades ganando más dinero. Si fuese menor, estaríamos perdiendo beneficios al perder esa unidad.

- Si la curva de la demanda es recta, el ingreso marginal puede calcularse fácilmente si tenemos en cuenta la siguiente relación:

  $$ingreso_{marginal}(x) = demanda(2x)$$

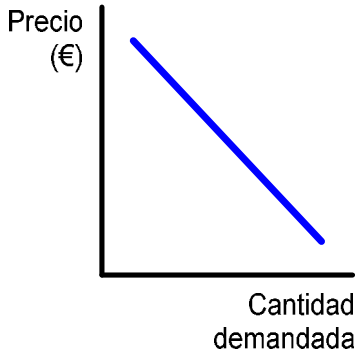
  En este caso, el precio ideal seguirá siendo aquél para el que el ingreso marginal iguale al coste marginal, es decir aquél que hace que $coste_{marginal}(x)=demanda(2x)$

Implemente las distintas estrategias en Java y compruebe que todas obtienen el mismo resultado si utilizamos los mismos datos de entrada.

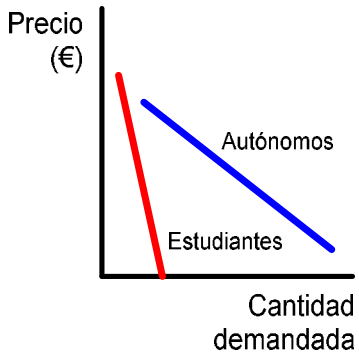
> NOTA: En vez de crear tres programas distintos, cree un único programa que acceda a una clase encargada de calcular el precio ideal al que hay que vender el producto para una demanda concreta. A continuación, modifique la implementación de la clase sin alterar la implementación del programa principal (desde donde se leen los datos de entrada y se muestran los resultados).

e. Una vez que hemos creado la infraestructura necesaria para analizar el comportamiento del mercado, podemos estudiar lo que sucede cuando modificamos nuestra política de precios.

- **Todo al mismo precio:** Si vendemos nuestro producto siempre al mismo precio, los ingresos que obtendremos serán, simplemente, el resultado de multiplicar *precio*unidades*. Tanto el número de unidades que vendemos como el precio vendrán establecidos por la curva de la demanda de teléfonos móviles.



- **Precios diferentes:** Podemos descomponer la demanda en función del tipo de clientes al que nos dirigimos, de tal forma que obtenemos dos curvas de demanda, una para profesionales autónomos y otra para estudiantes:



$$demanda_{total} = demanda_{autónomos} + demanda_{estudiantes}$$

Si suponemos que la empresa de telecomunicaciones tiene que invertir 100M€ en poner en marcha la red UMTS y el coste marginal de un teléfono móvil supone sólo 5€, calcule cuál es el precio al que habría que vender los móviles si la demanda fuese:

$$precio(cantidad) = 300 - c/40000$$

Podemos descomponer la demanda en dos segmentos:

$$precio_{estudiante}(cantidad) = 100 - cantidad/50000$$

$$precio_{autónomo}(cantidad) = 200 - cantidad/200000$$

Ahora, podemos analizar cuál sería el precio ideal al que tendríamos que venderle un teléfono a un estudiante y cuál sería el precio al que deberíamos ofrecerle un teléfono a un profesional autónomo (¿por qué no son iguales estos precios?).

Por tanto, disponemos de dos estrategias para establecer los precios de los nuevos teléfonos móviles:

- ¿Cómo venderá más móviles la empresa de telecomunicaciones?
- ¿Cómo ganará más dinero la empresa de telecomunicaciones?

Implemente en Java el proceso que hemos seguido para calcular las consecuencias de las distintas estrategias y poder analizar las situaciones que podrían llegar a producirse si cambiase la demanda de teléfonos móviles UMTS.

CUESTIONES PARA ANALIZAR CON MAYOR DETENIMIENTO:

¿Qué estrategias utiliza la empresa de telecomunicaciones para convencer a sus clientes de que deben pagar precios diferentes por el mismo servicio? ¿Están los autónomos subvencionando el uso de móviles por parte de los estudiantes? ¿Cómo se consigue eliminar de los autónomos la percepción de que pagan más de lo que podrían estar pagando?

¿Por qué se venden más caros los accesorios de un móvil en proporción a su coste con respecto al precio al que se vende el móvil en sí?

¿Por qué las líneas aéreas cobran menos dinero por un billete de ida y vuelta si pasamos el fin de semana en el destino?

¿Por qué se edita el mismo libro con distintas encuadernaciones y se cobra más por la edición con las tapas duras aunque el contenido del libro siga siendo el mismo?

Clases y objetos

Encapsulación

Herencia

- Redefinición de métodos y polimorfismo
- El Principio de Sustitución de Liskov
- Acerca de la sobrecarga de métodos
- Un ejemplo clásico: Figuras geométricas
- La palabra reservada `final`

Organización de las clases

- Organización física: ficheros
- Organización lógica: paquetes

Modificadores de acceso

Caso práctico: Vídeo-club

Encapsulación

– RECORDATORIO –

Clases…

Una clase es la especificación de un tipo de dato.

Una clase sirve

tanto de *módulo* (unidad de descomposición del software)

como de *tipo* (descripción de las características con las equipamos a los objetos de un conjunto).

… y objetos

Un objeto es una instancia de una clase.

Un objeto encapsula:

- **Datos** (atributos que le sirven para mantener su estado).
- **Operaciones** (métodos que definen su comportamiento).

Un objeto es una entidad autónoma con una funcionalidad concreta y bien definida.

Al programar, definimos una clase para especificar cómo se comportan y mantienen su estado los objetos de esa clase:

> Todos los objetos de una misma clase comparten sus atributos y el comportamiento que exhiben.

Una clase no es más que una especificación, por lo que para usarla hemos de instanciarla:

> Se crean tantos objetos de la clase como nos haga falta. Cada objeto proporcionará un servicio que podrá ser utilizado por otros objetos de nuestro sistema.

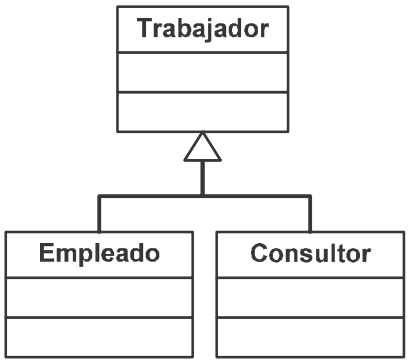
Herencia

Hay clases que comparten gran parte de sus características.

> El mecanismo conocido con el nombre de herencia permite reutilizar clases: Se crea una nueva clase que extiende la funcionalidad de una clase existente sin tener que reescribir el código asociado a esta última.

- La nueva clase, a la que se denomina **subclase**, puede poseer atributos y métodos que no existan en la clase original.

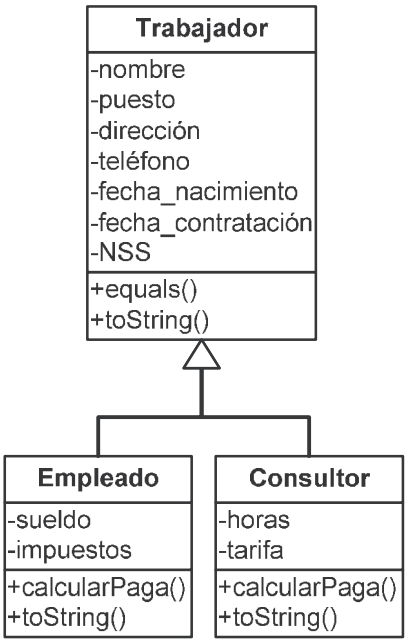
- Los objetos de la nueva clase **heredan** los atributos y los métodos **de la** clase original, que se denomina **superclase**.



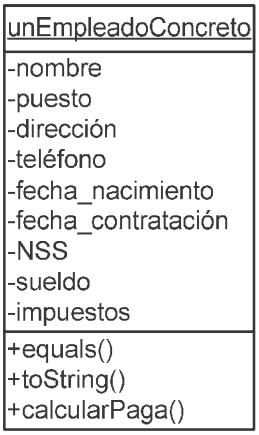
§ Trabajador es una clase genérica que sirve para almacenar datos como el nombre, la dirección, el número de teléfono o el número de la seguridad social de un trabajador.

§ Empleado es una clase especializada para representar los empleados que tienen una nómina mensual (encapsula datos como su salario anual o las retenciones del IRPF).

§ Consultor es una clase especializada para representar a aquellos trabajadores que cobran por horas (por ejemplo, registra el número de horas que ha trabajado un consultor y su tarifa horaria).



Las clases `Empleado` y `Consultor`,
además de los atributos y de las operaciones que definen,
heredan de `Trabajador` todos sus atributos y operaciones.



Un empleado concreto tendrá,
además de sus atributos y operaciones como `Empleado`,
todos los atributos correspondientes a la superclase `Trabajador`.

