

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 Stroustrup (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 coincide 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 únicamente:

- 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*):
 - o Se protegen los datos de accesos indebidos.
 - o Se distingue entre interfaz e implementación.
 - o 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 única)
- 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:

Cuenta

```
public class Cuenta  
{  
    ...  
}
```

Declaración en Java de una clase.

Representación de una clase con sus atributos:

Cuenta

-balance
-límite

```
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:

Cuenta

-balance : Dinero = 0
-límite : Dinero

```
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:

| Cuenta |
|-------------|
| -balance |
| -límite |
| +ingresar() |
| +retirar() |

```
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:

| Cuenta |
|---------------------------------|
| -balance : Dinero = 0 |
| -límite : Dinero |
| +ingresar(in cantidad : Dinero) |
| +retirar(in cantidad : Dinero) |

```
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:

| Motocicleta |
|-------------|
| -matrícula |
| -color |
| -velocidad |
| -en_marcha |
| +arrancar() |
| +acelerar() |
| +frenar() |
| +girar() |

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.

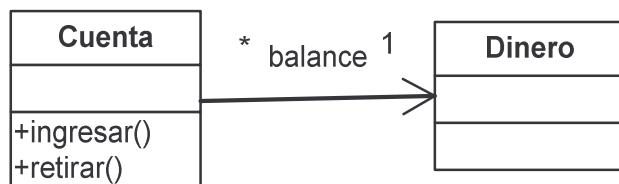


Gráficamente, se muestra como una línea continua que une las clases relacionadas entre sí.

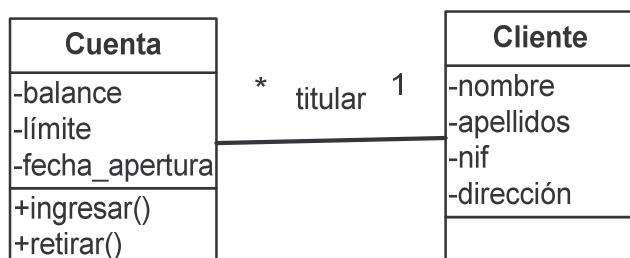
Navegación de las asociaciones

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).

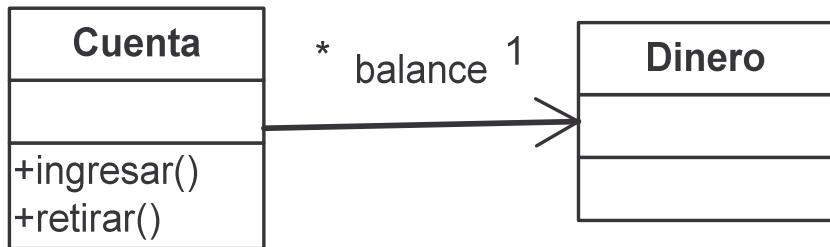
Gráficamente, cuando la asociación es unidireccional, la línea termina en una punta de flecha que indica el sentido de la asociación:



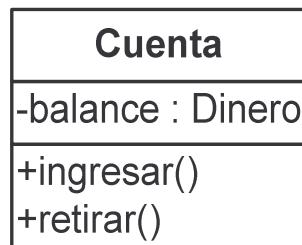
Asociación unidireccional



Asociación bidireccional



equivale a



```

class Cuenta
{
    private Dinero balance;

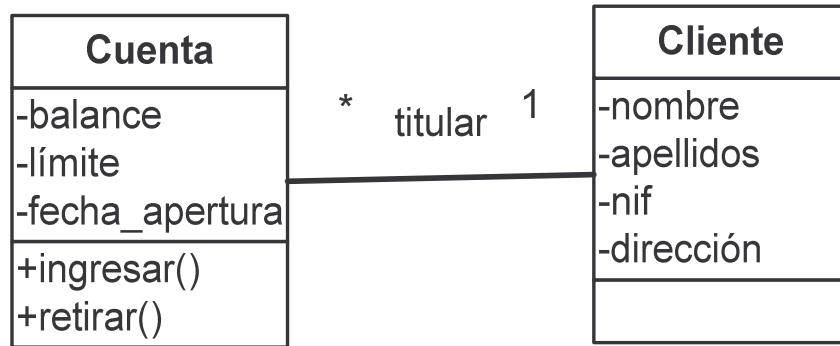
    public void ingresar (Dinero cantidad)
    {
        balance += cantidad;
    }

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

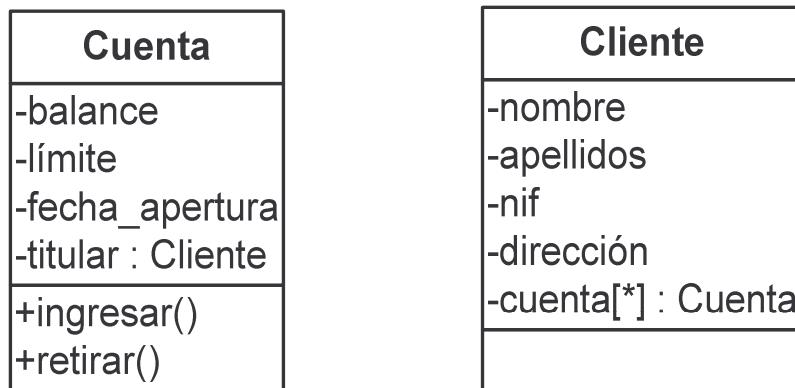
    public Dinero getSaldo ( )
    {
        return balance;
    }
}

```

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) |

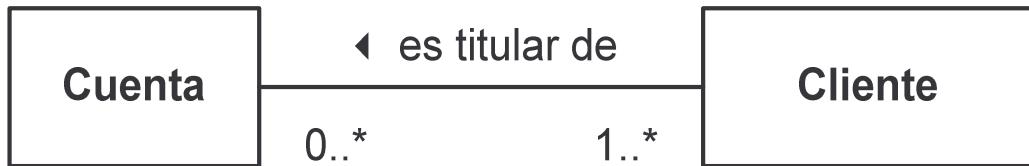
- Cuando la multiplicidad mínima es 0, la relación es opcional.
- Una multiplicidad mínima mayor o igual que 1 establece una relación obligatoria.



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.

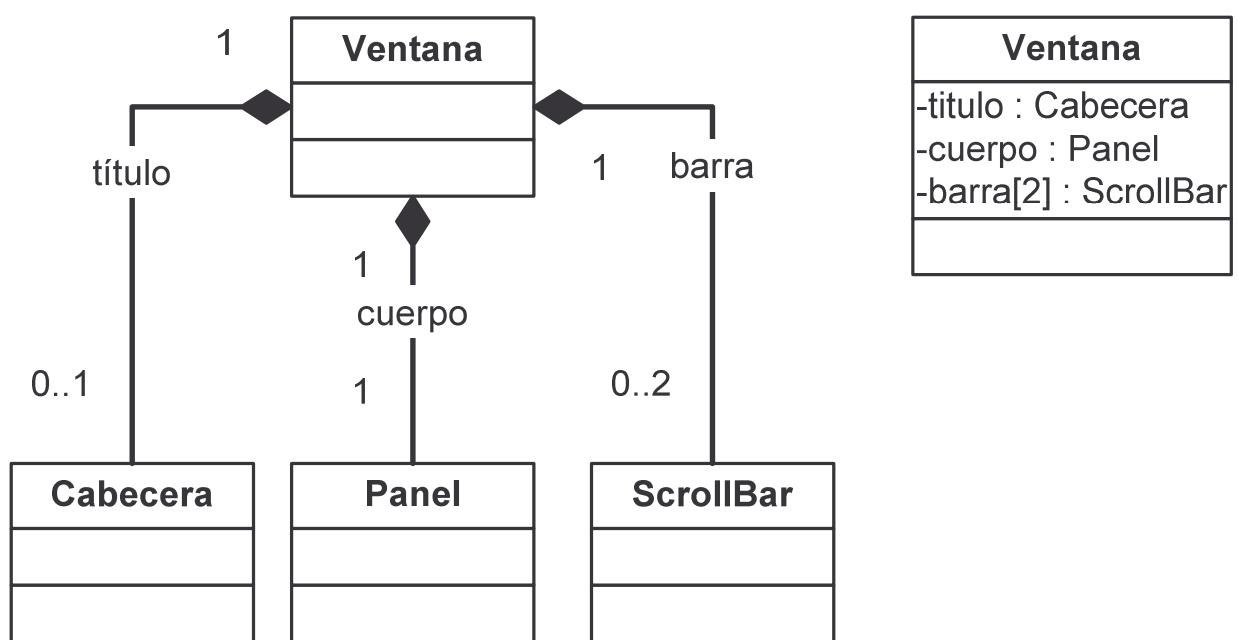
Agregación

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.

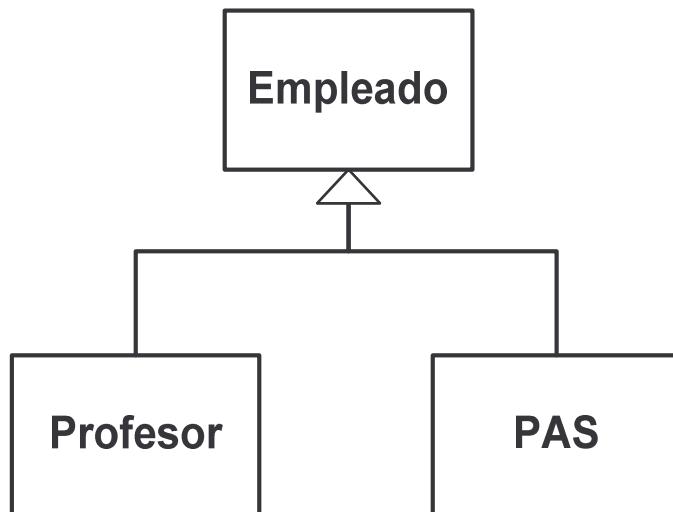
NOTA:

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...).



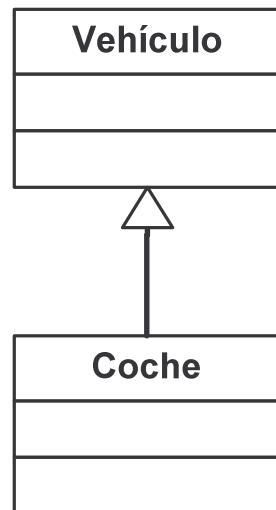
```
public class Empleado
{
    ...
}

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.



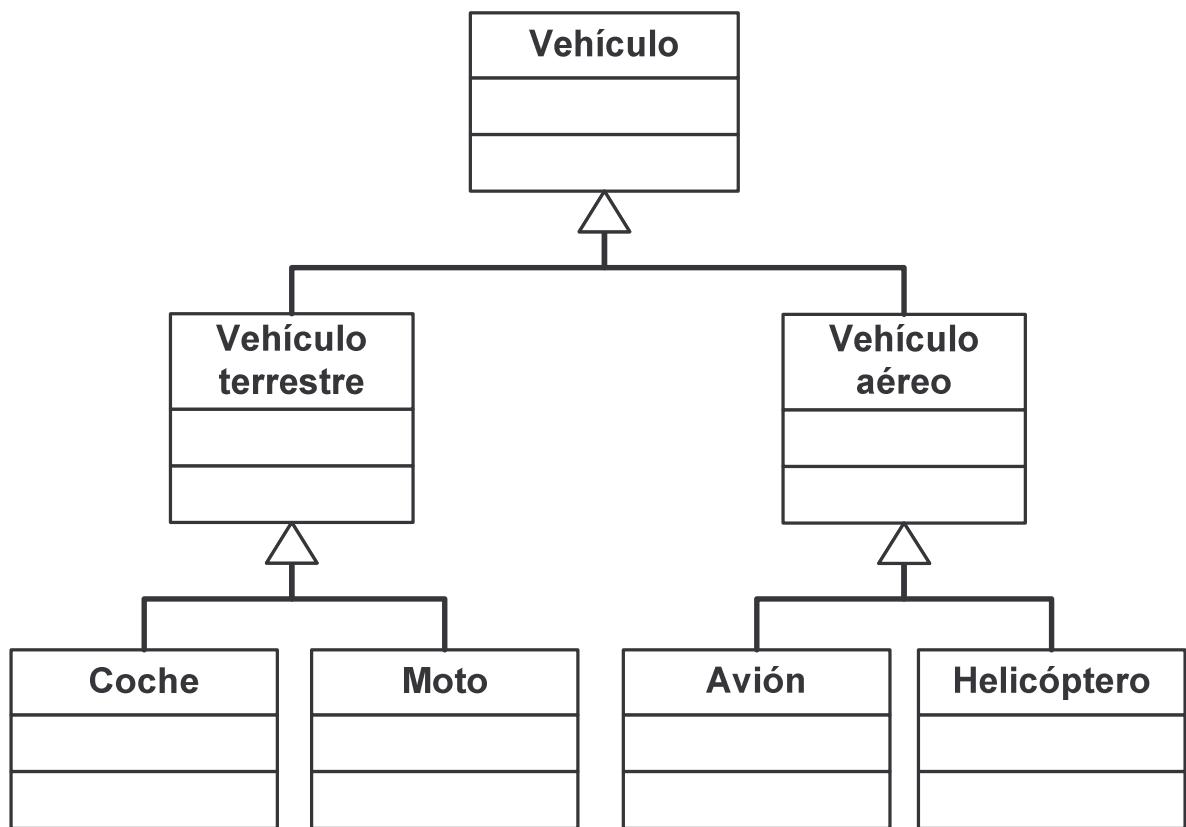
Instancias: **coches ⊂ vehículos**

- 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:

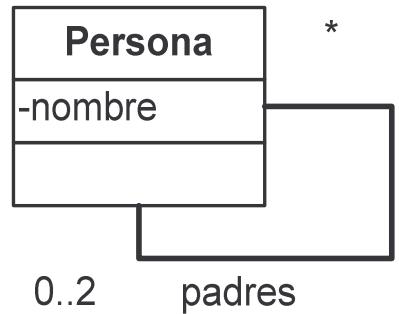
| Visibilidad | Significado | Java | UML |
|-------------|--|-----------|-----|
| Pública | Se puede acceder al miembro de la clase desde cualquier lugar. | public | + |
| Protegida | Sólo se puede acceder al miembro de la clase desde la propia clase o desde una clase que herede de ella. | protected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

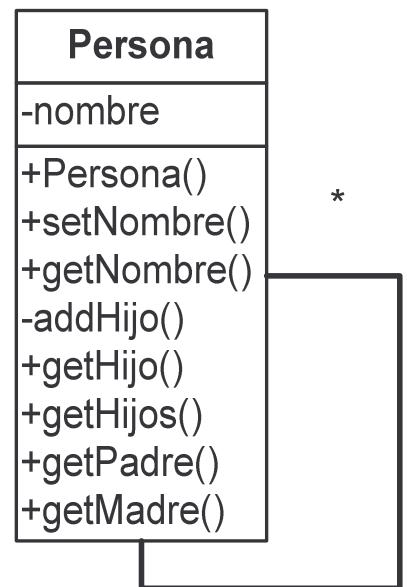
juan.nombre = "Juan";
carlos.nombre = "Carlos";
silvia.nombre = "Silvia";

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



0..2 padres

```
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”.