En Java:

```
import java.util.Date;          // Para las fechas

public class Trabajador
{
  private String nombre;
  private String puesto;
  private String direccion;
  private String telefono;
  private Date   fecha_nacimiento;
  private Date   fecha_contrato;
  private String NSS;

  // Constructor

  public Trabajador (String nombre, String NSS)
  {
    this.nombre = nombre;
    this.NSS = NSS;
  }

  // Métodos get & set
  // ...

  // Comparación de objetos

  public boolean equals (Trabajador t)
  {
    return this.NSS.equals(t.NSS);
  }

  // Conversión en una cadena de caracteres

  public String toString ()
  {
    return nombre + " (NSS "+NSS+")";
  }
}
```

NOTA: Siempre es recomendable definir los métodos `equals()` y `toString()`

```
public class Empleado extends Trabajador
{
  private double sueldo;
  private double impuestos;

  private final int PAGAS = 14;

  // Constructor

  public Empleado
        (String nombre, String NSS, double sueldo)
  {
    super(nombre,NSS);

    this.sueldo    = sueldo;
    this.impuestos = 0.3 * sueldo;
  }

  // Nómina

  public double calcularPaga ()
  {
    return (sueldo-impuestos)/PAGAS;
  }

  // toString

  public String toString ()
  {
    return "Empleado "+super.toString();
  }
}
```

- Con la palabra reservada **`extends`** indicamos que `Empleado` es una subclase de `Trabajador`.

- Con la palabra reservada **`super`** accedemos a miembros de la superclase desde la subclase.

  Generalmente, **en un constructor**, lo primero que nos encontramos es una llamada al constructor de la clase padre con `super(…)`. Si no ponemos nada, se llama al constructor por defecto de la superclase antes de ejecutar el constructor de la subclase.

```
class Consultor extends Trabajador
{
  private int    horas;
  private double tarifa;

  // Constructor

  public Consultor (String nombre, String NSS,
                    int horas, double tarifa)
  {
    super(nombre,NSS);

    this.horas  = horas;
    this.tarifa = tarifa;
  }

  // Paga por horas

  public double calcularPaga ()
  {
    return horas*tarifa;
  }

  // toString

  public String toString ()
  {
    return "Consultor "+super.toString();
  }
}
```

- La clase `Consultor` también define un método llamado `calcularPaga()`, si bien en este caso el cálculo se hace de una forma diferente por tratarse de un trabajador de un tipo distinto.

- Tanto la clase `Empleado` como la clase `Consultor` redefinen el método `toString()` que convierte un objeto en una cadena de caracteres.

  De hecho, `Trabajador` también redefine este método, que se hereda de la clase `Object`, la clase base de la que heredan todas las clases en Java.

Redefinición de métodos

Como hemos visto en el ejemplo con el método `toString()`, cada subclase hereda las operaciones de su superclase pero tiene la posibilidad de modificar localmente el comportamiento de dichas operaciones (redefiniendo métodos).

```
// Declaración de variables

Trabajador trabajador;
Empleado   empleado;
Consultor  consultor;

// Creación de objetos

trabajador = new Trabajador ("Juan", "456");

empleado = new Empleado  ("Jose", "123", 24000.0);

consultor = new Consultor ("Juan", "456", 10, 50.0);


// Salida estándar con toString()

System.out.println(trabajador);
```

```
Juan (NSS 456)
```

```
System.out.println(empleado);
```

```
Empleado Jose (NSS 123)
```

```
System.out.println(consultor);
```

```
Consultor Juan (NSS 456)
```

```
// Comparación de objetos con equals()

System.out.println(trabajador.equals(empleado));
```

```
false
```

```
System.out.println(trabajador.equals(consultor));
```

```
true
```

Polimorfismo

Al redefinir métodos, objetos de diferentes tipos
pueden responder de forma diferente a la misma llamada
(y podemos escribir código de forma general sin preocuparnos
del método concreto que se ejecutará en cada momento).

Ejemplo

Podemos añadirle a la clase `Trabajador` un método `calcularPaga` genérico (que no haga nada por ahora):

`public class Trabajador…`

```
public double calcularPaga ()
{
  return 0.0;                // Nada por defecto
}
```

En las subclases de `Trabajador`, no obstante, sí que definimos el método `calcularPaga()` para que calcule el importe del pago que hay que efectuarle a un trabajador (en función de su tipo).

`public class Empleado extends Trabajador…`

```
public double calcularPaga ()       // Nómina
{
  return (sueldo-impuestos)/PAGAS;
}
```

`class Consultor extends Trabajador…`

```
public double calcularPaga ()       // Por horas
{
  return horas*tarifa;
}
```

Como los consultores y los empleados son trabajadores, podemos crear un array de trabajadores con consultores y empleados:

```
...
Trabajador trabajadores[] = new Trabajador[2];

trabajadores[0] = new Empleado
                  ("Jose", "123", 24000.0);

trabajadores[1] = new Consultor
                  ("Juan", "456", 10, 50.0);
...
```

Una vez que tenemos un vector con todos los trabajadores de una empresa, podríamos crear un programa que realizase los pagos correspondientes a cada trabajador de la siguiente forma:

```
…
public void pagar (Trabajador trabajadores[])
{
  int i;

  for (i=0; i<trabajadores.length; i++)
      realizarTransferencia ( trabajadores[i],
                trabajadores[i].calcularPaga());
}
…
```

Para los trabajadores del vector anterior, se realizaría una transferencia de 1200 € para el empleado Jose y otra transferencia, esta vez de 500€, para el consultor Juan.

Cada vez que se invoca el método `calcularPaga()`, se busca automáticamente el código que en cada momento se ha de ejecutar en función del tipo de trabajador (**enlace dinámico**).

La búsqueda del método que se ha de invocar como respuesta a un mensaje dado se inicia con la clase del receptor. Si no se encuentra un método apropiado en esta clase, se busca en su clase padre (de la hereda la clase del receptor). Y así sucesivamente hasta encontrar la implementación adecuada del método que se ha de ejecutar como respuesta a la invocación original.

El Principio de Sustitución de Liskov

"Debe ser posible utilizar cualquier objeto instancia de una subclase en el lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado."

Barbara H. Liskov & Stephen N. Zilles:
"Programming with Abstract Data Types"
Computation Structures Group, Memo No 99, MIT, Project MAC, 1974.
(ACM SIGPLAN Notices, 9, 4, pp. 50-59, April 1974.)

El cumplimiento del Principio de Sustitución de Liskov permite obtener un comportamiento y diseño coherente:

Ejemplo

Cuando tengamos trabajadores,
sean del tipo particular que sean,
el método `calcularPaga()` siempre calculará
el importe del pago que hay que efectuar
en compensación por los servicios del trabajador.

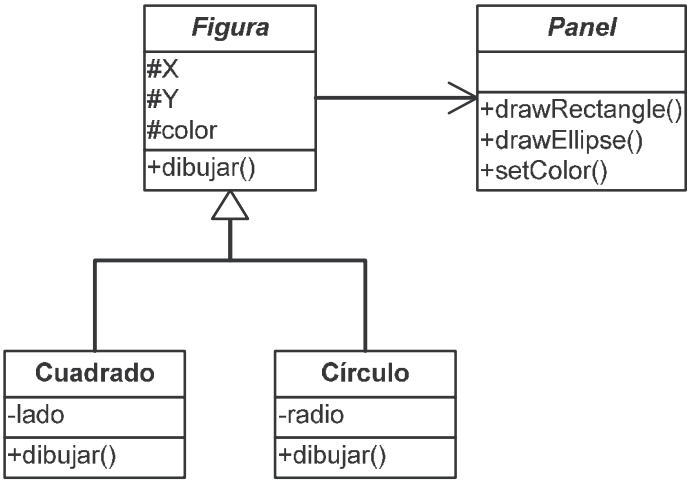
Acerca de la sobrecarga de métodos

No hay que confundir el polimorfismo con la sobrecarga de métodos (distintos métodos con el mismo nombre pero diferentes parámetros).

Ejemplo

Podemos definir varios constructores
para crear de distintas formas objetos de una misma clase.

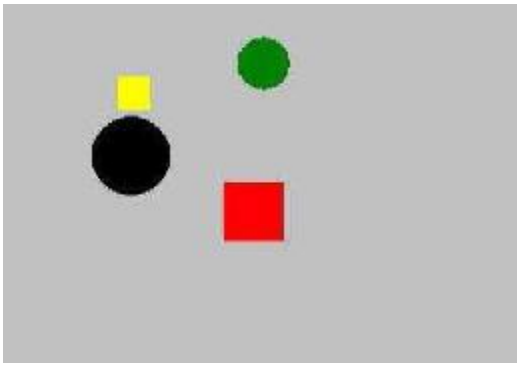
Un ejemplo clásico: Figuras geométricas

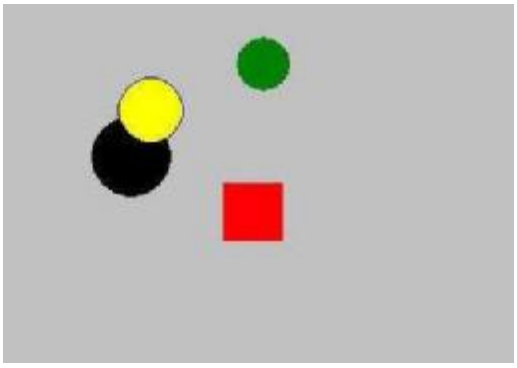



```
public class Figura
{
  protected double x;
  protected double y;
  protected Color  color;
  protected Panel  panel;


  public Figura (Panel panel, double x, double y)
  {
    this.panel = panel;
    this.x     = x;
    this.y     = y;
  }

  public void setColor (Color color)
  {
    this.color = color;
    panel.setColor(color);
  }

  public void dibujar ()
  {
    // No hace nada aquí…
  }
}
```

```
public class Circulo extends Figura
{
  private double radio;

  public Circulo(Panel panel,
                 double x, double y, double radio)
  {
    super(panel,x,y);
    this.radio = radio;
  }

  public void dibujar ()
  {
    panel.drawEllipse(x,y, x+2*radio, y+2*radio);
  }
}
```

```
public class Cuadrado extends Figura
{
  private double lado;

  public Cuadrado(Panel panel,
                  double x, double y, double lado)
  {
    super(panel,x,y);
    this.lado = lado;
  }

  public void dibujar ()
  {
    panel.drawRectangle(x,y, x+lado, y+lado);
  }
}
```





La palabra reservada `final`

En Java, usando la palabra reservada `final`, podemos:

1. Evitar que un **método** se pueda redefinir en una subclase:

```
class Consultor extends Trabajador
{
  …
  public final double calcularPaga ()
  {
    return horas*tarifa;
  }
  …
}
```

Aunque creemos subclases de `Consultor`, el dinero que se le pague siempre será en función de las horas que trabaje y de su tarifa horaria (y eso no podremos cambiarlo aunque queramos).

2. Evitar que se puedan crear subclases de una **clase** dada:

```
public final class Circulo extends Figura
…

public final class Cuadrado extends Figura
…
```

Al usar `final`, tanto `Circulo` como `Cuadrado` son ahora clases de las que no se pueden crear subclases.

En ocasiones, una clase será "final"…
porque no tenga sentido crear subclases o, simplemente, porque deseamos que la clase no se pueda extender.

RECORDATORIO:
En Java, `final` también se usa para definir constantes simbólicas.

Organización de las clases

Organización física: Ficheros

En Java, el código correspondiente a cualquier clase pública ha de estar definida en un fichero independiente con extensión `.java`.

El nombre del fichero ha de coincidir con el nombre de la clase.

En ocasiones, en un fichero se pueden incluir varias clases si sólo una de ellas es pública (esto es, las demás son únicamente clases auxiliares que utilizamos para implementar la funcionalidad correspondiente a la clase pública).

> *Ejemplo*
> Las clases que se utilizan para implementar manejadores de eventos en aplicaciones con interfaces gráficas de usuario.

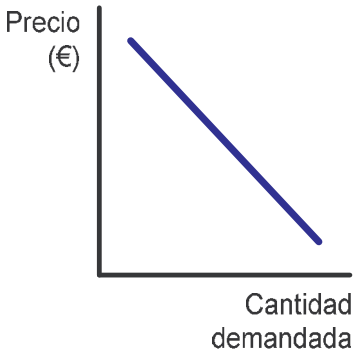
Una vez compilada, una clase, sea pública o no, da lugar a un fichero con extensión `.class` en el que se almacenan los bytecodes correspondientes al código de la clase.

Cuando ejecutemos una aplicación que utilice una clase particular, el fichero `.class` correspondiente a la clase debe ser accesible a partir del valor que tenga en ese momento la variable de entorno CLASSPATH:

> El fichero `.class` debe encontrarse en una de las carpetas/directorios incluidos en el CLASSPATH.
>
> Java también permite incluir ficheros comprimidos en el CLASSPATH, en formato ZIP, con extensión `.zip` o `.jar`, por lo que el fichero `.class` puede encontrarse dentro de uno de los ficheros de este tipo incluidos en el CLASSPATH.

Organización lógica: Paquetes

- Las clases en Java se agrupan en paquetes.
- Todas las clases compiladas en el mismo directorio (carpeta) se consideran pertenecientes a un mismo paquete.
- El paquete al que pertenece una clase se indica al comienzo del fichero en el que se define la clase con la palabra reservada `package`.
- El nombre del paquete ha de cumplir las mismas normas que cualquier otro identificador en Java.




```
package economics;

public class Demanda
{
  private double pendiente;
  private double precioMaximo;


  public Demanda (double precioMax, double pendiente)
  {
    this.pendiente = pendiente;
    this.precioMaximo = precioMaximo;
  }


  public double getPrecio (long cantidad)
  {
    return precioMaximo + cantidad*pendiente;
  }

  public long getCantidadDemandada (double precio)
  {
    return (long) ((precio-precioMaximo)/pendiente);
  }
}
```

- Cuando una clase pertenece a un paquete (p.ej. `economics`), el fichero `.java` ha de colocarse en un subdirectorio del directorio que aparezca en el CLASSPATH (p.ej. `personal/economics` si `personal` es lo que aparece en el CLASSPATH).

- Los paquetes se pueden organizar de forma jerárquica, de forma que `economics.markets` será un subpaquete del paquete `economics`

  Los ficheros `.java/.class` correspondientes deberán colocarse en el directorio `personal/economics/markets`

- Cuando usamos una clase que no está en el mismo paquete en el que nos encontramos, hemos de incluir una sentencia `import` al comienzo del fichero `.java` o utilizar el nombre completo de la clase a la que hacemos referencia en el código (esto es, `paquete.Clase`).

```
package economics.markets;

public class AnalisisEconomico
{
  private economics.Demanda  demanda;
  private economics.Costes   costes;
  private economics.Ingresos ingresos;
  …
}
```

o bien…

```
package economics.markets;

import economics.*;

public class AnalisisEconomico
{
  private Demanda  demanda;
  private Costes   costes;
  private Ingresos ingresos;
  …
}
```

Ejemplo

La biblioteca de clases estándar de Java
incluye cientos de clases organizadas en multitud de paquetes:

Paquete	Descripción
`java.lang`	Clases centrales de la plataforma Java (números, cadenas y objetos). No es necesario incluir la sentencia `import` cuando se usan clases de este paquete.
`java.awt`	Clases para crear interfaces gráficas y dibujar figuras e imágenes.
`java.applet`	Clases necesarias para crear applets.
`java.io`	Clases para realizar operaciones de entrada/salida (p.ej. uso de ficheros).
`java.util`	Utilidades varias: fechas, generadores de números aleatorios, vectores de tamaño dinámico, etcétera.
`java.net`	Para implementar aplicaciones distribuidas (que funcionen en redes de ordenadores).
`java.rmi`	Para crear aplicaciones distribuidas con mayor comodidad [*Remote Method Invocation*].
`java.sql`	Clases necesarias para acceder a bases de datos.
`javax.swing`	Para crear interfaces gráficas de usuario con componentes 100% escritos en Java.

Modificadores de acceso

Se pueden establecer distintos niveles de encapsulación para los miembros de una clase (atributos y operaciones) en función de desde dónde queremos que se pueda acceder a ellos:

Visibilidad	Significado	Java	UML
Pública	Se puede acceder al miembro de la clase desde cualquier lugar.	`public`	+
Protegida	Sólo se puede acceder al miembro de la clase desde la propia clase o desde una clase que herede de ella.	`protected`	#
Por defecto	**Se puede acceder a los miembros de una clase desde cualquier clase en el mismo paquete**		~
Privada	Sólo se puede acceder al miembro de la clase desde la propia clase.	`private`	-

La encapsulación
permite agrupar datos y operaciones en un objeto,
de tal forma los detalles del objeto se ocultan a sus usuarios
(ocultamiento de información):

A un objeto se accede a través de sus métodos públicos (su **interfaz**), por lo que no es necesario conocer su **implementación**.

Para encapsular el estado de un objeto,
sus atributos se declaran como variables de instancia privadas.

```
package economics;

public class Costes
{
  private double costeInicial;
  private double costeMarginal;
  …
}
```

Como consecuencia, se han de emplear métodos `get` para permitir que se pueda acceder al estado de un objeto:

```
public class Costes…
  public double getCosteInicial ()
  {
    return costeInicial;
  }
  public double getCosteMarginal ()
  {
    return costeMarginal;
  }
```

Si queremos permitir que se pueda modificar el estado de un objeto desde el exterior, implementaremos métodos `set`:

```
public class Costes…
  public void setCosteInicial (double inicial)
  {
    this.costeInicial = inicial;
  }
  public void setCosteMarginal (double marginal)
  {
    this.costeMarginal = marginal;
  }
```

OBSERVACIONES FINALES:

- Que los miembros de una clase sean privados quiere decir que no se puede acceder a ellos desde el exterior de la clase (ni siquiera desde sus propias subclases), lo que permite mantener la encapsulación de los objetos.

- La visibilidad protegida relaja esta restricción ya que permite acceder a los miembros de una clase desde sus subclases.

  No obstante, su uso tiende a crear jerarquías de clases fuertemente acopladas, algo que procuraremos evitar.

```
public class Figura
{
  protected double x;
  protected double y;
  protected Color  color;
  protected Panel  panel;
  …
}

public class Cuadrado extends Figura
{
  …
  // Desde cualquier sitio de la implementación
  // de la clase Cuadrado se puede acceder a los
  // miembros protegidos de la clase Figura
  …
}
```

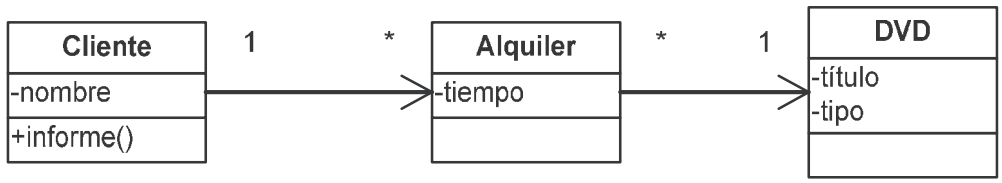
p.ej. Si tuviésemos que localizar un error que afectase al color de la figura no bastaría con examinar el código de la clase `Figura`. También tendríamos que analizar el uso que se hace del atributo color en todas las subclases de `Figura`.

Caso práctico
Alquiler de películas en un vídeo-club

Adaptado de *"Refactoring"* © Martin Fowler, 2000

Supongamos que tenemos que desarrollar una aplicación que gestione los alquileres de DVDs en un vídeo-club.