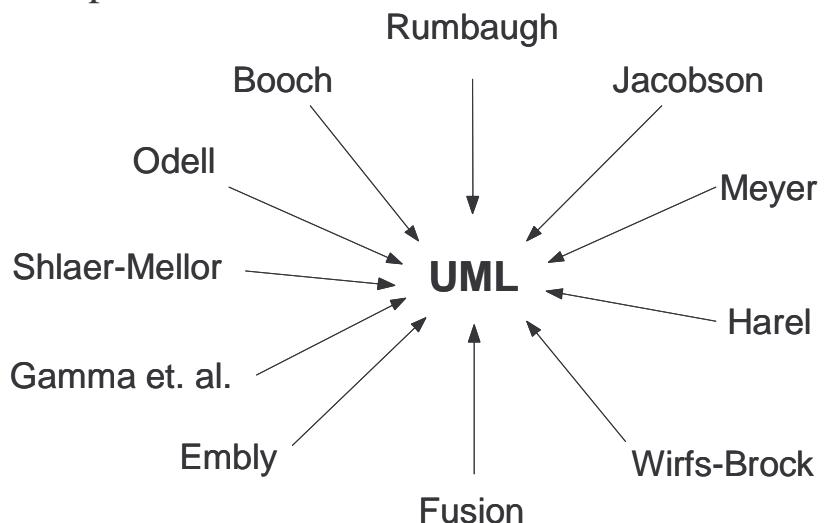
Todo sistema puede describirse desde distintos puntos de vista:

- 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.

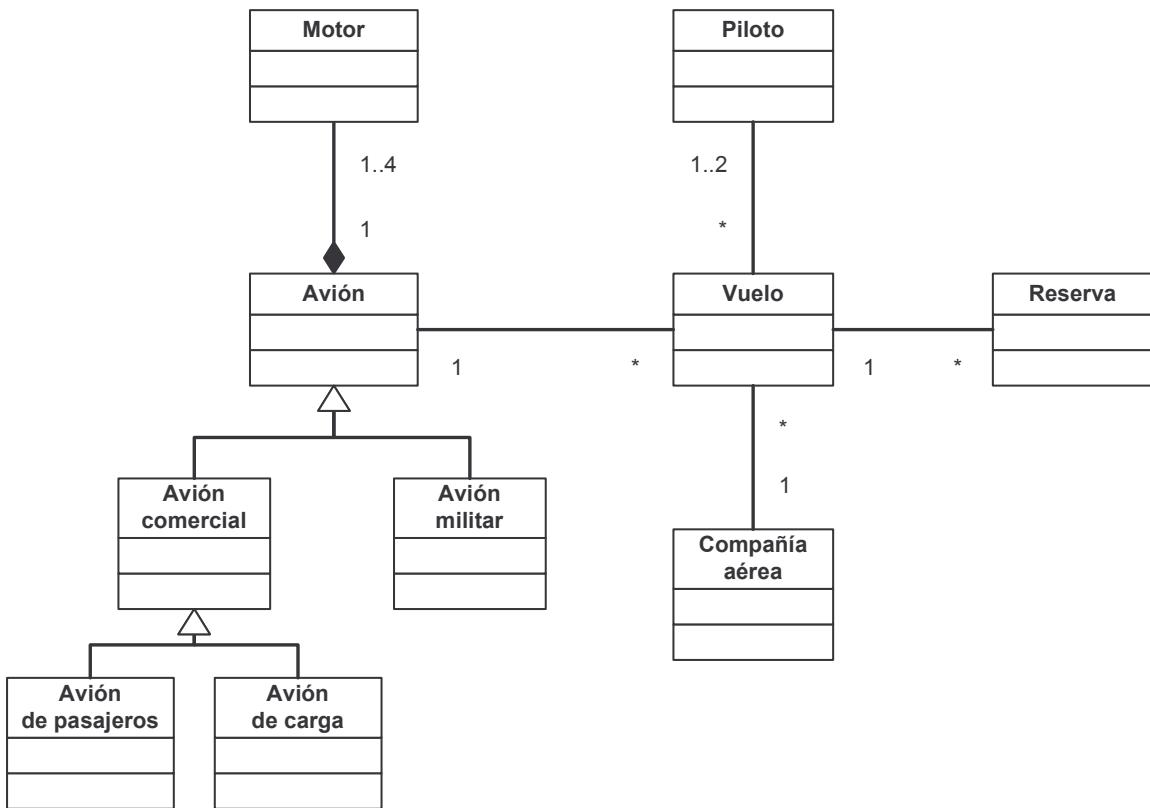


Inconvenientes de UML

- 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]]

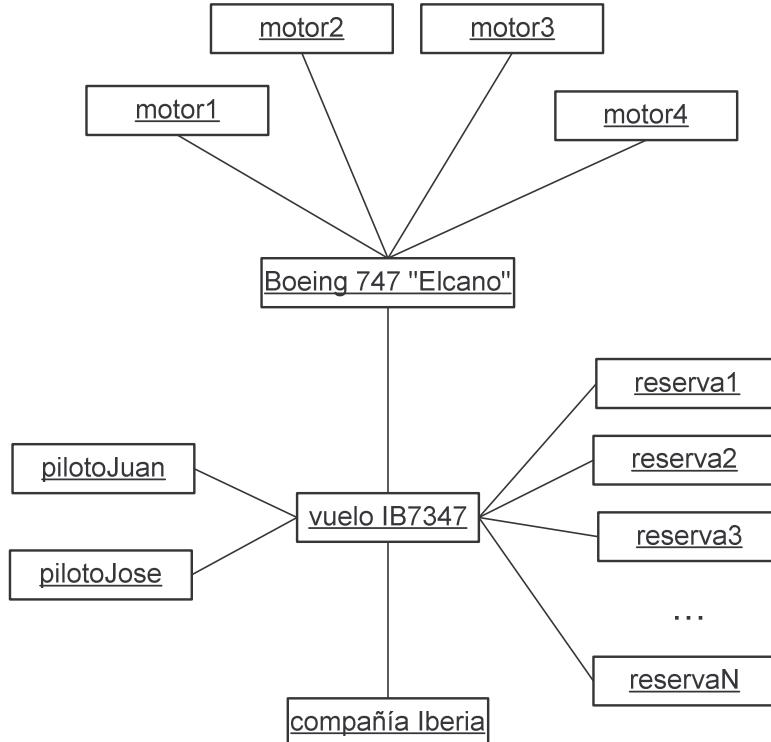
Operaciones

[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:

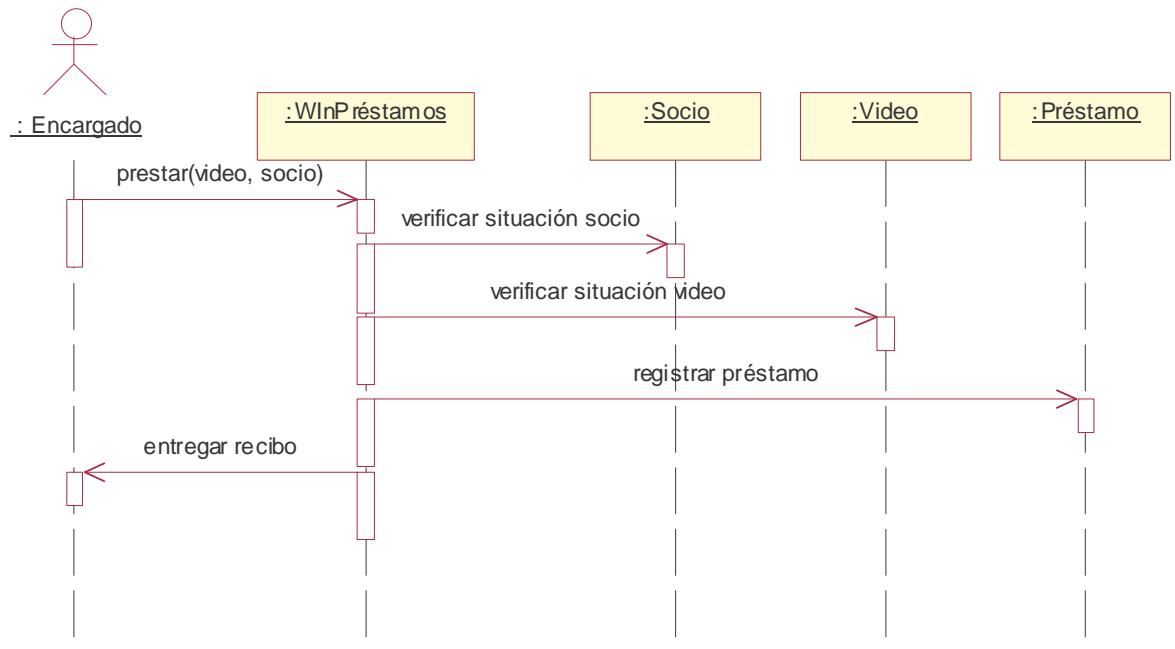
Los identificadores subrayados indican que se trata de objetos.

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

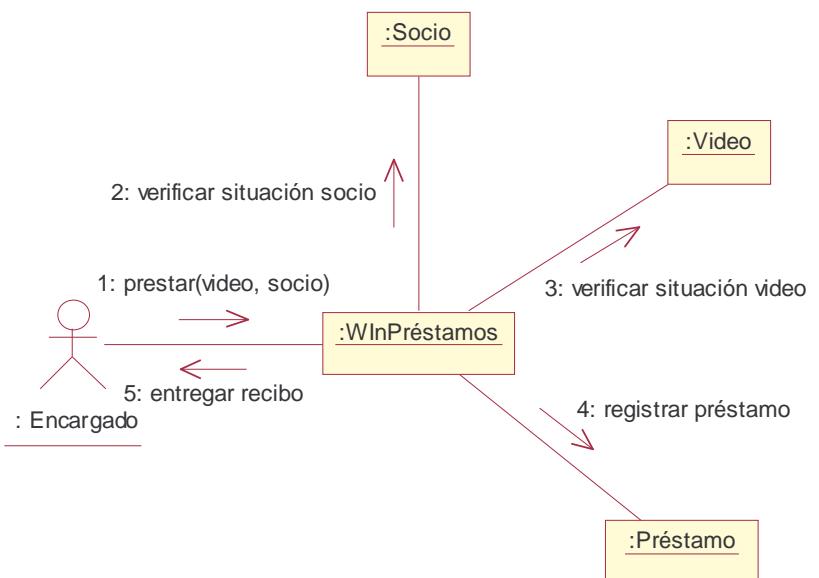
Resaltan la ordenación temporal de los mensajes que se intercambian.



Diagramas de comunicación (UML 2.0)

= Diagramas de colaboración (UML 1.x)

Resaltan la organización estructural
de los objetos que 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 (cuando 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.

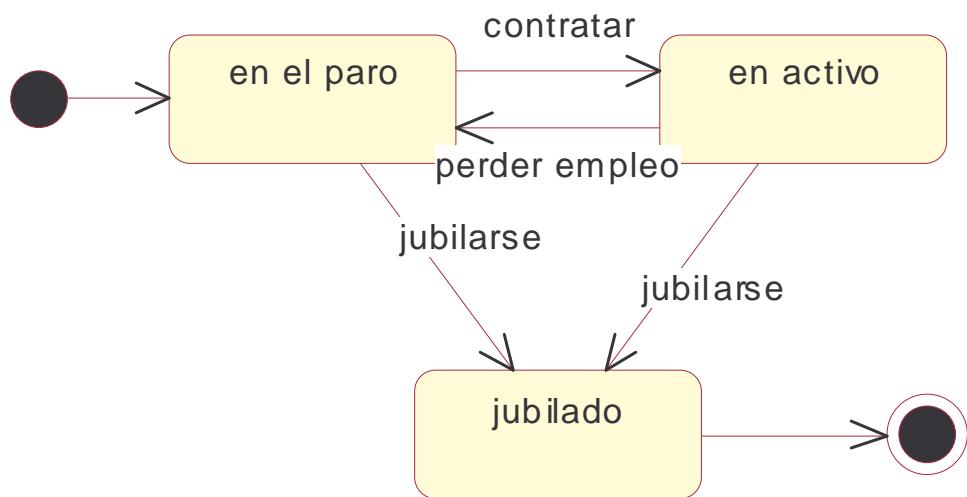
Otros diagramas UML para representar aspectos dinámicos del sistema