Inicialmente, nuestro diagrama de clases sería similar al siguiente:




```
public class DVD
{
  // Constantes simbólicas

  public static final int INFANTIL = 2;
  public static final int NORMAL   = 0;
  public static final int NOVEDAD  = 1;

  // Variables de instancia

  private String _titulo;
  private int    _tipo;

  // Constructor

  public DVD (String titulo, int tipo)
  {
    _titulo = titulo;
    _tipo   = tipo;
  }
```

```
  // Acceso a las variables de instancia

  public int getTipo()
  {
    return _tipo;
  }

  public void setTipo (int tipo)
  {
    _tipo = tipo;
  }

  public String getTitulo ()
  {
    return _titulo;
  }
}
```

```
public class Alquiler
{
  private DVD _dvd;
  private int _tiempo;

  public Alquiler (DVD dvd, int tiempo)
  {
    _dvd = dvd;
    _tiempo = tiempo;
  }

  public int getTiempo()
  {
    return _tiempo;
  }

  public DVD getDVD()
  {
    return _dvd;
  }
}
```

```
import java.util.Vector;


public class Cliente
{
  // Variables de instancia

  private String _nombre;
  private Vector _alquileres =new Vector();

  // Constructor

  public Cliente (String nombre)
  {
    _nombre = nombre;
  }

  // Acceso a las variables de instancia

  public String getNombre()
  {
    return _nombre;
  }

  // Registrar alquiler

  public void nuevoAlquiler (Alquiler alquiler)
  {
    _alquileres.add(alquiler);
  }

  // Emitir un informe del cliente

  public String informe()
  {
    double      total;
    double      importe;
    int         puntos;
    int         i;
    Alquiler    alquiler;
    String      salida;

    total  = 0;
    puntos = 0;
    salida = "Informe para " + getNombre() + "\n";
```

```
// Recorrido del vector de alquileres

for (i=0; i<_alquileres.size(); i++) {

    importe  = 0;
    alquiler = (Alquiler) _alquileres.get(i);

    // Importe del alquiler

    switch (alquiler.getDVD().getTipo()) {

      case DVD.NORMAL:
           importe += 2;
           if (alquiler.getTiempo()>2)
              importe += (alquiler.getTiempo()-2)*1.5;
           break;

      case DVD.NOVEDAD:
           importe += alquiler.getTiempo() * 3;
           break;

      case DVD.INFANTIL:
           importe += 1.5;
           if (alquiler.getTiempo()>3)
              importe += (alquiler.getTiempo()-3)*1.5;
           break;
    }

    // Programa de puntos

    puntos++;

    if (  (alquiler.getDVD().getTipo()==DVD.NOVEDAD)
       && (alquiler.getTiempo()>1) )
       puntos++;                        // Bonificación

    // Mostrar detalles del alquiler

    salida += "\t" + alquiler.getDVD().getTitulo()
            + "\t" + String.valueOf(importe) + " €\n";

    // Acumular total

    total += importe;
}
```

```
    // Pie del informe
    salida += "IMPORTE TOTAL = "
            + String.valueOf(total) + " €\n";
    salida += "Dispone de "
            + String.valueOf(puntos) + " puntos\n";
    return salida;
  }
}
```

Paso 1: Extraer método de **`informe()`**

El método `informe` es excesivamente largo…

```
public class Cliente…

  public double precio (Alquiler alquiler)
  {
    double importe = 0;
    switch (alquiler.getDVD().getTipo()) {
      case DVD.NORMAL:
           importe += 2;
           if (alquiler.getTiempo()>2)
              importe += (alquiler.getTiempo()-2)*1.5;
           break;
      case DVD.NOVEDAD:
           importe += alquiler.getTiempo() * 3;
           break;
      case DVD.INFANTIL:
           importe += 1.5;
           if (alquiler.getTiempo()>3)
              importe += (alquiler.getTiempo()-3)*1.5;
           break;
    }
    return importe;
  }
```

```
public String informe()
{
  double      total;
  double      importe;
  int         puntos;
  int         i;
  Alquiler    alquiler;
  String      salida;

  total  = 0;
  puntos = 0;
  salida = "Informe para " + getNombre() +  "\n";

  for (i=0; i<_alquileres.size(); i++) {

      alquiler = (Alquiler) _alquileres.get(i);
      importe  = precio(alquiler);

      // Programa de puntos

      puntos++;

      if ((alquiler.getDVD().getTipo()==DVD.NOVEDAD)
         && (alquiler.getTiempo()>1))
         puntos++;                    // Bonificación

      // Mostrar detalles del alquiler

      salida += "\t" + alquiler.getDVD().getTitulo()
              + "\t" + String.valueOf(importe) + "€\n";

      // Acumular total

      total += importe;
  }

  // Pie del recibo

  salida += "IMPORTE TOTAL = "
          + String.valueOf(total) + " €\n";

  salida += "Dispone de "
          + String.valueOf(puntos) + " puntos\n";

  return salida;
}
```

Debemos comprobar que calculamos correctamente los precios, para lo que preparamos una batería de casos de prueba:

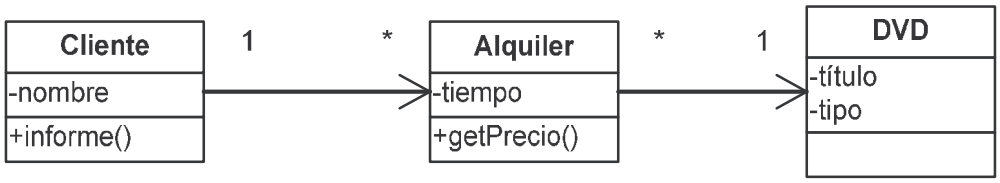
```
import junit.framework.*;

public class AlquilerTest extends TestCase
{
  private Cliente  cliente;
  private DVD      casablanca;
  private DVD      indy;
  private DVD      shrek;
  private Alquiler alquiler;
                                                  // Infraestructura
  public static void main(String args[])
  {
    junit.swingui.TestRunner.main (
          new String[] {"AlquilerTest"});
  }

  public AlquilerTest(String name)
  {
    super(name);
  }

  public void setUp ()
  {
    cliente = new Cliente("Kane");
    casablanca = new DVD("Casablanca", DVD.NORMAL);
    indy = new DVD("Indiana Jones XIII", DVD.NOVEDAD);
    shrek = new DVD("Shrek", DVD.INFANTIL);
  }
                                                  // Casos de prueba
  public void testNormal1 ()
  {
    alquiler = new Alquiler(casablanca,1);
    assertEquals( cliente.precio(alquiler), 2.0, 0.001);
  }

  public void testNormal2 ()
  {
    alquiler = new Alquiler(casablanca,2);
    assertEquals( cliente.precio(alquiler), 2.0, 0.001);
  }

  public void testNormal3 ()
  {
    alquiler = new Alquiler(casablanca,3);
    assertEquals( cliente.precio(alquiler), 3.5, 0.001);
  }
```

```
  public void testNormal7 ()
  {
    alquiler = new Alquiler(casablanca,7);
    assertEquals( cliente.precio(alquiler), 9.5, 0.001);
  }
  public void testNovedad1 ()
  {
    alquiler = new Alquiler(indy,1);
    assertEquals( cliente.precio(alquiler), 3.0, 0.001);
  }
  public void testNovedad2 ()
  {
    alquiler = new Alquiler(indy,2);
    assertEquals( cliente.precio(alquiler), 6.0, 0.001);
  }
  public void testNovedad3 ()
  {
    alquiler = new Alquiler(indy,3);
    assertEquals( cliente.precio(alquiler), 9.0, 0.001);
  }
  public void testInfantil1 ()
  {
    alquiler = new Alquiler(shrek,1);
    assertEquals( cliente.precio(alquiler), 1.5, 0.001);
  }
  public void testInfantil3 ()
  {
    alquiler = new Alquiler(shrek,3);
    assertEquals( cliente.precio(alquiler), 1.5, 0.001);
  }
  public void testInfantil4 ()
  {
    alquiler = new Alquiler(shrek,4);
    assertEquals( cliente.precio(alquiler), 3.0, 0.001);
  }
  public void testInfantil7 ()
  {
    alquiler = new Alquiler(shrek,7);
    assertEquals( cliente.precio(alquiler), 7.5, 0.001);
  }
}
```

11/11

Paso 2: Mover el método `precio()`

En realidad, `precio` no usa datos de `Cliente`, por lo que resulta más que razonable convertirlo en un método de la clase `Alquiler`…




```
public class Alquiler
{
  …
  public double getPrecio ()
  {
    double importe = 0;

    switch (getDVD().getTipo()) {

      case DVD.NORMAL:
           importe += 2;
           if (getTiempo()>2)
              importe += (getTiempo()-2) * 1.5;
           break;

      case DVD.NOVEDAD:
           importe += getTiempo() * 3;
           break;

      case DVD.INFANTIL:
           importe += 1.5;
           if (getTiempo()>3)
              importe += (getTiempo()-3) * 1.5;
           break;
    }

    return importe;
  }
}
```

- Cuando un método de una clase (`Cliente`) accede continuamente a los miembros de otra clase (`Alquiler`) pero no a los de su clase (`Cliente`), es conveniente mover el método a la clase cuyos datos utiliza. Además, el código resultante será más sencillo.