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



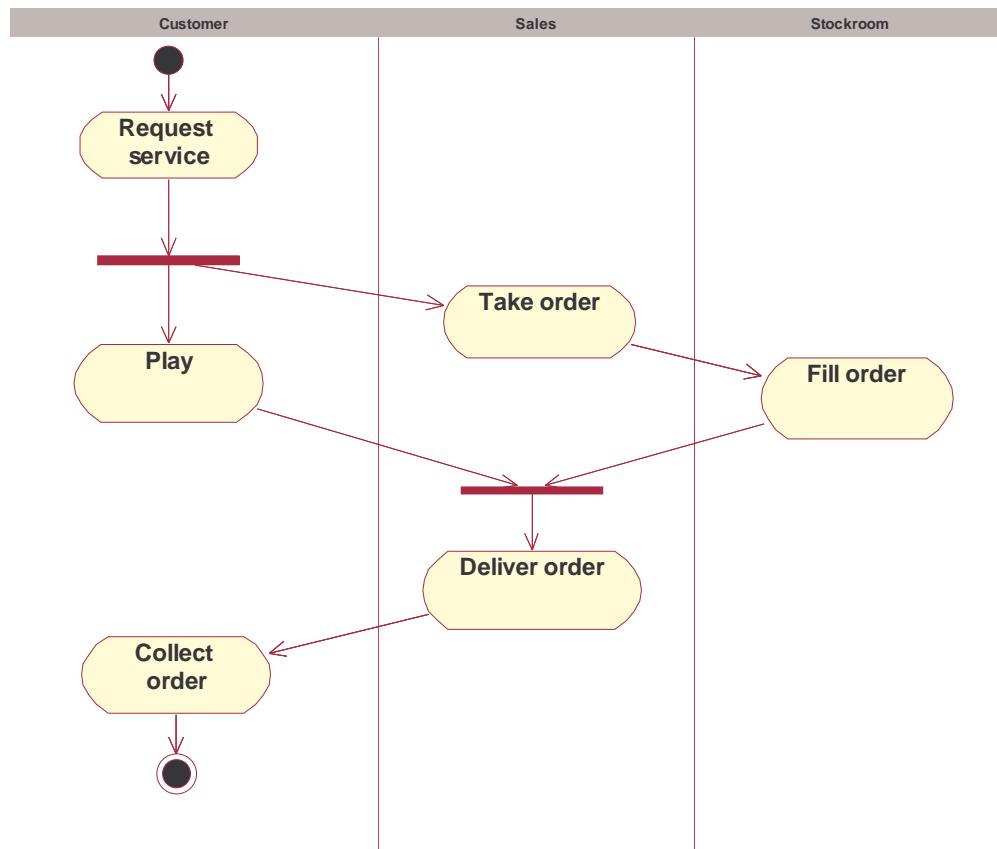
Los diagramas de uso se suelen utilizar en el modelado del sistema desde el punto de vista de sus usuarios para representar las acciones que realiza cada tipo de usuario.

- **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).

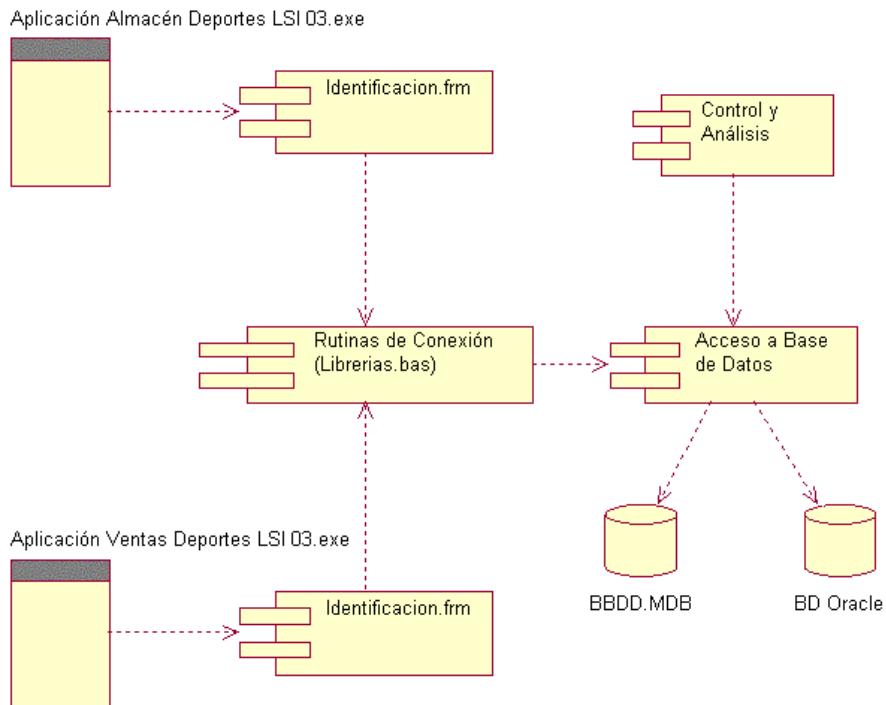
- **Diagramas de actividades**
(flujo de control en el sistema)



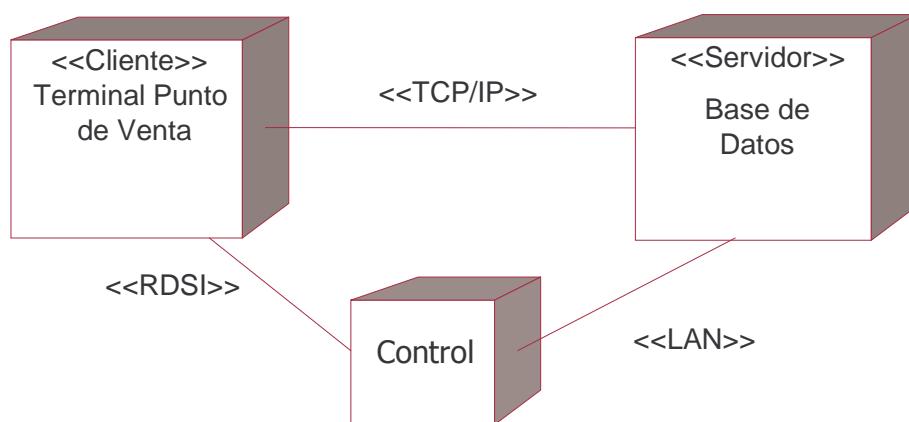
Los diagramas de actividades muestran el orden en el que se van realizando tareas dentro de un sistema (el flujo de control de las actividades).

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

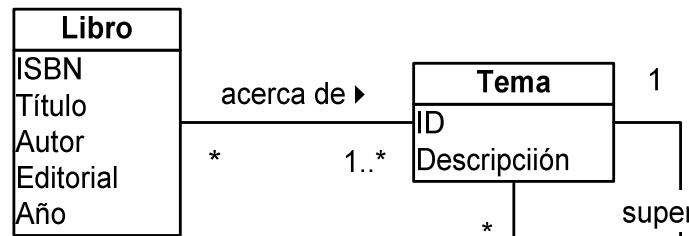
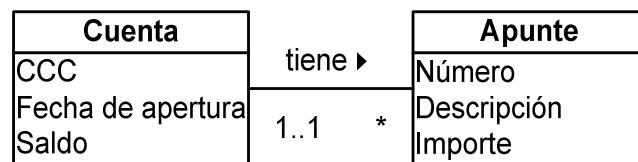
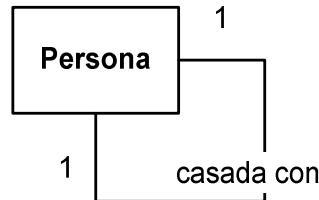
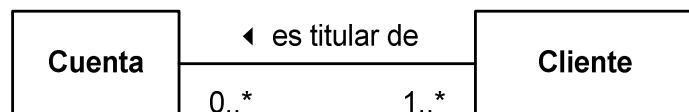
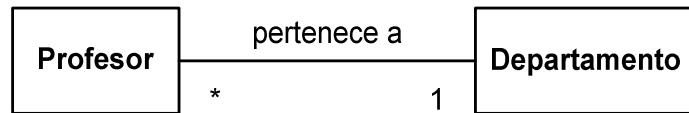
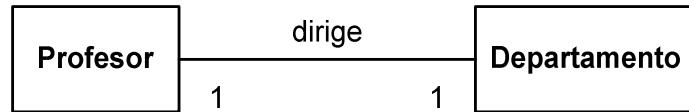
1. Proponga tres ejemplos de objetos del mundo real:

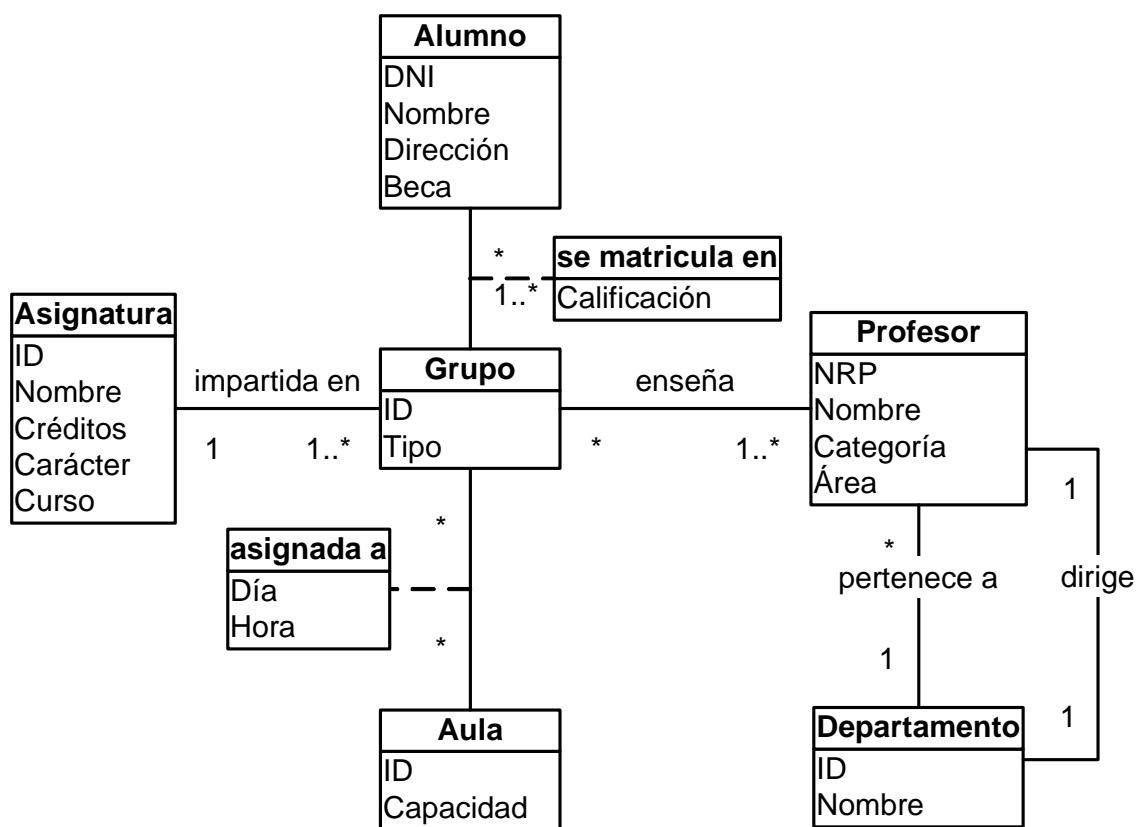
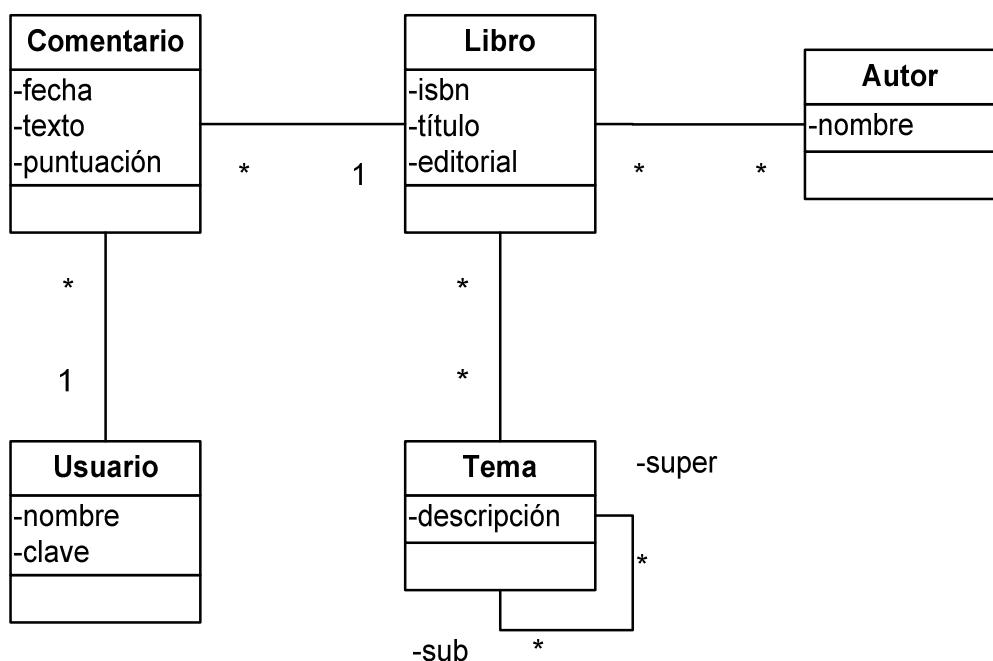
- Para cada uno de ellos, determine la clase a la que pertenecen.
- Asóciele a cada clase un identificador descriptivo adecuado.
- Enumere varios atributos y operaciones para cada una de las clases.
- Represente gráficamente las clases utilizando la notación UML.
- A partir de los diagramas UML, escriba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

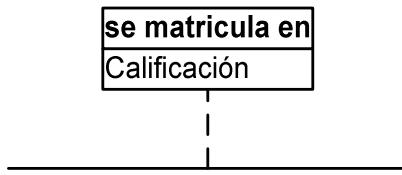
3. Definir adecuadamente las clases en Java que se derivan de los siguientes diagramas de clases UML:





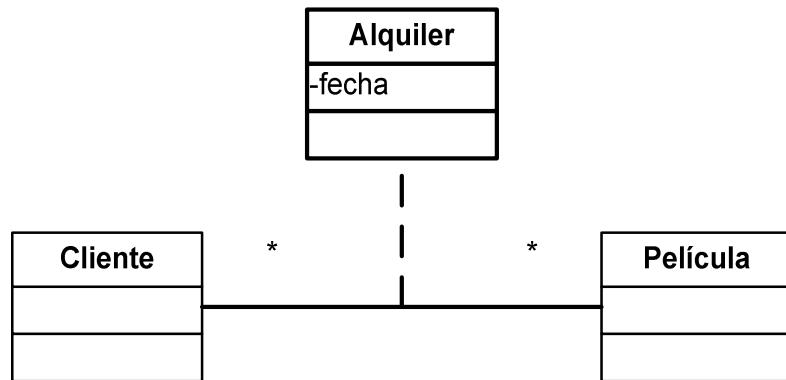
Nota: CLASES ASOCIACIÓN

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

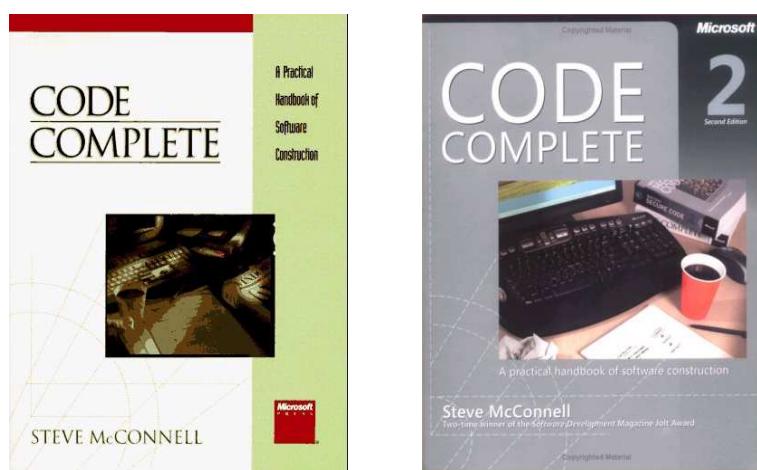
Cohesión y acoplamiento

Bibliografía

Steve McConnell: “*Code Complete*”.