Como hemos cambiado `precio()` de sitio, tenemos que cambiar las llamadas a `precio()` que había en nuestra clase `Cliente`:

```
public class Cliente
{
  …

  public String informe()
  {
    …
    for (i=0; i<_alquileres.size(); i++) {
        alquiler = (Alquiler) _alquileres.get(i);
        importe  = alquiler.getPrecio();
        …
    }
    …
  }
}
```

Además, deberemos actualizar nuestros casos de prueba:

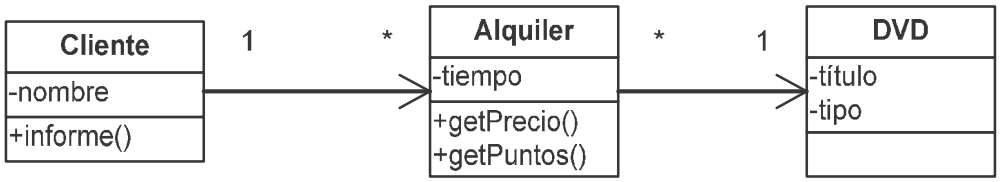
p.ej.

```
  public void testNormal3 ()
  {
    alquiler = new Alquiler(casablanca,3);
    assertEquals( alquiler.getPrecio(), 3.5, 0.001);
  }
```

Cuando hayamos realizado todos los cambios,
volveremos a ejecutar los casos de prueba
para comprobar que todo sigue funcionando correctamente.

11/11

Paso 3: Extraer el cálculo correspondiente al programa de puntos



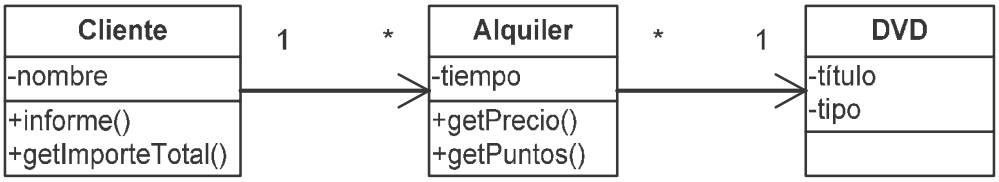

```
public class Alquiler
{
  …
  public int getPuntos ()
  {
    int puntos = 1;

    // Bonificación

    if (  (getDVD().getTipo() == DVD.NOVEDAD)
       && (getTiempo()>1))
       puntos++;

    return puntos;
  }
}
```

```
public class Cliente
{
  …
  public String informe()
  {
    …
    for (i=0; i<_alquileres.size(); i++) {
        alquiler = (Alquiler) _alquileres.get(i);
        importe  = alquiler.getPrecio();
        puntos  += alquiler.getPuntos();
        …
    }
    …
  }
}
```

Paso 4: Separar los cálculos de las operaciones de E/S

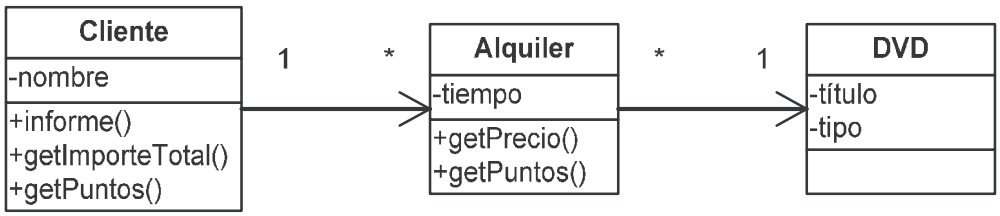
En el método `informe()`, estamos mezclando cálculos útiles con las llamadas a `System.out.println()` que generar el informe:

- Creamos un método independiente para calcular el gasto total realizado por un cliente:




```
public class Cliente…

  public double getImporteTotal ()
  {
    int      i;
    double   total;
    Alquiler alquiler;

    total = 0;

    for (i=0; i<_alquileres.size(); i++) {
        alquiler = (Alquiler) _alquileres.get(i);
        total   += alquiler.getPrecio();
    }

    return total;
  }
```

Creamos otro método independiente para calcular los puntos acumulados por un cliente en el programa de puntos del vídeo-club:




```
public class Cliente…

  public int getPuntos()
  {
    int      i;
    int      puntos;
    Alquiler alquiler;

    puntos = 0;

    for (i=0; i<_alquileres.size(); i++) {
        alquiler = (Alquiler) _alquileres.get(i);
        puntos  += alquiler.getPuntos();
    }

    return puntos;
  }
```

Al separar los cálculos de las operaciones de E/S, podemos preparar casos de prueba que comprueben el funcionamiento correcto del cálculo del total y del programa de puntos del vídeo-club.

- EJERCICIO -

Tras los cambios anteriores en la clase `Cliente`, la generación del informe es bastante más sencilla que antes:

```
public class Cliente…

  public String informe()
  {
    int      i;
    Alquiler alquiler;
    String   salida;

    salida = "Informe para " + getNombre() + "\n";

    for (i=0; i<_alquileres.size(); i++) {

        alquiler = (Alquiler) _alquileres.get(i);

        salida += "\t"
                + alquiler.getDVD().getTitulo()
                + "\t"
                + String.valueOf(alquiler.getPrecio())
                + " €\n";
    }

    salida += "IMPORTE TOTAL = "
            + String.valueOf(getImporteTotal())
            + " €\n";

    salida += "Dispone de "
            + String.valueOf(getPuntos())
            + " puntos\n";

    return salida;
  }
```

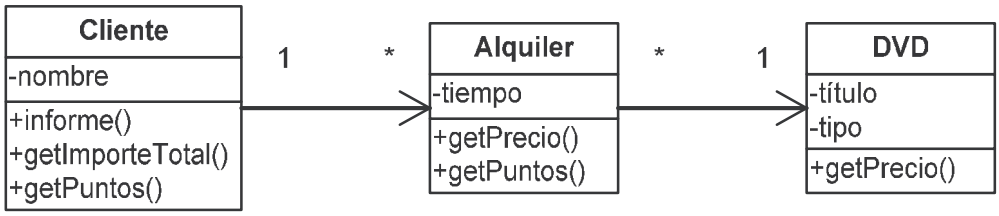
Paso 5: Nueva funcionalidad – Informes en HTML

Una vez que nuestro método `informe()` se encarga únicamente de realizar las tareas necesarias para generar el informe en sí, resulta casi trivial añadirle a nuestra aplicación la posibilidad de generar los informes en HTML (el formato utilizado para crear páginas web):

```
public class Cliente…

  public String informeHTML()
  {
    int         i;
    Alquiler    alquiler;
    String      salida;

    salida = "<H1>Informe para "
           + "<I>" + getNombre() + "</I>"
           + "</H1>\n";

    salida += "<UL>";

    for (i=0; i<_alquileres.size(); i++) {

        alquiler = (Alquiler) _alquileres.get(i);

        salida += "<LI>" + alquiler.getDVD().getTitulo()
                + "(" + alquiler.getPrecio() + " €)\n";
    }

    salida += "</UL>";

    salida += "<P>IMPORTE TOTAL = "
            + "<B>" + getImporteTotal() + " €</B>\n";


    salida += "<P>Dispone de "
            + "<I>" + getPuntos() + " puntos</I>\n";

    return salida;
  }
```

Paso 6: Mover el método `getPrecio()`

Movemos la implementación del método `getPrecio()` de la clase `Alquiler` al lugar que parece más natural si los precios van ligados a la película que se alquila:



public class DVD…

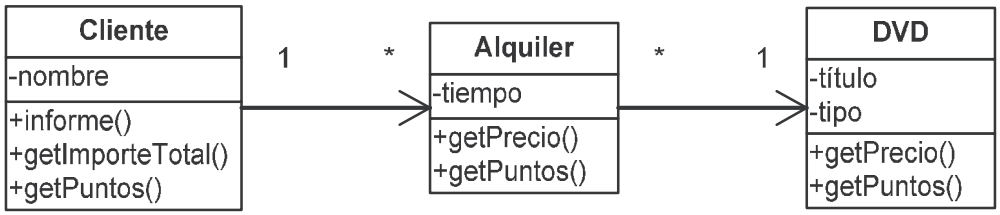
```
  public double getPrecio (int tiempo)
  {
    double importe = 0;

    switch (getTipo()) {

      case DVD.NORMAL:
           importe += 2;
           if (tiempo>2)
              importe += (tiempo-2) * 1.5;
           break;

      case DVD.NOVEDAD:
           importe += tiempo * 3;
           break;

      case DVD.INFANTIL:
           importe += 1.5;
           if (tiempo>3)
              importe += (tiempo-3) * 1.5;
           break;
    }

    return importe;
  }
```

public class Alquiler…

```
  public double getPrecio()
  {
    return _dvd.getPrecio(_tiempo);
  }
```

Paso 7: Mover el método `getPuntos ()`

Hacemos lo mismo con el método `getPuntos()`:




```
public class DVD…

  public int getPuntos (int tiempo)
  {
    int puntos = 1;

    // Bonificación

    if ((getTipo() == DVD.NOVEDAD) && (tiempo>1))
       puntos++;

    return puntos;
  }
```

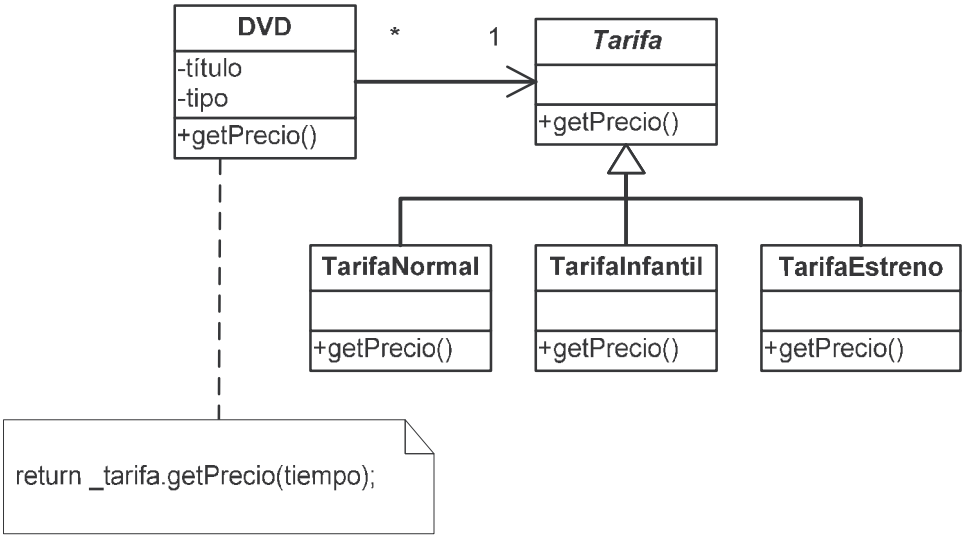
```
public class Alquiler…

  public int getPuntos ()
  {
    return _dvd.getPuntos(_tiempo);
  }
```