Microsoft Press, 2004 [2^a edición] ISBN 0735619670.

Microsoft Press, 1994 [1^a 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

- ü Reducir la complejidad del programa (“divide y vencerás”).
- ü Eliminar código duplicado.
- ü Mejorar la legibilidad del código.
- ü Limitar los efectos de los cambios (aislar aspectos concretos).
- ü Ocultar detalles de implementación
(*ocultación de información*)
p.ej. algoritmos complejos
- ü Promover la reutilización de código
p.ej. componentes reutilizables y familias de productos
- ü Facilitar la adaptación del código a nuevas necesidades
p.ej. interfaces de usuario
- ü 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

- ü El nombre debe describir todo lo que hace el subprograma.
- ü Se deben evitar nombres genéricos que no dicen nada (vg. `calcular`)
- ü 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...

- ü Nos dicen cómo hemos de usar los objetos de una clase.
- ü 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:

Ü El método oculta los detalles de implementación.

Ü 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 qué 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.

- β 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.println(...);  
- 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 responsable.

- ü A partir de una referencia a un objeto, podemos llamar a uno de sus métodos con el **operador .**
- ü Tras el nombre del método, entre paréntesis, se han de indicar sus parámetros (si es que los tiene).
- ü 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:

- ü A través de los parámetros se especifican los datos de entrada que requiere el método para realizar su tarea.
- ü Los parámetros definidos en la cabecera del método se denominan **parámetros formales**.
- ü 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.
- ü 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.

- ü 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).
- ü 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).

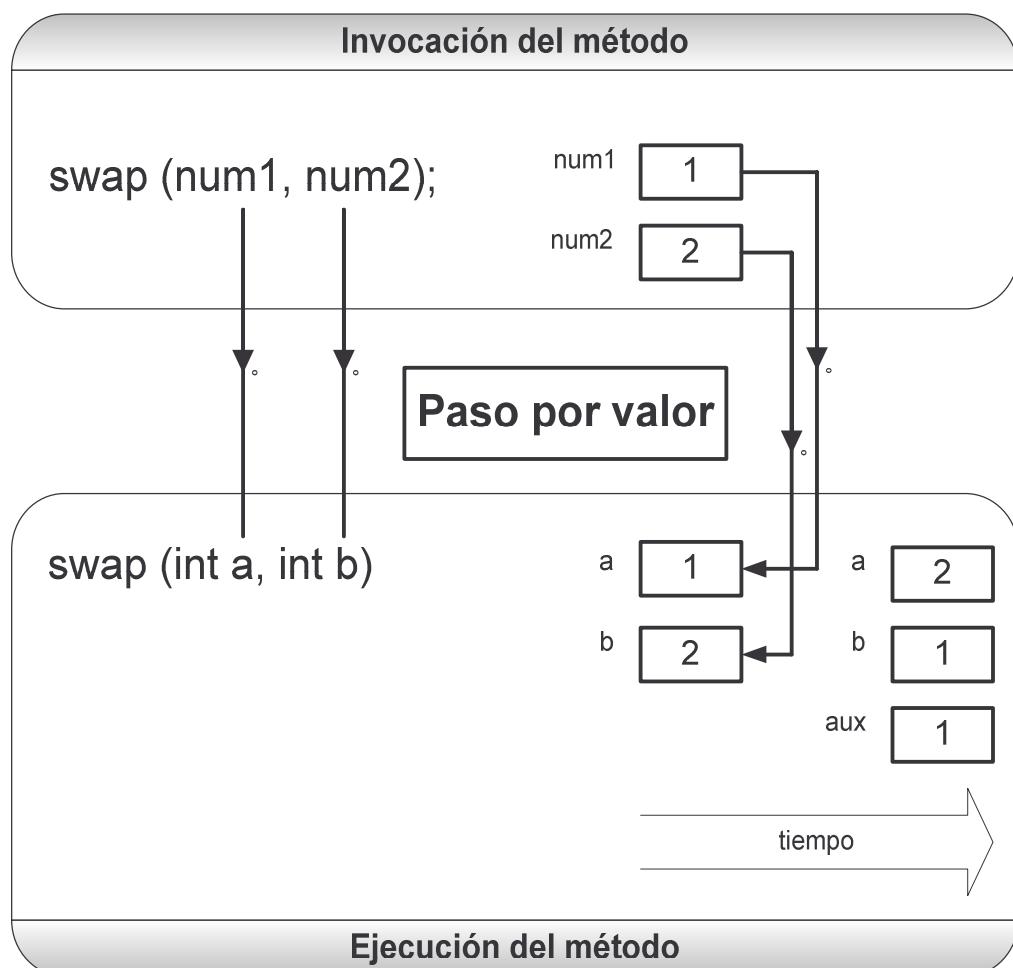
¡OJO! Como las variables de tipos no primitivos son, en realidad, referencias a objetos, lo que se copia en este caso es la referencia al objeto (no el objeto en sí).

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);
}
```

- ü 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

- ü 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 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`)

Ü 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 exponentiales (`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.

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.

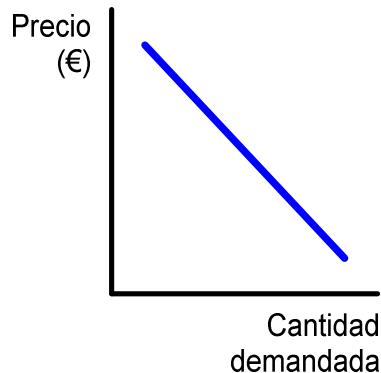
PISTA:

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:

- 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 pagararlo. 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.

- 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:

$$\text{coste}_{\text{total}} = \text{coste}_{\text{inicial}} + \text{unidades} * \text{coste}_{\text{marginal}}$$

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:

$$\text{ingresos} = \text{precio} * \text{unidades}$$

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:

$$\text{ingreso}_{\text{marginal}}(x) = \text{ingreso}_{\text{total}}(x) - \text{ingreso}_{\text{total}}(x-1)$$

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:

$$\text{ingreso}_{\text{marginal}}(x) = \text{demanda}(2x)$$

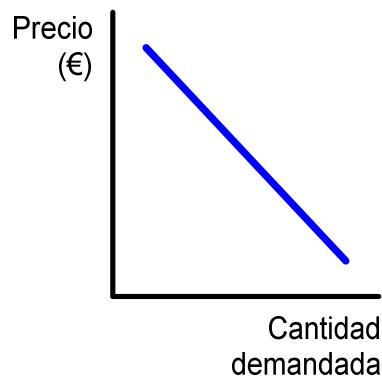
En este caso, el precio ideal seguirá siendo aquél para el que el ingreso marginal iguale al coste marginal, es decir aquél que hace que $\text{coste}_{\text{marginal}}(x)=\text{demanda}(2x)$

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

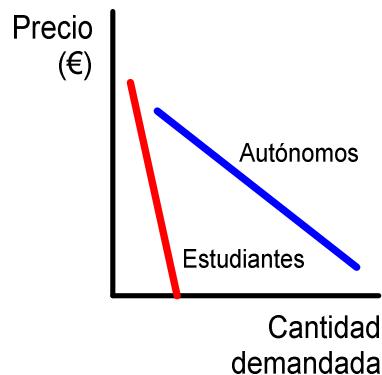
NOTA: En vez de crear tres programas distintos, cree un único programa que acceda a una clase encargada de calcular el precio ideal al que hay que vender el producto para una demanda concreta. A continuación, modifique la implementación de la clase sin alterar la implementación del programa principal (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?

Estructuras de control

Programación estructurada

Estructuras condicionales

- La sentencia `if`
- La cláusula `else`
- Encadenamiento y anidamiento
- El operador condicional `? :`
- La sentencia `switch`

Estructuras repetitivas/iterativas

- El bucle `while`
- El bucle `for`
- El bucle `do...while`
- Bucles anidados

Cuestiones de estilo

Las estructuras de control
controlan la ejecución de las instrucciones de un programa
(especifican el orden en el que se realizan las acciones)

Programación estructurada

IDEA CENTRAL:

Las estructuras de control de un programa
sólo deben tener **un punto de entrada y un punto de salida.**

La programación estructurada...

mejora la productividad de los programadores.

mejora la legibilidad del código resultante.

La ejecución de un programa estructurado progresá **disciplinadamente**,
en vez de saltar de un sitio a otro de forma impredecible

Gracias a ello, los programas...

resultan más fáciles de probar

se pueden depurar más fácilmente

se pueden modificar con mayor comodidad

En programación estructurada sólo se emplean tres construcciones:

Ü Secuencia

Conjunto de sentencias que se ejecutan en orden

Ejemplos:

Sentencias de asignación y llamadas a rutinas.

Ü Selección

Elige qué sentencias se ejecutan en función de una condición.

Ejemplos:

Estructuras de control condicional `if-then-else` y `case/switch`

Ü Iteración

Las estructuras de control repetitivas repiten conjuntos de instrucciones.

Ejemplos:

Bucles `while`, `do...while` y `for`.

Teorema de Böhm y Jacopini (1966):

Cualquier programa de ordenador
puede diseñarse e implementarse
utilizando únicamente las tres construcciones estructuradas
(secuencia, selección e iteración; esto es, sin sentencias `goto`).

Böhm, C. & Jacopini, G.: “Flow diagrams, Turing machines, and languages only with two formation rules”. Communications of the ACM, 1966, Vol. 9, No. 5, pp. 366-371

Dijkstra, E.W.: “Goto statement considered harmful”. Communications of the ACM, 1968, Vol. 11, No. 3, pp. 147-148

Estructuras de control condicionales

Por defecto,
las instrucciones de un programa se ejecutan secuencialmente:

El orden secuencial de ejecución no altera el flujo de control del programa respecto al orden de escritura de las instrucciones.

Sin embargo, al describir la resolución de un problema, es normal que tengamos que tener en cuenta condiciones que influyen sobre la secuencia de pasos que hay que dar para resolver el problema:

Según se cumplan o no determinadas condiciones,
la secuencia de pasos involucrada en la realización de una tarea será diferente

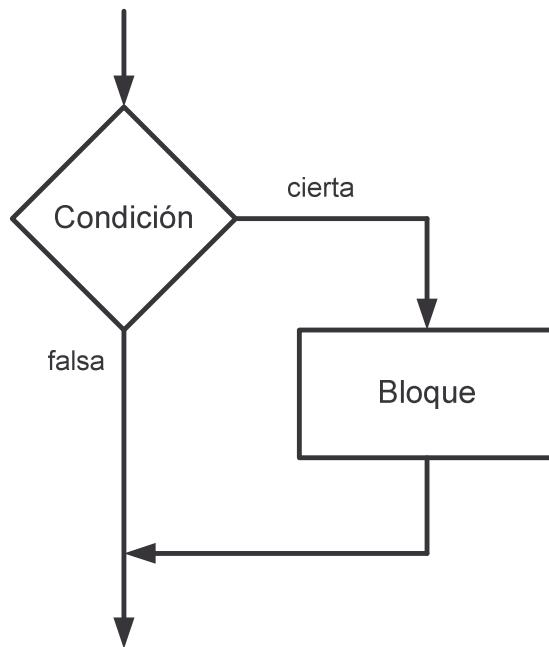
Las estructuras de control condicionales o selectivas nos permiten decidir qué ejecutar y qué no en un programa.

Ejemplo típico

Realizar una división sólo si el divisor es distinto de cero.

*La estructura de control condicional **if***

La sentencia **if** nos permite elegir si se ejecuta o no un bloque de instrucciones.



Sintaxis

```
if (condición)  
    sentencia;
```

```
if (condición) {  
    bloque  
}
```

donde **bloque** representa un bloque de instrucciones.

Bloque de instrucciones:

Secuencia de instrucciones encerradas entre dos llaves { }

Consideraciones acerca del uso de la sentencia `if`

- Olvidar los paréntesis al poner la condición del `if` es un error sintáctico (los paréntesis son necesarios)
- No hay que confundir el operador de comparación `==` con el operador de asignación `=`
- Los operadores de comparación `==`, `!=`, `<=` y `>=` han de escribirse sin espacios.
- `=>` y `=<` no son operadores válidos en Java.
- El fragmento de código afectado por la condición del `if` debe sangrarse para que visualmente se interprete correctamente el ámbito de la sentencia `if`:

```
if (condición) {  
    // Aquí se incluye el código  
    // que ha de ejecutarse  
    // cuando se cumple la condición del if  
}
```

- Aunque el uso de llaves no sea obligatorio cuando el `if` sólo afecta a una sentencia, es recomendable ponerlas siempre para delimitar explícitamente el ámbito de la sentencia `if`.

Error común:

```
if (condición);  
    sentencia;
```

es interpretado como

```
if (condición)  
    ;      // Sentencia vacía  
    sentencia;
```

¡¡¡La sentencia siempre se ejecutaría!!!

Ejemplo

Comparación de números (Deitel & Deitel)

| Operador | Significado |
|----------|---------------|
| == | Igual |
| != | Distinto |
| < | Menor |
| > | Mayor |
| <= | Menor o igual |
| >= | Mayor o igual |

```
import javax.swing.JOptionPane;

public class Comparison
{
    public static void main( String args[] )
    {
        // Declaración de variables

        String primerDato, segundoDato;
        String resultado;
        int      dato1, dato2;

        primerDato  = JOptionPane.showInputDialog
                        ( "Primer dato:" );
        segundoDato = JOptionPane.showInputDialog
                        ( "Segundo dato:" );

        dato1 = Integer.parseInt( primerDato );
        dato2 = Integer.parseInt( segundoDato );

        resultado = " ";

        if ( dato1 == dato2 )
            resultado += dato1 + " == " + dato2;

        if ( dato1 != dato2 )
            resultado += dato1 + " != " + dato2;
    }
}
```

```

if ( dato1 < dato2 )
    resultado += "\n" + dato1 + " < " + dato2;

if ( dato1 > dato2 )
    resultado += "\n" + dato1 + " > " + dato2;

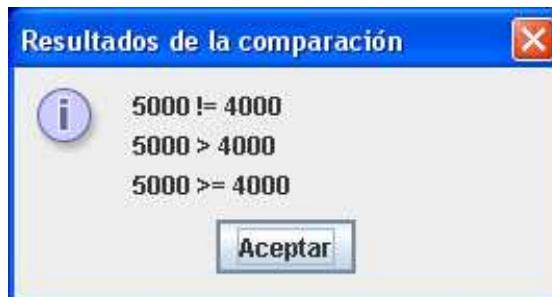
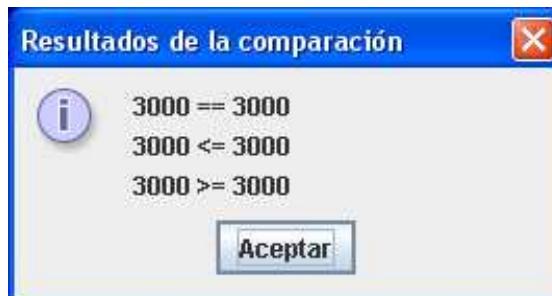
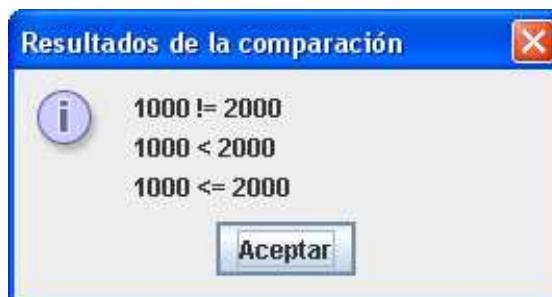
if ( dato1 <= dato2 )
    resultado += "\n" + dato1 + " <= " + dato2;

if ( dato1 >= dato2 )
    resultado += "\n" + dato1 + " >= " + dato2;

JOptionPane.showMessageDialog
( null, resultado,
  "Resultados de la comparación",
  JOptionPane.INFORMATION_MESSAGE );

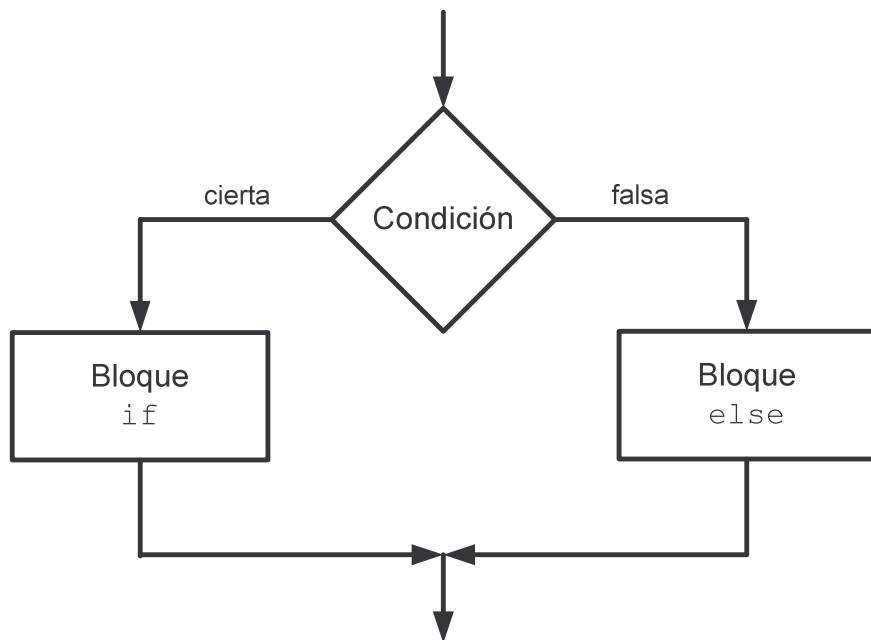
System.exit( 0 );
}
}

```



La cláusula else

Una sentencia **if**, cuando incluye la cláusula **else**, permite ejecutar un bloque de código si se cumple la condición y otro bloque de código diferente si la condición no se cumple.



Sintaxis

```
if (condición)
    sentencial;
else
    sentencia2;
```

```
if (condición) {
    bloquel
} else {
    bloque2
}
```

Los bloques de código especificados representan dos alternativas complementarias y excluyentes

Ejemplo

```
import javax.swing.JOptionPane;

public class IfElse
{
    public static void main( String args[ ] )
    {
        String primerDato, segundoDato;
        String resultado;
        int     dato1, dato2;

        primerDato = JOptionPane.showInputDialog
                      ( "Primer dato:" );
        segundoDato = JOptionPane.showInputDialog
                      ( "Segundo dato:" );

        dato1 = Integer.parseInt( primerDato );
        dato2 = Integer.parseInt( segundoDato );

        resultado = " ";

        if ( dato1 == dato2 )
            resultado += dato1 + " == " + dato2;
        else
            resultado += dato1 + " != " + dato2;

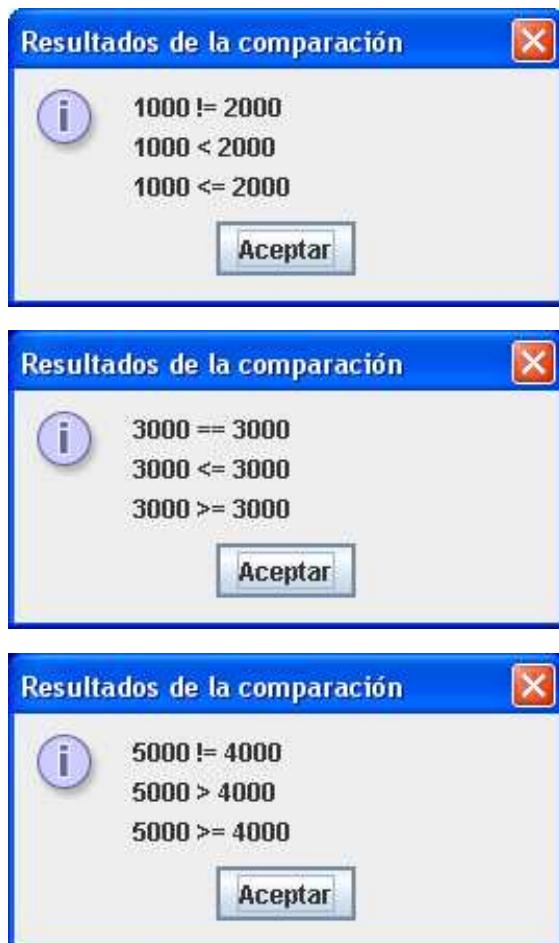
        if ( dato1 < dato2 )
            resultado += "\n" + dato1 + " < " + dato2;
        else
            resultado += "\n" + dato1 + " >= " + dato2;

        if ( dato1 > dato2 )
            resultado += "\n" + dato1 + " > " + dato2;
        else
            resultado += "\n" + dato1 + " <= " + dato2;

        JOptionPane.showMessageDialog
                      ( null, resultado,
                        "Resultados de la comparación",
                        JOptionPane.INFORMATION_MESSAGE );

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

Con esta versión del programa de comparación de números obtenemos los mismos resultados que antes, si bien nos ahorrados tres comparaciones:



La sentencia

```
if (condición)
    sentencial;
else
    sentencia2;
```

es equivalente a

```
if (condición)
    sentencial;

if (!condición)
    sentencia2;
```

Nota acerca de la comparación de objetos

Cuando el operador == se utiliza para comparar objetos,
lo que se compara son las referencias a los objetos
y no el estado de los objetos en sí.

Ejemplo

```
public class Complex
{
    private double real;
    private double imag;

    // Constructor

    public Complex (double real, double imag)
    {
        this.real = real;
        this.imag = imag;
    }

    // equals se suele definir para comparar objetos

    public boolean equals (Complex c)
    {
        return (this.real == c.real)
            && (this.imag == c.imag);
    }
}

...
Complex c1 = new Complex(2,1);
Complex c2 = new Complex(2,1);

if (c1 == c2)
    System.out.println("Las referencias son iguales");
else
    System.out.println("Las referencias no son iguales");

if (c1.equals(c2))
    System.out.println("Los objetos son iguales");
else
    System.out.println("Los objetos no son iguales");
```

Encadenamiento

Las sentencias `if` se suelen encadenar:

if ... else if ...

```
import javax.swing.JOptionPane;

public class IfChain
{
    public static void main( String args[] )
    {
        String entrada;
        String resultado;
        float nota;

        entrada = JOptionPane.showInputDialog
                    ( "Calificación numérica:" );
        nota = Float.parseFloat( entrada );

        if ( nota >= 9 )
            resultado = "Sobresaliente";
        else if ( nota >= 7 )
            resultado = "Notable";
        else if ( nota >= 5 )
            resultado = "Aprobado";
        else
            resultado = "Suspensos";

        JOptionPane.showMessageDialog
                    ( null, resultado,
                      "Calificación final",
                      JOptionPane.INFORMATION_MESSAGE );

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

El if encadenado anterior equivale a:

```
import javax.swing.JOptionPane;

public class IfChain2
{
    public static void main( String args[ ] )
    {
        String entrada;
        String resultado;
        float nota;

        entrada = JOptionPane.showInputDialog
                    ( "Calificación numérica:" );
        nota = Float.parseFloat( entrada );

        resultado = "";

        if ( nota >= 9 )
            resultado = "Sobresaliente";

        if ( (nota>=7) && (nota<9) )
            resultado = "Notable";

        if ( (nota>=5) && (nota<7) )
            resultado = "Aprobado";

        if ( nota < 5 )
            resultado = "Suspensos";

        JOptionPane.showMessageDialog
                    ( null, resultado,
                      "Calificación final",
                      JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );
    }
}
```

Anidamiento

Las sentencias `if` también se pueden anidar unas dentro de otras.

Ejemplo: Resolución de una ecuación de primer grado $ax+b=0$

```
import javax.swing.JOptionPane;

public class IfNested
{
    public static void main( String args[ ] )
    {
        String entrada;
        String resultado;
        float a,b,x;

        entrada = JOptionPane.showInputDialog
                  ( "Coeficiente a" );
        a = Float.parseFloat( entrada );

        entrada = JOptionPane.showInputDialog
                  ( "Coeficiente b" );
        b = Float.parseFloat( entrada );

        if (a!=0) {
            x = -b/a;
            resultado = "La solución es " + x;
        } else {
            if (b!=0) {
                resultado = "No tiene solución.";
            } else {
                resultado = "Solución indeterminada.";
            }
        }

        JOptionPane.showMessageDialog
                  ( null, resultado,
                    "Solución de la ecuación de primer grado",
                    JOptionPane.INFORMATION_MESSAGE );

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

El if anidado anterior equivale a ...

```
import javax.swing.JOptionPane;

public class IfNested2
{
    public static void main( String args[ ] )
    {
        String entrada;
        String resultado;
        float a,b,x;

        entrada = JOptionPane.showInputDialog
                  ( "Coeficiente a" );
        a = Float.parseFloat( entrada );

        entrada = JOptionPane.showInputDialog
                  ( "Coeficiente b" );
        b = Float.parseFloat( entrada );

        resultado = " ";

        if ( a!=0 ) {
            x = -b/a;
            resultado = "La solución es " + x;
        }

        if ( (a==0) && (b!=0) ) {
            resultado = "No tiene solución.";
        }

        if ( (a==0) && (b==0) ) {
            resultado = "Solución indeterminada.";
        }

        JOptionPane.showMessageDialog
                  ( null, resultado,
                    "Solución de la ecuación de primer grado",
                    JOptionPane.INFORMATION_MESSAGE );

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

En este caso, se realizan 5 comparaciones en vez de 2.

El operador condicional ?:

Java proporciona una forma de abreviar una sentencia `if`

El operador condicional ?:
permite incluir una condición dentro de una expresión.

Sintaxis

```
variable = condición? expresión1: expresión2;
```

“equivale” a

```
if (condición)
    variable = expresión1;
else
    variable = expresión2;
```

Sólo se evalúa una de las expresiones,
por lo que deberemos ser cuidadosos con los efectos colaterales.

Ejemplos

```
max = (x>y)? x : y;
```

```
min = (x<y)? x : y;
```

```
med = (x<y)? ((y<z)? y: ((z<x)? x: z)) :
        ((x<z)? x: ((z<y)? y: z));
```

```
nick = (nombre!=null)? nombre : "desconocido";
```

Selección múltiple con la sentencia switch

Permite seleccionar entre varias alternativas posibles

Sintaxis