11/11

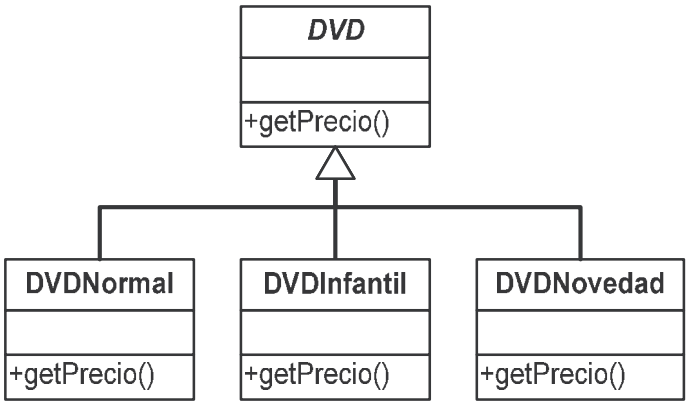
Como siempre, la ejecución de los casos de prueba nos confirma que todo funciona correctamente.

Paso 8: Reemplazar lógica condicional con polimorfismo

Queremos darle más flexibilidad a la forma en la que se fijan los precios de los alquileres de películas, para que se puedan añadir nuevas categorías (por ejemplo, en la creación de promociones) o para que se puedan ajustar las tarifas:



NOTA:
¿Por qué no creamos una jerarquía de tipos de películas?



Porque las películas pueden cambiar de categoría con el tiempo (dejan de ser una novedad) pero siguen manteniendo su identidad.

Nos creamos la jerarquía correspondiente a las políticas de precios:

```
public abstract class Tarifa
{
  public abstract int getTipo();
}
```

```
class TarifaNormal extends Tarifa
{
  public int getTipo()
  {
    return DVD.NORMAL;
  }
}
```

```
class TarifaInfantil extends Tarifa
{
  public int getTipo()
  {
    return DVD.INFANTIL;
  }
}
```

```
class TarifaEstreno extends Tarifa
{
  public int getTipo()
  {
    return DVD.NOVEDAD;
  }
}
```

A continuación, en la clase `DVD`, sustituimos el atributo `tipo` por un objeto de tipo `Tarifa`, en el cual recaerá luego la responsabilidad de establecer el precio correcto:

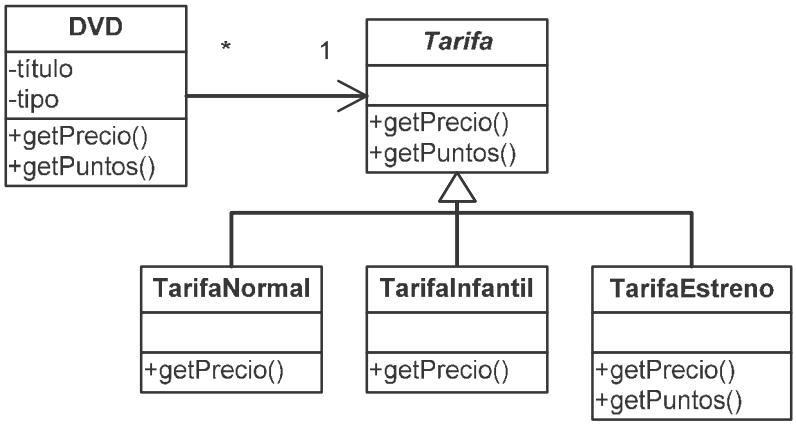
```
public class DVD…

  private String _titulo;
  private Tarifa _tarifa;


  public DVD (String titulo, int tipo)
  {
    _titulo = titulo;

    setTipo(tipo);
  }


  public int getTipo()
  {
    return _tarifa.getTipo();
  }


  public void setTipo (int tipo)
  {
    switch (tipo) {

      case DVD.NORMAL:
           _tarifa = new TarifaNormal();
           break;

      case DVD.NOVEDAD:
           _tarifa = new TarifaEstreno();
           break;

      case DVD.INFANTIL:
           _tarifa = new TarifaInfantil();
           break;
    }
  }
```

Finalmente, cambiamos la implementación de `getPrecio()`:

```
public abstract class Tarifa…
  public abstract double getPrecio (int tiempo);

class TarifaNormal extends Tarifa…
  public double getPrecio (int tiempo)
  {
    double importe = 2.0;
    if (tiempo>2)
       importe += (tiempo-2) * 1.5;
    return importe;
  }

class TarifaInfantil extends Tarifa
  public double getPrecio (int tiempo)
  {
    double importe = 1.5;
    if (tiempo>3)
       importe += (tiempo-3) * 1.5;
    return importe;
  }

class TarifaEstreno extends Tarifa
  public double getPrecio (int tiempo)
  {
    return tiempo * 3.0;
  }

public class DVD…
  public double getPrecio (int tiempo)
  {
    return _tarifa.getPrecio(tiempo);
  }
```

Paso 9: Reemplazar lógica condicional con polimorfismo II

Repetimos el mismo proceso de antes para el método `getPuntos()`:



public abstract class Tarifa…

```
  public int getPuntos (int tiempo)
  {
    return 1;
  }
```

class TarifaNovedad extends Tarifa…

```
  public int getPuntos (int tiempo)
  {
    return (tiempo>1)? 2: 1;
  }
```

public class DVD…

```
  public int getPuntos (int tiempo)
  {
    return _tarifa.getPuntos(tiempo);
  }
```

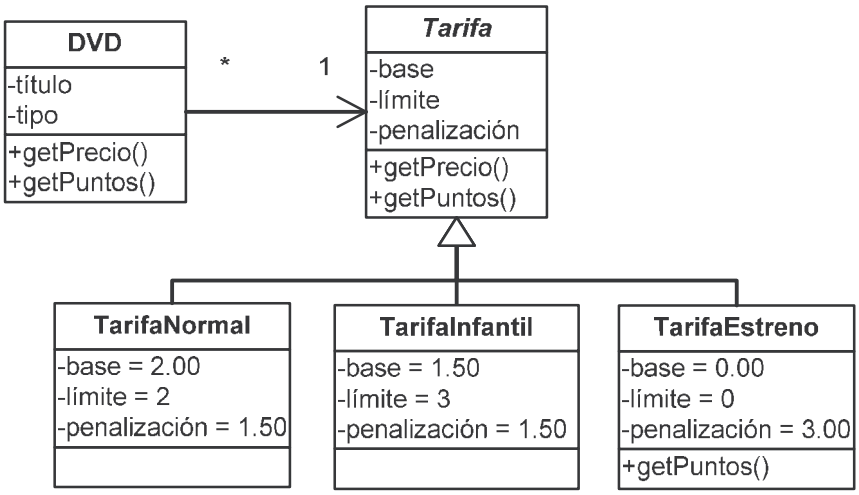
Paso 10: Mejoras adicionales

Todavía nos quedan algunas cosas que podríamos mejorar…

Las constantes simbólicas definidas en la clase `DVD` son, en realidad, meros identificadores que utilizamos para las diferenciar las distintas políticas de precios, por lo que deberíamos pasarlas a la clase genérica `Tarifa`.

Podríamos, directamente, eliminar esas constantes artificiales y forzar que, al crear un objeto de tipo `DVD`, el constructor reciba directamente como parámetro un objeto de alguno de los tipos derivados de `Tarifa` (lo que nos facilitaría el trabajo enormemente si se establecen nuevas políticas de precios).

Las distintas políticas de precios tienen características en común, por lo que podríamos reorganizar la jerarquía de clases teniendo en cuenta los rasgos comunes que se utilizan para establecer los precios (esto es, el precio base, el límite de tiempo y la penalización por tiempo).



Clases y objetos
Relación de ejercicios

1. Identifique los datos que decidiría utilizar para almacenar el estado de los siguientes objetos en función del contexto en el que se vayan a utilizar:

    a. Un punto en el espacio.

    b. Un segmento de recta.

    c. Un polígono.

    d. Una manzana (de las que se venden en un mercado).

    e. Una carta (en Correos)

    f. Un libro (en una biblioteca)

    g. Un libro (en una librería)

    h. Una canción (en una aplicación para un reproductor MP3).

    i. Una canción (en una emisora de radio)

    j. Un disco de música (en una tienda de música).

    k. Un disco de música (en una discoteca).

    l. Un teléfono móvil (en una tienda de telefonía)

    m. Un teléfono móvil (en el sistema de una empresa de telecomunicaciones)

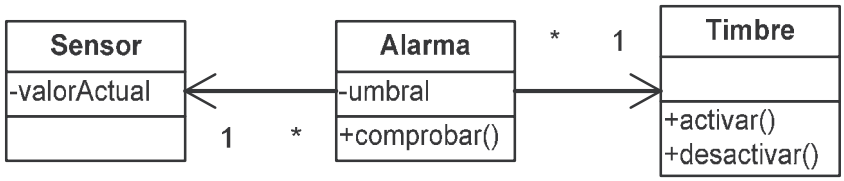
    n. Un ordenador (en una tienda de Informática)

    o. Un ordenador (en una red de ordenadores)

    p. Un ordenador (en el inventario de una organización)

    Declare las correspondientes clases en Java, defina los constructores que considere adecuados e implemente los correspondientes métodos para el acceso y la modificación del estado de los objetos (esto es, los métodos `get` y `set`).