```
switch (expresión) {  
  
    case expr_cte1:  
        bloque1;  
        break;  
  
    case expr_cte2:  
        bloque2;  
        break;  
    ...  
    case expr_cteN:  
        bloqueN;  
        break;  
  
    default:  
        bloque_por_defecto;  
}
```

- Se selecciona a partir de la evaluación de una única expresión.
- La expresión del switch ha de ser de tipo entero (int).
- Los valores de cada caso del switch han de ser constantes.
- En Java, cada bloque de código de los que acompañan a un posible valor de la expresión entera ha de terminar con una sentencia break;
- La etiqueta default marca el bloque de código que se ejecuta por defecto (cuando al evaluar la expresión se obtiene un valor no especificado por los casos anteriores del switch).
- En Java, se pueden poner varias etiquetas seguidas acompañando a un único fragmento de código si el fragmento de código que ha de ejecutarse es el mismo para varios valores de la expresión entera que gobierna la ejecución del switch.

Ejemplo

```
public class Switch
{
    public static void main( String args[ ] )
    {
        String resultado;
        int nota;

        // Entrada de datos
        ...

        switch (nota) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
                resultado = "Suspensos";
                break;

            case 5:
            case 6:
                resultado = "Aprobado";
                break;

            case 7:
            case 8:
                resultado = "Notable";
                break;

            case 9:
            case 10:
                resultado = "Sobresaliente";
                break;

            default:
                resultado = "Error";
        }

        // Salida de resultados
        ...
    }
}
```

Si tuviésemos que trabajar con datos de tipo real, no podríamos usar `switch` (usaríamos `ifs` encadenados).

Estructuras de control repetitivas/iterativas

A menudo es necesario ejecutar una instrucción o un bloque de instrucciones más de una vez.

Ejemplo

Implementar un programa que calcule la suma de N números leídos desde teclado.

Podríamos escribir un programa en el que apareciese repetido el código que deseamos que se ejecute varias veces, pero...

- ⌘ Nuestro programa podría ser demasiado largo.
- ⌘ Gran parte del código del programa estaría duplicado, lo que dificultaría su mantenimiento en caso de que tuviésemos que hacer cualquier cambio, por trivial que fuese éste.
- ⌘ Una vez escrito el programa para un número determinado de repeticiones (p.ej. sumar matrices 3x3), el mismo programa no podríamos reutilizarlo si necesitásemos realizar un número distinto de operaciones (p.ej. sumar matrices 4x4).

Las **estructuras de control repetitivas o iterativas**, también conocidas como “**bucles**”, nos permiten resolver de forma elegante este tipo de problemas. Algunas podemos usarlas cuando conocemos el número de veces que deben repetirse las operaciones. Otras nos permiten repetir un conjunto de operaciones mientras se cumpla una condición.

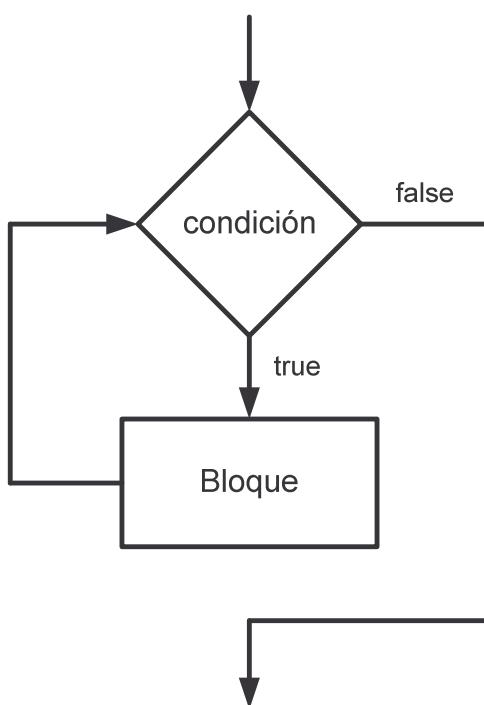
Iteración: Cada repetición de las instrucciones de un bucle.

El bucle while

Permite repetir la ejecución de un conjunto de sentencias mientras se cumpla una condición:

```
while (condición)  
    sentencia;
```

```
while (condición) {  
    bloque  
}
```



El bucle while terminará su ejecución cuando deje de verificarse la condición que controla su ejecución.

Si, inicialmente, no se cumple la condición, el cuerpo del bucle no llegará a ejecutarse.

MUY IMPORTANTE

En el cuerpo del bucle debe existir algo que haga variar el valor asociado a la condición que gobierna la ejecución del bucle.

Ejemplo

Tabla de multiplicar de un número

```
public class While1
{
    public static void main( String args[] )
    {
        int n; // Número
        int i; // Contador

        n = Integer.parseInt( args[0] );
        i = 0;

        while (i<=10) {
            System.out.println (n+ " x " +i+ " = " +(n*i));
            i++;
        }
    }
}
```

Ejemplo

Divisores de un número

```
public class While2
{
    public static void main( String args[] )
    {
        int n;
        int divisor;

        n = Integer.parseInt( args[0] );

        System.out.println("Los divisores son:");

        divisor = n;

        while (divisor>0) {

            if ((n%divisor) == 0)
                System.out.println(divisor);

            divisor--;
        }
    }
}
```

En los ejemplos anteriores,
se conoce de antemano el número de iteraciones
que han de realizarse (cuántas veces se debe ejecutar el bucle):

La expresión del `while` se convierte en una simple
comprobación del valor de una variable contador.

El contador es una variable que se incrementa o decrementa en
cada iteración y nos permite saber la iteración en la que nos
encontramos en cada momento.

En el cuerpo del bucle, siempre se incluye una sentencia

`contador++;`

o bien

`contador--;`

para que, eventualmente,
la condición del `while` deje de cumplirse.

En otras ocasiones, puede que no conozcamos de antemano cuántas
iteraciones se han de realizar.

La condición del `while` puede que tenga un aspecto diferente
pero, en el cuerpo del bucle, deberá seguir existiendo algo que
modifique el resultado de evaluar la condición.

Ejemplo

Sumar una serie de números
hasta que el usuario introduzca un cero

```
import javax.swing.JOptionPane;
public class While3
{
    public static void main( String args[] )
    {
        float    valor;
        float    suma;

        suma = 0;
        valor = leerValor();

        while (valor!=0) {
            suma += valor;
            valor = leerValor();
        }

        mostrarValor("Suma de los datos", suma);
        System.exit(0);
    }

    private static float leerValor ()
    {
        String entrada;

        entrada = JOptionPane.showInputDialog
                  ( "Introduzca un dato:" );

        return Float.parseFloat(entrada);
    }

    private static void mostrarValor
                    (String mensaje, float valor)
    {
        JOptionPane.showMessageDialog
                  ( null, valor, mensaje,
                    JOptionPane.INFORMATION_MESSAGE );
    }
}
```

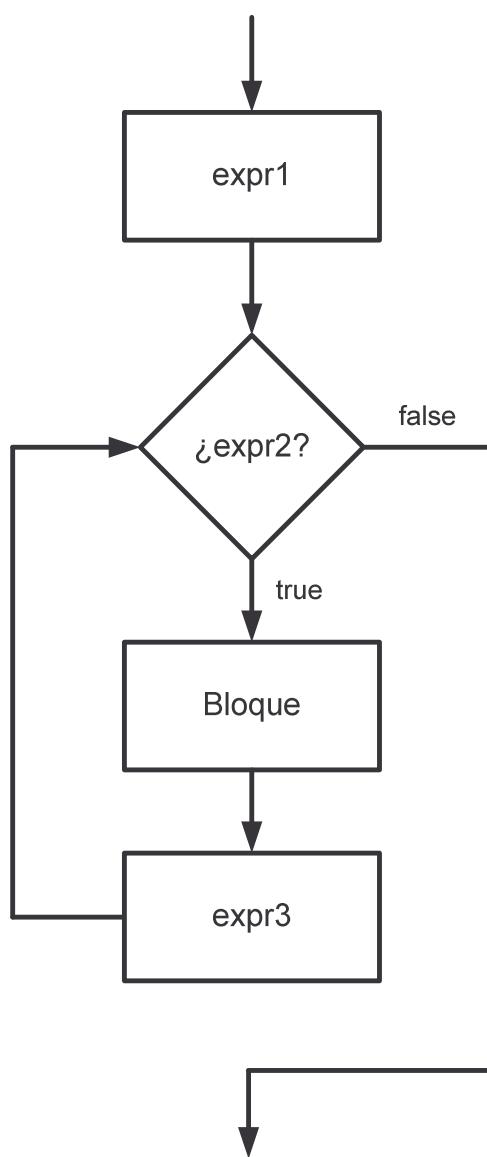
El valor introducido determina en cada iteración
si se termina o no la ejecución del bucle.

El bucle for

Se suele emplear en sustitución del bucle while cuando se conoce el número de iteraciones que hay que realizar.

Sintaxis

```
for (expr1; expr2; expr3) {  
    bloque;  
}
```



Equivalencia entre **for** y **while**

Un fragmento de código como el que aparecía antes con un bucle while:

```
i = 0;  
  
while (i<=10) {  
    System.out.println (n+" x "+i+" = "+(n*i));  
    i++;  
}
```

puede abreviarse si utilizamos un bucle for:

```
for (i=0; i<=10; i++) {  
    System.out.println (n+" x "+i+" = "+(n*i));  
}
```

Como este tipo de estructuras de control es muy común, el lenguaje nos ofrece una forma más compacta de representar un bucle cuando sabemos cuántas veces ha de ejecutarse el cuerpo del bucle.

En general,

```
for (expr1; expr2; expr3) {  
    bloque;  
}
```

equivale a

```
expr1;  
while (expr2) {  
    bloque;  
    expr3;  
}
```

Ejemplo

Cálculo del factorial de un número

Bucle **for**

```
public class FactorialFor
{
    public static void main( String args[ ] )
    {
        long i,n,factorial;

        n = Integer.parseInt( args[0] );

        factorial = 1;

        for (i=1; i<=n; i++) {
            factorial *= i;
        }

        System.out.println ( "f( "+n+" ) = " + factorial );
    }
}
```

Bucle **while**

```
public class FactorialWhile
{
    public static void main( String args[ ] )
    {
        long i,n,factorial;

        n = Integer.parseInt( args[0] );

        factorial = 1;
        i = 1;

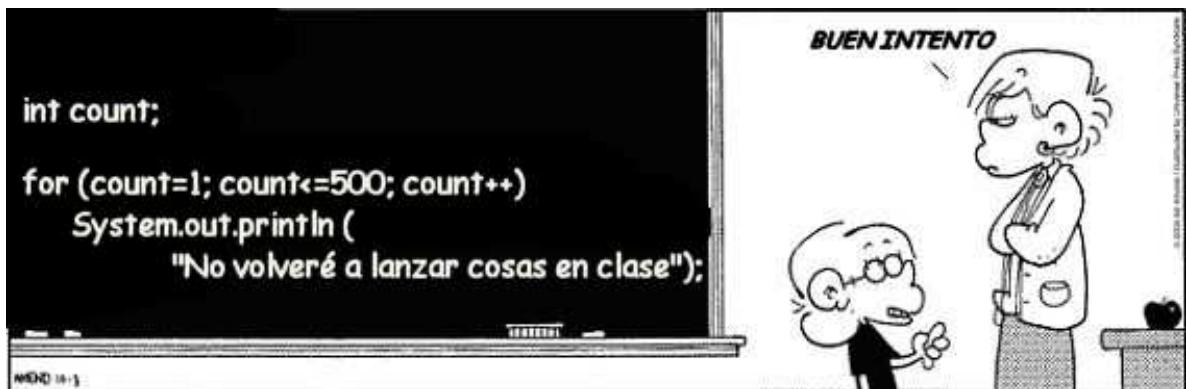
        while (i<=n) {
            factorial *= i;
            i++;
        }

        System.out.println ( "f( "+n+" ) = " + factorial );
    }
}
```

```
for (expr1; expr2; expr3) {  
    bloque;  
}
```

En un bucle for

- ü La primera expresión, `expr1`, suele contener inicializaciones de variables separadas por comas. En particular, siempre aparecerá la inicialización de la variable que hace de contador.
 - Las instrucciones que se encuentran en esta parte del `for` sólo se ejecutarán una vez antes de la primera ejecución del cuerpo del bucle (`bloque`).
- ü La segunda expresión, `expr2`, es la que contiene una expresión booleana (la que aparecería en la condición del bucle `while` equivalente para controlar la ejecución del cuerpo del bucle).
- ü La tercera expresión, `expr3`, contiene las instrucciones, separadas por comas, que se deben ejecutar al finalizar cada iteración del bucle (p.ej. el incremento/decremento de la variable contador).
- ü El bloque de instrucciones `bloque` es el ámbito del bucle (el bloque de instrucciones que se ejecuta en cada iteración).



| Cabecera del bucle | Número de iteraciones |
|--|-----------------------------------|
| for (<i>i</i> =0; <i>i</i> < <i>N</i> ; <i>i</i> ++) | <i>N</i> |
| for (<i>i</i> =0; <i>i</i> <= <i>N</i> ; <i>i</i> ++) | <i>N</i> +1 |
| for (<i>i</i> = <i>k</i> ; <i>i</i> < <i>N</i> ; <i>i</i> ++) | <i>N</i> - <i>k</i> |
| for (<i>i</i> = <i>N</i> ; <i>i</i> >0; <i>i</i> --) | <i>N</i> |
| for (<i>i</i> = <i>N</i> ; <i>i</i> >=0; <i>i</i> --) | <i>N</i> +1 |
| for (<i>i</i> = <i>N</i> ; <i>i</i> > <i>k</i> ; <i>i</i> --) | <i>N</i> - <i>k</i> |
| for (<i>i</i> =0; <i>i</i> < <i>N</i> ; <i>i</i> += <i>k</i>) | <i>N</i> / <i>k</i> |
| for (<i>i</i> = <i>j</i> ; <i>i</i> < <i>N</i> ; <i>i</i> += <i>k</i>) | (<i>N</i> - <i>j</i>)/ <i>k</i> |
| for (<i>i</i> =1; <i>i</i> < <i>N</i> ; <i>i</i> *= <i>2</i>) | $\log_2 N$ |
| for (<i>i</i> =0; <i>i</i> < <i>N</i> ; <i>i</i> *= <i>2</i>) | ∞ |
| for (<i>i</i> = <i>N</i> ; <i>i</i> >0; <i>i</i> /= <i>2</i>) | $\log_2 N + 1$ |

suponiendo que *N* y *k* sean enteros positivos

Bucles infinitos

Un bucle infinito es un bucle que se repite “infinitas” veces:

```
for ( ; )           /*bucle infinito*/
while (true)       /*bucle infinito*/
```

Si nunca deja de cumplirse la condición del bucle,
nuestro programa se quedará indefinidamente
ejecutando el cuerpo del bucle, sin llegar a salir de él.

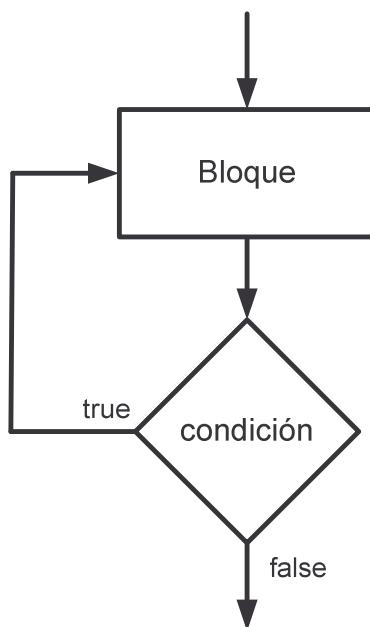
El bucle do while

Tipo de bucle, similar al while, que realiza la comprobación de la condición después de ejecutar el cuerpo del bucle.

Sintaxis

```
do
    sentencia;
while (condición);

do {
    bloque
} while (condición);
```



- El bloque de instrucciones se ejecuta, al menos, una vez.
- El bucle do while resulta especialmente indicado para validar datos de entrada (comprobar que los valores de entrada obtenidos están dentro del rango de valores que el programa espera).

En todos nuestros programas debemos asegurarnos
de que se obtienen datos de entrada válidos
antes de realizar cualquier tipo de operación con ellos.

Ejemplo

Cálculo del factorial

comprobando el valor del dato de entrada

```
import javax.swing.JOptionPane;

public class FactorialDoWhile
{
    public static void main( String args[ ] )
    {
        long n;

        do {
            n = leerEntero(0,20);
        } while ( (n<0) || (n>20) );

        mostrarMensaje ( "f( "+n+" ) = " + factorial(n));
        System.exit(0);
    }

    private static long factorial (long n)
    {
        long i;
        long factorial = 1;

        for (i=1; i<=n; i++) {
            factorial *= i;
        }

        return factorial;
    }

    private static int leerEntero (int min, int max)
    {
        String entrada = JOptionPane.showInputDialog
                        ( "Introduzca un valor entero"
                        + " (entre "+min+" y "+max+"): " );

        return Integer.parseInt(entrada);
    }

    private static void mostrarMensaje (String mensaje)
    {
        JOptionPane.showMessageDialog(null,mensaje);
    }
}
```

Ejemplo

Cálculo de la raíz cuadrada de un número

```
import javax.swing.JOptionPane;

public class Sqrt
{
    public static void main( String args[ ] )
    {
        double n;
        do {
            n = leerReal ( "Introduzca un número positivo" );
        } while (n<0);
        mostrarMensaje( "La raíz cuadrada de "+n
                        + " es aproximadamente "+raiz(n));
        System.exit(0);
    }

    private static double raiz (double n)
    {
        double r;      // Raíz cuadrada del número
        double prev;   // Aproximación previa de la raíz
        r = n/2;
        do {
            prev = r;
            r = (r+n/r)/2;
        } while (Math.abs(r-prev) > 1e-6);
        return r;
    }

    private static double leerReal (String mensaje)
    {
        String entrada;
        entrada = JOptionPane.showInputDialog(mensaje);
        return Double.parseDouble(entrada);
    }

    private static void mostrarMensaje (String mensaje)
    {
        JOptionPane.showMessageDialog(null,mensaje);
    }
}
```

Bucles anidados

Los bucles también se pueden anidar:

```
for (i=0; i<N; i++) {  
    for (j=0; j<N; j++) {  
        printf("( %d, %d ) ", i, j);  
    }  
}
```

genera como resultado:

```
( 0 , 0 ) ( 0 , 1 ) ( 0 , 2 ) ... ( 0 , N )  
( 1 , 0 ) ( 1 , 1 ) ( 1 , 2 ) ... ( 1 , N )  
...  
( N , 0 ) ( N , 1 ) ( N , 2 ) ... ( N , N )
```

Ejemplo

```
int n,i,k;  
  
...  
n = 0; // Paso 1  
for (i=1; i<=2; i++) { // Paso 2  
    for (k=5; k>=1; k-=2) { // Paso 3  
        n = n + i + k; // Paso 4  
    } // Paso 5  
} // Paso 6  
... // Paso 7
```

| Paso | 1 | 2 | 3 | 4 | 5 | 3 | 4 | 5 | 3 | 4 | 5 | 3 | 6 | 2 | 3 | 4 | 5 | 3 | 4 | 5 | 3 | 4 | 5 | 3 | 6 | 2 | 7 |
|------|---|---|---|---|---|---|----|---|----|----|---|---|---|---|---|---|----|---|---|----|---|---|----|---|---|---|---|
| N | 0 | | | 6 | | | 10 | | | 12 | | | | | | | 19 | | | 24 | | | 27 | | | | |
| i | ? | 1 | | | | | | | | | | | | 2 | | | | | | | | | | | | 3 | |
| k | ? | | 5 | | 3 | | 1 | | -1 | | | | | 5 | | 3 | | 1 | | -1 | | | | | | | |

Cuestiones de estilo

Escribimos código para que lo puedan leer otras personas,
no sólo para que lo traduzca el compilador.

Identificadores

- q Los identificadores deben ser **descriptivos**

ß p, i, s...

ü precio, izquierda, suma...

- q En ocasiones, se permite el uso de nombres cortos para variables locales cuyo significado es evidente (p.ej. bucles controlados por contador)

ü for (elemento=0; elemento<N; elemento++) ...

ü for (i=0; i<N; i++) ...

Constantes

- q Se considera una mala costumbre incluir literales de tipo numérico (“**números mágicos**”) en medio del código. Se prefiere la definición de constantes simbólicas (con final).

ß for (i=0; i<79; i++) ...

ü for (i=0; i<columnas-1; i++) ...

Expresiones

q *Expresiones booleanas:*

Es aconsejable escribirlas como se dirían en voz alta.

ß if (!(bloque<actual)) ...

ü if (bloque >= actual) ...

q *Expresiones complejas:*

Es aconsejable dividirlas para mejorar su legibilidad

ß x += (xp = (2*k<(n-m) ? c+k: d-k));

ü if (2*k < n-m)
 xp = c+k;
else
 xp = d-k;

 x += xp;

ü max = (a > b) ? a : b;

Comentarios

q Comentarios descriptivos: Los comentarios deben comunicar algo. Jamás se utilizarán para “parafrasear” el código y repetir lo que es obvio.

ß i++; /* Incrementa el contador */

ü /* Recorrido secuencial de los datos*/

 for (i=0; i<N; i++) ...

Estructuras de control

- ❑ Sangrías:

Conviene utilizar espacios en blanco o separadores para delimitar el ámbito de las estructuras de control de nuestros programas.

- ❑ Líneas en blanco:

Para delimitar claramente los distintos bloques de código en nuestros programas dejaremos líneas en blanco entre ellos.

- ❑ Salvo en la cabecera de los bucles `for`,

sólo incluiremos una sentencia por línea de código.

- ❑ Sean cuales sean las convenciones utilizadas al escribir código (p.ej. uso de sangrías y llaves), hay que ser consistente en su utilización.

```
while (...) {  
    ...  
}
```

```
while (...) {  
    ...  
}
```

```
for (...; ...; ...) {  
    ...  
}
```

```
for (...; ...; ...) {  
    ...  
}
```

```
if (...) {  
    ...  
}
```

```
if (...) {  
    ...  
}
```

El código bien escrito es más fácil de leer, entender y mantener
(además, seguramente tiene menos errores)

Estructuras de control

Relación de ejercicios

1. ¿Cuántas veces se ejecutaría el cuerpo de los siguientes bucles `for`?

`for (i=1; i<10; i++) ...`

`for (i=30; i>1; i-=2) ...`

`for (i=30; i<1; i+=2) ...`

`for (i=0; i<30; i+=4) ...`

2. Ejecute paso a paso el siguiente bucle:

`c = 5;`

`for (a=1; a<5; a++)
c = c - 1;`

3. Escriba un programa que lea una serie de N datos y nos muestre: el número de datos introducidos, la suma de los valores de los datos, la media del conjunto de datos, el máximo, el mínimo, la varianza y la desviación típica.

PISTA: La varianza se puede calcular a partir de la suma de los cuadrados de los datos.

4. Diseñe un programa que lea los coeficientes de una ecuación de segundo grado $ax^2+bx+c=0$ y calcule sus dos soluciones. El programa debe responder de forma adecuada para cualquier caso que se pueda presentar.
5. Diseñe un programa que lea los coeficientes de un sistema de dos ecuaciones lineales con dos incógnitas y calcule su solución. El programa debe responder de forma adecuada cuando el sistema de ecuaciones no sea compatible determinado.

$$\left. \begin{array}{l} ax + by = c \\ dx + ey = f \end{array} \right\}$$

6. Dada una medida de tiempo expresada en horas, minutos y segundos con valores arbitrarios, elabore un programa que transforme dicha medida en una expresión correcta. Por ejemplo, dada la medida $3h\ 118m\ 195s$, el programa deberá obtener como resultado $5h\ 1m\ 15s$. Realice el programa sin utilizar los operadores de división entera (/ y %).
7. Escriba un programa en C que nos calcule el cambio que debe dar la caja de un supermercado: Dado un precio y una cantidad de dinero, el programa nos dirá cuántas monedas deben darse como cambio de tal forma que el número total de monedas sea mínimo. Realice el programa sin utilizar los operadores de división entera (/ y %).
8. Implemente un programa que lea un número decimal y lo muestre en pantalla en hexadecimal (base 16). El cambio de base se realiza mediante divisiones sucesivas por 16 en las cuales los restos determinan los dígitos hexadecimales del número según la siguiente correspondencia:

| Resto | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|--------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| Dígito | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

Por ejemplo:

$$\begin{array}{r}
 65029 \quad |_{16} \\
 5 \quad 4064 \quad |_{16} \\
 \downarrow \qquad \downarrow \\
 0 \quad 254 \quad |_{16} \\
 \downarrow \qquad \downarrow \qquad \downarrow \\
 14 \quad 15 \\
 \downarrow \qquad \downarrow \qquad \downarrow \\
 5 \quad 0 \quad E \quad F \longrightarrow FE05 \\
 \end{array}$$

$65029_{10} = FE05_{16}$

9. Escriba una función (un método) que obtenga la letra del DNI a partir del número. Para ello debe obtener el resto de dividir el número entre 23. La letra asociada al número vendrá dada por este resto en función de la siguiente tabla:

| | | | | | |
|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| $0 \rightarrow T$ | $1 \rightarrow R$ | $2 \rightarrow W$ | $3 \rightarrow A$ | $4 \rightarrow G$ | $5 \rightarrow M$ |
| $6 \rightarrow Y$ | $7 \rightarrow F$ | $8 \rightarrow P$ | $9 \rightarrow D$ | $10 \rightarrow X$ | $11 \rightarrow B$ |
| $12 \rightarrow N$ | $13 \rightarrow J$ | $14 \rightarrow Z$ | $15 \rightarrow S$ | $16 \rightarrow Q$ | $17 \rightarrow V$ |
| $18 \rightarrow H$ | $19 \rightarrow L$ | $20 \rightarrow C$ | $21 \rightarrow K$ | $22 \rightarrow E$ | $23 \rightarrow T$ |

10. Escriba una función que, a partir de los dígitos de un ISBN, calcule el carácter de control con el que termina todo ISBN. Para calcular el carácter de control, debe multiplicar cada dígito por su posición (siendo el dígito de la izquierda el que ocupa la posición 1), sumar los resultados obtenidos y hallar el resto de dividir por 11. El resultado será el carácter de control, teniendo en cuenta que el carácter de control es ‘X’ cuando el resto vale 10.

11. Implemente un programa que calcule la suma de los 100 primeros términos de las siguientes sucesiones:

$$a_n = a_{n-1} + n$$

$$a_n = \frac{a_{n-1}}{n}$$

$$a_n = (-1)^n \frac{n^2 - 1}{2n + 1}$$

12. Realice un programa que calcule los valores de la función:

$$f(x, y) = \frac{\sqrt{x}}{y^2 - 1}$$

para los valores de (x, y) con $x = -50, -48 \dots 0 \dots 48, 50$ e $y = -40, -39 \dots 0 \dots 39, 40$

13. Implemente funciones que nos permitan calcular x^n , $n!$ y $\binom{n}{m}$

14. Escriba un programa que muestre en pantalla todos los números primos entre 1 y n , donde n es un número positivo que recibe el programa como parámetro.

15. Escriba una función que, dados dos números enteros, nos diga si cualquiera de ellos divide o no al otro.

16. Implemente un programa que calcule los divisores de un número entero.

17. Implemente una función que nos devuelva el máximo común divisor de dos números enteros.

18. Implemente una función que nos devuelva el mínimo común múltiplo de dos números enteros.

19. Diseñe e implemente un programa que realice la descomposición en números primos de un número entero.

20. Escriba un programa que lea números enteros hasta que se introduzcan 10 números o se introduzca un valor negativo. El programa mostrará entonces el valor medio de los números introducidos (sin contar el número negativo en caso de que éste se haya indicado).

21. Implemente una función que nos diga si un número ha conseguido o no el reintegro en el sorteo de la ONCE. Un número de cinco cifras consigue el reintegro si su primera o última cifra coincide con la primera o última cifra del número agraciado en el sorteo.

22. Diseñe un programa para jugar a adivinar un número entre 0 y 100. El programa irá dando pistas al jugador indicándole si el número introducido por el jugador es menor o mayor que el número que tiene que adivinar. El juego termina cuando el jugador adivina el número o decide terminar de jugar (por ejemplo, escribiendo un número negativo).

23. Amplíe el programa del ejercicio anterior permitiendo que el jugador juegue tantas veces como desee. El programa deberá mantener las estadísticas del jugador y mostrárselas al final de cada partida (número medio de intentos para adivinar el número, número de veces que el jugador abandona, mejor partida y peor partida).

24. Aplicar el método de Newton-Raphson a los siguientes problemas:

- a. Calcular la raíz cuadrada de un número.
- b. Calcular la raíz cúbica de un número.
- c. Calcular la raíz n-ésima de un número

El método de Newton-Raphson es un método general para la obtención de los ceros de una función. Para ello se van generando los términos de la sucesión

$$x_{n+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

donde $f(x)$ es la función cuyo cero deseamos obtener y $f'(x)$ es la derivada de la función. Por ejemplo, para calcular la raíz cuadrada de un número n , hemos de obtener un cero de la función $f(x) = x^2 - n$

Partiendo de un valor inicial x_0 cualquiera, se van generando términos de la sucesión hasta que la diferencia entre dos términos consecutivos de la sucesión sea inferior a una precisión especificada de antemano (p.ej. 10^{-6}) .

25. Uso y manipulación de fechas:

- a. Diseñe una clase para representar fechas.
- b. Escriba un método estático que nos diga el número de días de un mes (¡ojo con los años bisiestos!).
- c. Añada a su clase un método que nos indique el número de días del mes al que pertenece la fecha.
- d. Incluya, en su clase `Fecha`, un método que nos diga el número de días que hay desde una fecha determinada hasta otra.
- e. Implemente un método que nos diga el día de la semana correspondiente a una fecha concreta (p.ej. el 1 de diciembre de 2004 fue miércoles).
- f. Escriba un programa que muestre el calendario de un mes concreto.

NOTA: Compruebe el correcto funcionamiento de todos los programas con varios valores para sus entradas. En el caso de los ejercicios en que se pide la creación de un método o función, escriba programas auxiliares que hagan uso del método creado.

Vectores y matrices

Arrays

Declaración

Creación

Acceso a los elementos de un array

Manipulación de vectores y matrices

Algoritmos de ordenación

Ordenación por selección

Ordenación por inserción

Ordenación por intercambio directo (método de la burbuja)

Ordenación rápida (QuickSort)

Algoritmos de búsqueda

Búsqueda lineal

Búsqueda binaria

Apéndice: Cadenas de caracteres

Arrays

Un array es una estructura de datos que contiene una colección de datos del mismo tipo

Ejemplos

Temperaturas mínimas de los últimos treinta días

Valor de las acciones de una empresa durante la última semana

...

Propiedades de los arrays

- Los arrays se utilizan como **contenedores** para almacenar datos relacionados (en vez de declarar variables por separado para cada uno de los elementos del array).
- Todos los **datos** incluidos en el array son **del mismo tipo**. Se pueden crear arrays de enteros de tipo `int` o de reales de tipo `float`, pero en un mismo array no se pueden mezclar datos de tipo `int` y datos de tipo `float`.
- El tamaño del array se establece cuando se crea el array (con el operador `new`, igual que cualquier otro objeto).
- A los elementos del array se accederá a través de la posición que ocupan dentro del conjunto de elementos del array.

Terminología

Los arrays unidimensionales se conocen con el nombre de **vectores**.

Los arrays bidimensionales se conocen con el nombre de **matrices**.

Declaración

Para declarar un array,
se utilizan corchetes para indicar que se trata de un array
y no de una simple variable del tipo especificado.

Vector (array unidimensional):