2. Cree una clase denominada `Alarma` cuyos objetos activen un objeto de tipo `Timbre` cuando el valor medido por un `Sensor` supere un umbral preestablecido:



Implemente en Java todo el código necesario para el funcionamiento de la alarma, suponiendo que la alarma comprueba si debe activar o desactivar el timbre cuando se invoca el método `comprobar()`.

3. Cree una subclase de `Alarma` denominada `AlarmaLuminosa` que, además de activar el timbre, encienda una luz (que representaremos con un objeto de tipo `Bombilla`).

   NOTA: Procure eliminar la aparición de código duplicado al crear la subclase de `Alarma` y asegúrese de que, cuando se activa la alarma luminosa se enciende la luz de alarma y también suena la señal sonora asociada al timbre.

4. Diseñe jerarquías de clases para representar los siguientes conjuntos de objetos:

   a. Una colección de CDs, entre los cuales hay discos de música (CDs de audio), discos de música en MP3 (CD-ROMs con música), discos de aplicaciones (CD-ROMs con software) y discos de datos (CD-ROMs con datos y documentos).

   b. Los diferentes productos que se pueden encontrar en una tienda de electrónica, que tienen un conjunto de características comunes (precio, código de barras…) y una serie de características específicas de cada producto.

   c. Los objetos de una colección de monedas/billetes/sellos.

   Implemente en Java las jerarquías de clases que haya diseñado (incluyendo sus variables de instancia, sus constructores y sus métodos `get`/`set`). A continuación, escriba sendos programas que realicen las siguientes tareas:

   a. Buscar y mostrar todos los datos de un CD concreto (se recomienda definir el método toString en cada una de las subclases de CD).

   b. Crear un carrito de la compra en el que se pueden incluir productos y emitir un ticket en el que figuren los datos de cada producto del carrito, incluyendo su precio y el importe total de la compra.

   c. Un listado de todos los objetos coleccionables cuya descripción incluya una cadena de caracteres que el programa reciba como parámetro.

5. Implemente un programa que cree un objeto de la clase `Random` del paquete `java.util`, genere un número entero aleatoriamente y lo muestre en pantalla.

6. Cree un paquete denominado `documentos`…

    a. Incluya en él dos clases, `Factura` y `Pedido`, para representar facturas y pedidos, respectivamente.

    b. A continuación, ya fuera del paquete, cree un pequeño programa que cree objetos de ambos tipos y los muestre por pantalla.

    c. Añada un tercer tipo de documento, `PedidoUrgente`, que herede directamente de `Pedido`. Compruebe que el programa anterior sigue funcionando correctamente si reemplazamos un `Pedido` por un `PedidoUrgente`.

    d. Cree un nuevo tipo de documento, denominado `Contrato`, e inclúyalo en el subpaquete `documentos.RRHH`. En este último paquete, incluya también un tipo de documento `CV` para representar el currículum vitae de una persona.

    e. Si no lo ha hecho ya, cree una clase genérica `Documento` de la que hereden (directa o indirectamente) todas las demás clases que hemos definido para representar distintos tipos de documentos.

    f. Implemente un pequeño programa que cree un documento de un tipo seleccionado por el usuario. Muestre por pantalla el documento independientemente del tipo concreto de documento que se haya creado en el paso anterior.

OBSERVACIONES:

- Para cada clase que defina, determine qué miembros de la clase han de ser públicos (`public`), cuáles han de mantenerse privados (`private`) y, si lo considera oportuno, cuáles serían miembros protegidos (`protected`).

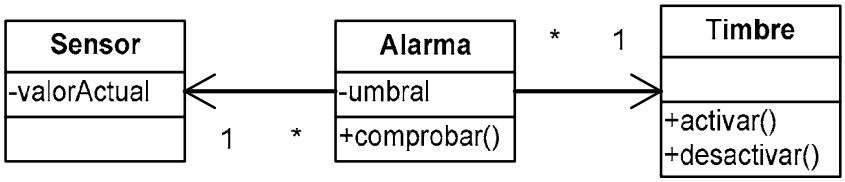
- Tenga en cuenta que no siempre se debe permitir la modificación desde el exterior de una variable de instancia (esto es, habrá variables de instancia a las que asociemos un método `get` pero no un método `set` y, de hacerlo, éste puede que sea privado o protegido).

- Analice también qué métodos de una clase deben declararse con la palabra reservada final para que no se puedan redefinir en subclases y qué clases han de ser “finales” (esto es, aquellas clases de las que no queramos permitir que se creen subclases).

- En los distintos programas de esta relación de ejercicios puede resultar necesaria la creación de colecciones de objetos de distintos tipos (p.ej. arrays de CDs, productos, objetos coleccionables o documentos).

Clases y objetos
Ejercicio resuelto

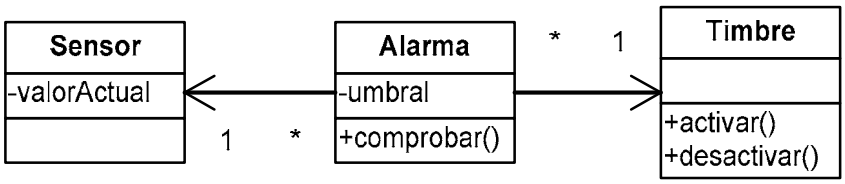
Cree una clase denominada `Alarma` cuyos objetos activen un objeto de tipo `Timbre` cuando el valor medido por un `Sensor` supere un umbral preestablecido:



Implemente en Java todo el código necesario para el funcionamiento de la alarma, suponiendo que la alarma comprueba si debe activar o desactivar el timbre cuando se invoca el método `comprobar()`.

Cree una subclase de `Alarma` denominada `AlarmaLuminosa` que, además de activar el timbre, encienda una luz (que representaremos con un objeto de tipo `Bombilla`).

> NOTA: Procure eliminar la aparición de código duplicado al crear la subclase de `Alarma` y asegúrese de que, cuando se activa la alarma luminosa, se enciende la luz de alarma y también suena la señal sonora asociada al timbre.

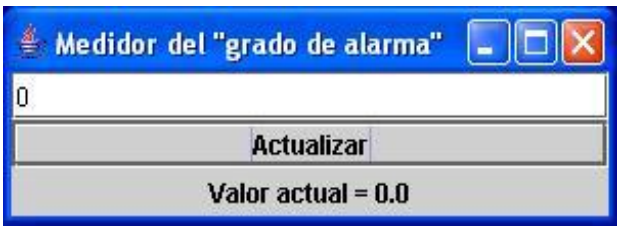


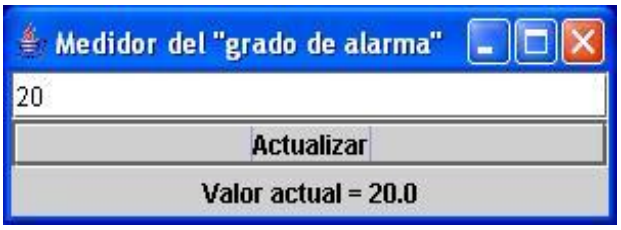

```
/**
 * Alarma
 */

public class Alarma
{

  private Sensor sensor;
  private Timbre timbre;
  private double umbral;

  /**
   * Constructor
   */

  public Alarma
        (Sensor sensor, Timbre timbre, double umbral)
  {
    this.sensor = sensor;
    this.timbre = timbre;
    this.umbral = umbral;
  }

  /**
   * Comprobar estado de la alarma
   */

  public void comprobar ()
  {
    if (sensor.getValorActual()>umbral) {
       timbre.activar();
    } else {
       timbre.desactivar();
    }
  }

}
```

Programa principal:

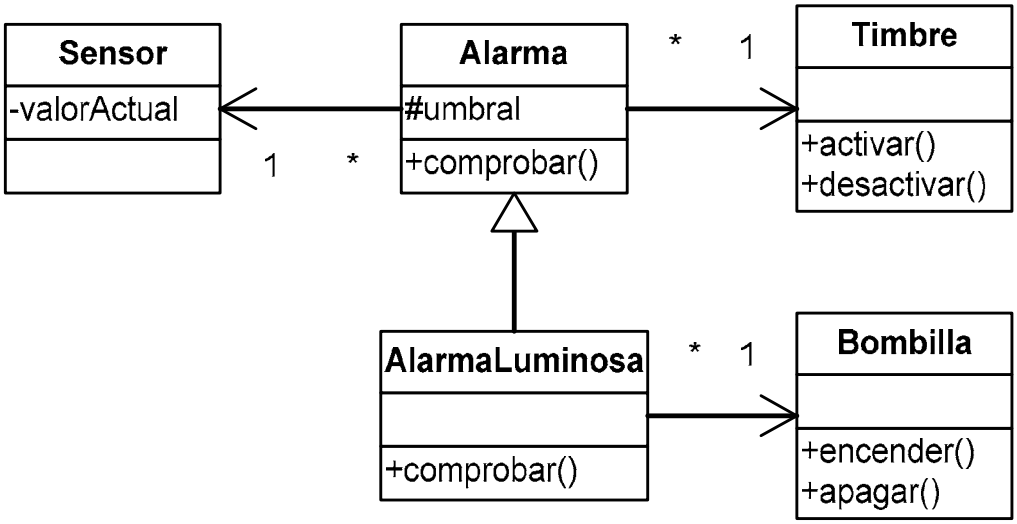
```
public class Programa
{
  // Programa principal

  public static void main(String[] args)
  {
    SensorSwing sensor = new SensorSwing();
    Timbre timbre = new Timbre();
    Alarma alarma = new Alarma (sensor, timbre, 0.0);

    sensor.setAlarma(alarma);
  }
}
```

- `SensorSwing` es un tipo particular de `Sensor` que se encarga de llamar al método `comprobar()` de la alarma en cuanto se produce un cambio en el valor medido por el sensor.





Versión 1: Redefinición del método `comprobar()`



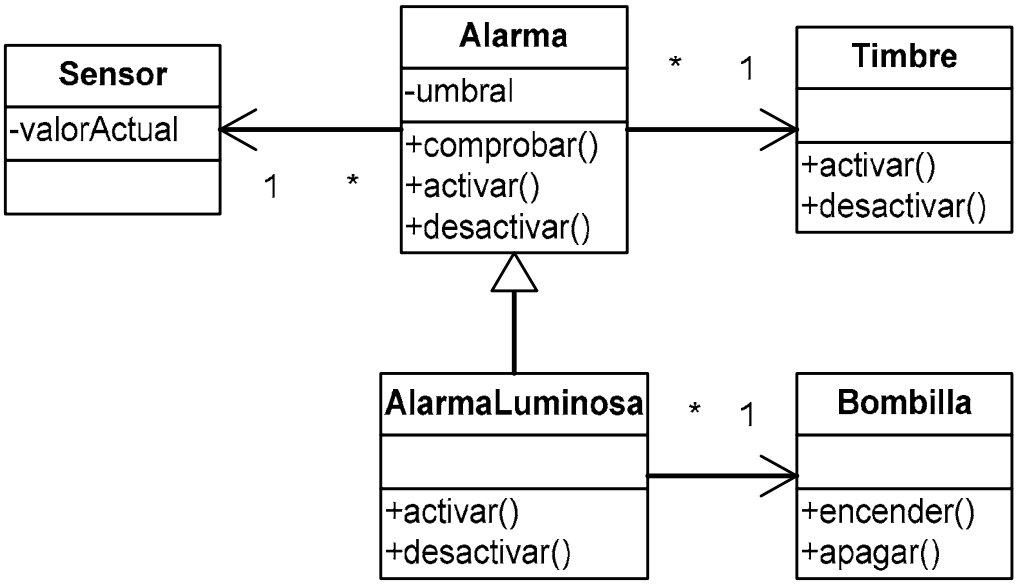

```
public class Alarma
{
  protected Sensor sensor;
  protected Timbre timbre;
  protected double umbral;

  /**
   * Constructor
   */

  public Alarma
         (Sensor sensor, Timbre timbre, double umbral)
  …

  /**
   * Comprobar estado de la alarma
   */

  public void comprobar ()
  {
    if (sensor.getValorActual()>umbral) {
       timbre.activar();
    } else {
       timbre.desactivar();
    }
  }
}
```

```
/**
 * Alarma luminosa (versión 1)
 */

public class AlarmaLuminosa extends Alarma
{
  private Bombilla bombilla;

  /**
   * Constructor
   */

  public AlarmaLuminosa
         ( Sensor sensor, Timbre timbre,
           Bombilla bombilla, double umbral)
  {
    super(sensor,timbre,umbral);

    this.bombilla = bombilla;
  }

  /**
   * Redefinición del método comprobar()
   */

  public void comprobar ()
  {
    super.comprobar();

    if (sensor.getValorActual()>umbral) {
       bombilla.encender();
    } else {
       bombilla.apagar();
    }
  }
}
```

- Acceso a miembros protegidos
  (“se rompe” la encapsulación de los objetos).
- Existencia de código duplicado (`if...`)

NOTA: Hemos de acordarnos de llamar al método `comprobar()` de la clase `Alarma` para que también suene el timbre.

Versión 2: Uso correcto de la redefinición de métodos



La redefinición de los métodos `activar()` & `desactivar()` nos permitirá:

- Reutilizar la lógica incluida en el método `comprobar()` sin tener que duplicar código (la clase `Alarma` es la única responsable de la activación de la alarma, hecho que conllevará distintas consecuencias en función del tipo particular de alarma).

- Eliminar una posible fuente de errores (anteriormente, si cambiasen las condiciones bajo las cuales se activa una `Alarma`, también tendríamos que revisar la implementación de `comprobar()` en la clase `AlarmaLuminosa`).

- Mantener la encapsulación de los objetos de la clase Alarma (no se podrá acceder a sus variables de instancia desde el exterior de la clase)

```
/**
 * Alarma
 */

public class Alarma
{
  private Sensor sensor;
  private Timbre timbre;
  private double umbral;

  /**
   * Constructor
   */
  public Alarma
         (Sensor sensor, Timbre timbre, double umbral)
  {
    this.sensor = sensor;
    this.timbre = timbre;
    this.umbral = umbral;
  }

  /**
   * Comprobar estado de la alarma
   */
  public final void comprobar ()
  {
    if (sensor.getValorActual()>umbral) {
       activar();
    } else {
       desactivar();
    }
  }

  public void activar ()
  {
    timbre.activar();
  }

  public void desactivar ()
  {
    timbre.desactivar();
  }
}
```

```
/**
 * Alarma luminosa (versión 2)
 */

public class AlarmaLuminosa extends Alarma
{
  private Bombilla bombilla;

  /**
   * Constructor
   */
  public AlarmaLuminosa
         ( Sensor sensor, Timbre timbre,
           Bombilla bombilla, double umbral)
  {
    super(sensor,timbre,umbral);
    this.bombilla = bombilla;
  }

  /**
   * Redefinición del método activar()
   */
  public void activar ()
  {
    super.activar();
    bombilla.encender();
  }

  /**
   * Redefinición del método desactivar()
   */
  public void desactivar ()
  {
    super.desactivar();
    bombilla.apagar();
  }
}
```

NOTA: Hemos de acordarnos de llamar a los métodos correspondientes de la clase `Alarma` para que el timbre siga sonando…

Apéndice: Clases auxiliares
Una posible implementación de sensores, timbres y bombillas

Sensor.java

```
/**
 * Clase genérica para representar un sensor
 */

public class Sensor
{
  /**
   * Valor actual medido por el sensor
   */

  private double valorActual;

  /**
   * Acceso al valor medido por el sensor
   */

  public double getValorActual ()
  {
    return valorActual;
  }

  /**
   * Modificación del valor del sensor
   * (algo de lo que se encargarán
   * las subclases particulares de Sensor)
   */

  protected void setValorActual (double valor)
  {
    valorActual = valor;
  }
}
```

SensorSwing.java

```
/**
 * Sensor "ficticio"
 * (demostración del uso de SWING)
 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SensorSwing extends Sensor
                         implements ActionListener
{
  private JFrame     frame;
  private JPanel     panel;
  private JButton    button;
  private JTextField editor;
  private JLabel     info;

  private Alarma     alarma;

/**
 * Constructor
 */
public SensorSwing()
{
  // Ventana (JFrame)

  frame = new JFrame("Medidor de… ");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(300,100);

  // Panel (JPanel)
  // …donde se colocan los controles de la interfaz

  panel = new JPanel(new GridLayout(3, 1));

  // Controles de la interfaz de usuario
  // -----------------------------------

  editor = new JTextField(5);
  info   = new JLabel("...", SwingConstants.CENTER);
  button = new JButton("Actualizar");
```

```
    // Manejador de eventos asociado al botón
    button.addActionListener(this);

    // Colocar los controles en el panel
    panel.add(editor);
    panel.add(button);
    panel.add(info);

    // Botón por defecto
    frame.getRootPane().setDefaultButton(button);

    // Añadir el panel a la ventana
    frame.getContentPane().add(panel);

    // Mostrar la ventana
    frame.setVisible(true);
  }

  /**
   * Alarma asociada
   */

  public void setAlarma (Alarma alarma)
  {
    this.alarma = alarma;
  }

  /**
   * Pulsación del botón.
   */

  public void actionPerformed(ActionEvent event)
  {
    double valor = Double.parseDouble(editor.getText());

    setValorActual(valor);
    info.setText("Valor actual = " + valor);

    if (alarma!=null) {
       alarma.comprobar();
    }
  }
}
```

Timbre.java

```
import java.awt.Toolkit;


/**
 * Timbre
 */

public class Timbre
{

  private static final int PITIDOS = 5;

  /**
   * Activar el timbre
   */

  public void activar ()
  {
    int i;

    for (i=0; i<PITIDOS; i++) {
        Toolkit.getDefaultToolkit().beep();

        try {
          Thread.sleep(500);
        } catch (InterruptedException e) {
        }
    }
  }

  /**
   * Desactivar el timbre
   */

  public void desactivar ()
  {
  }

}
```

Bombilla.java

Una ventana roja o verde en función del estado de la bombilla…

```
import java.awt.*;
import javax.swing.*;

/**
 * Bombilla
 */

public class Bombilla
{
  private JFrame frame;

  /**
   * Constructor
   */

  public Bombilla()
  {
    frame = new JFrame("Bombilla");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(200,200);
    frame.setVisible( true );

    apagar();
  }

  /**
   * Encender la bombilla
   */

  public void encender ()
  {
    frame.getContentPane().setBackground( Color.red );
  }

  /**
   * Apagar la bombilla
   */

  public void apagar ()
  {
    frame.getContentPane().setBackground( Color.green );
  }
}
```