```
tipo identificador[ ];
```

o bien

```
tipo[] identificador;
```

donde

tipo es el tipo de dato de los elementos del vector

identificador es el identificador de la variable.

Matriz (array bidimensional):

```
tipo identificador[ ][ ];
```

o bien

```
tipo[][] identificador;
```

NOTA: No es una buena idea que el identificador del array
termine en un dígito, p.ej. vector3

Creación

Los arrays se crean con el operador new.

Vector (array unidimensional):

```
vector = new tipo[elementos];
```

Entre corchetes se indica el tamaño del vector.

tipo debe coincidir con el tipo con el que se haya declarado el vector.

vector debe ser una variable declarada como tipo[]

Ejemplos

```
float[] notas = new float[ALUMNOS];
```

```
int[] temperaturas = new int[7];
```

Matriz (array bidimensional):

```
matriz = new tipo[filas][columnas];
```

Ejemplo

```
int[][] temperaturas = new int[12][31];
```

Vso

Para acceder a los elementos de un array,
utilizamos índices
(para indicar la posición del elemento dentro del array)

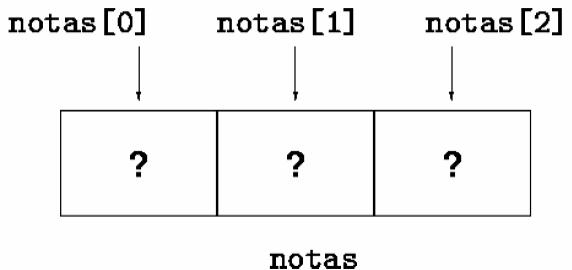
Vector (array unidimensional):

```
vector[índice]
```

- En Java, el índice de la primera componente de un vector es siempre 0.
- El tamaño del array puede obtenerse utilizando la propiedad `vector.length`
- Por tanto, el índice de la última componente es `vector.length-1`

Ejemplo

```
float[] notas = new float[3];
```



Matriz (array bidimensional):

```
matriz[índice1][índice2]
```

Una matriz, en realidad, es un vector de vectores:

- En Java, el índice de la primera componente de un vector es siempre 0, por lo que `matriz[0][0]` será el primer elemento de la matriz.
- El tamaño del array puede obtenerse utilizando la propiedad `array.length`:
 - `matriz.length` nos da el número de filas
 - `matriz[0].length` nos da el número de columnas
- Por tanto, el último elemento de la matriz es `matriz[matriz.length-1][matriz[0].length-1]`

Inicialización en la declaración

Podemos asignarle un valor inicial a los elementos de un array en la propia declaración

```
int vector[ ] = {1, 2, 3, 5, 7};  
int matriz[][] = { {1,2,3}, {4,5,6} };
```

El compilador deduce automáticamente las dimensiones del array.

Manipulación de vectores y matrices

Las operaciones se realizan componente a componente

Ejemplo: Suma de los elementos de un vector

```
static float media (float datos[])
{
    int    i;
    int    n = datos.length;
    float suma = 0;

    for (i=0; i<n; i++)
        suma = suma + datos[i];

    return suma/n;
}
```

No es necesario utilizar todos los elementos de un vector, por lo que, al trabajar con ellos, se puede utilizar una variable entera adicional que nos indique el número de datos que realmente estamos utilizando:

El tamaño del vector nos dice cuánta memoria se ha reservado para almacenar datos del mismo tipo, no cuántos datos del mismo tipo tenemos realmente en el vector.

Ejemplo: Suma de los n primeros elementos de un vector

```
static float media (float datos[], int n)
{
    int    i;
    float suma = 0;

    for (i=0; i<n; i++)
        suma = suma + datos[i];

    return suma/n;
}
```

Ejemplo

```
public class Vectores
{
    public static void main (String[] args)
    {
        int pares[ ] = { 2, 4, 6, 8, 10 } ;
        int impares[ ] = { 1, 3, 5, 7, 9 } ;

        mostrarVector(pares);
        System.out.println("MEDIA=" + media(pares)) ;

        mostrarVector(impares);
        System.out.println("MEDIA=" + media(impares));
    }

    static void mostrarVector (int datos[])
    {
        int i;

        for (i=0; i<datos.length; i++)
            System.out.println(datos[i]);
    }

    static float media (int datos[])
    {
        int i;
        int n = datos.length;
        int suma = 0;

        for (i=0; i<n; i++)
            suma = suma + datos[i];

        return suma/n;
    }
}
```

```

static int[] leerVector (int datos)
{
    int    i;
    int[] vector = new int[datos];

    for (i=0; i<datos; i++)
        vector[i] = leerValor();

    return vector;
}

```

IMPORTANTE:

Cuando se pasa un array como parámetro,
se copia una referencia al array y no el conjunto de valores en sí.

Por tanto, tenemos que tener cuidado con los efectos colaterales
que se producen si, dentro de un módulo,
modificamos un vector que recibimos como parámetro.

Ejemplo

El siguiente método lee los elementos de un vector ya creado

```

static void leerVector (int[] datos)
{
    int    i;

    for (i=0; i<datos.length; i++)
        datos[i] = leerValor();
}

```

Copia de arrays

La siguiente asignación sólo copia las referencias, no crea un nuevo array:

```
int[] datos = pares;
```

Para copiar los elementos de un array, hemos de crear un nuevo array y copiar los elementos uno a uno

```
int[] datos = new int[pares.length];  
  
for (i=0; i<pares.length; i++)  
    datos[i] = pares[i]
```

También podemos utilizar una función predefinida en la biblioteca de clases estándar de Java:

```
System.arraycopy(from, fromIndex, to, toIndex, n);
```

```
int[] datos = new int[pares.length];  
System.arraycopy(pares, 0, datos, 0, pares.length);
```

EXTRA:

La biblioteca de clases de Java incluye una clase auxiliar llamada **java.util.Arrays** que incluye como métodos algunas de las tareas que se realizan más a menudo con vectores:

- `Arrays.sort(v)` ordena los elementos del vector.
- `Arrays.equals(v1, v2)` comprueba si dos vectores son iguales.
- `Arrays.fill(v, val)` rellena el vector `v` con el valor `val`.
- `Arrays.toString(v)` devuelve una cadena que representa el contenido del vector.
- `Arrays.binarySearch(v, k)` busca el valor `k` dentro del vector `v` (que previamente ha de estar ordenado).

Ejemplos

Un programa que muestra los parámetros que le indicamos en la línea de comandos:

```
public class Eco
{
    public static void main(String args[])
    {
        int i;

        for (i=0; i<args.length; i++)
            System.out.println(args[i]);
    }
}
```

Un método que muestra el contenido de una matriz:

```
public static void mostrarMatriz (double matriz[][][])
{
    int i,j;
    int filas = matriz.length;
    int columnas = matriz[0].length;

    // Recorrido de las filas de la matriz

    for (i=0; i<filas; i++) {

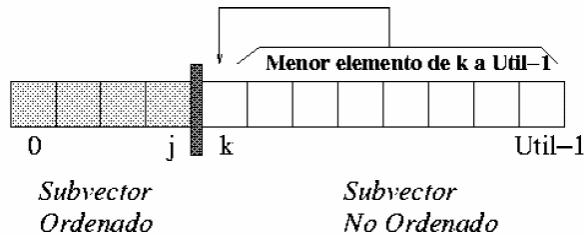
        // Recorrido de las celdas de una fila

        for (j=0; j<columnas; j++) {

            System.out.println ( "matriz["+i+"]["+j+"]=" +
                                + matriz[i][j] );
        }
    }
}
```

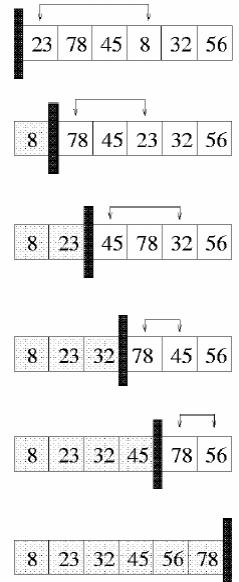
Algoritmos de ordenación

Ordenación por selección



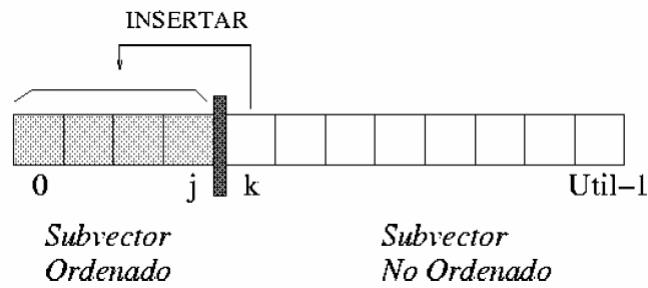
```
static void ordenarSeleccion (double v[])
{
    double tmp;
    int i, j, pos_min;
    int N = v.length;

    for (i=0; i<N-1; i++) {
        // Menor elemento del vector v[i..N-1]
        pos_min = i;
        for (j=i+1; j<N; j++)
            if (v[j]<v[pos_min])
                pos_min = j;
        // Coloca el mínimo en v[i]
        tmp = v[i];
        v[i] = v[pos_min];
        v[pos_min] = tmp;
    }
}
```



En cada iteración, se selecciona el menor elemento del subvector no ordenado y se intercambia con el primer elemento de este subvector.

Ordenación por inserción



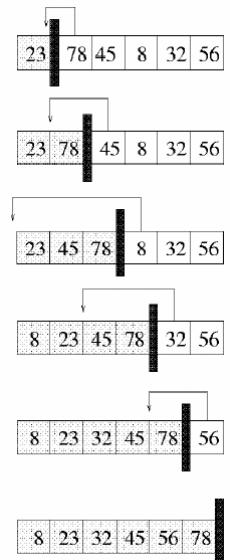
```
static void ordenarInsercion (double v[])
{
    double tmp;
    int i, j;
    int N = v.length;

    for (i=1; i<N; i++) {

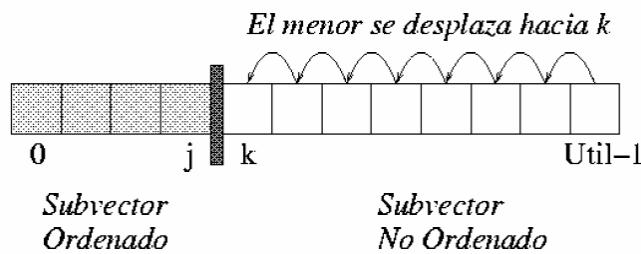
        tmp = v[i];

        for (j=i; (j>0) && (tmp<v[j-1]); j--)
            v[j] = v[j-1];

        v[j] = tmp;
    }
}
```



Ordenación por intercambio directo (método de la burbuja)



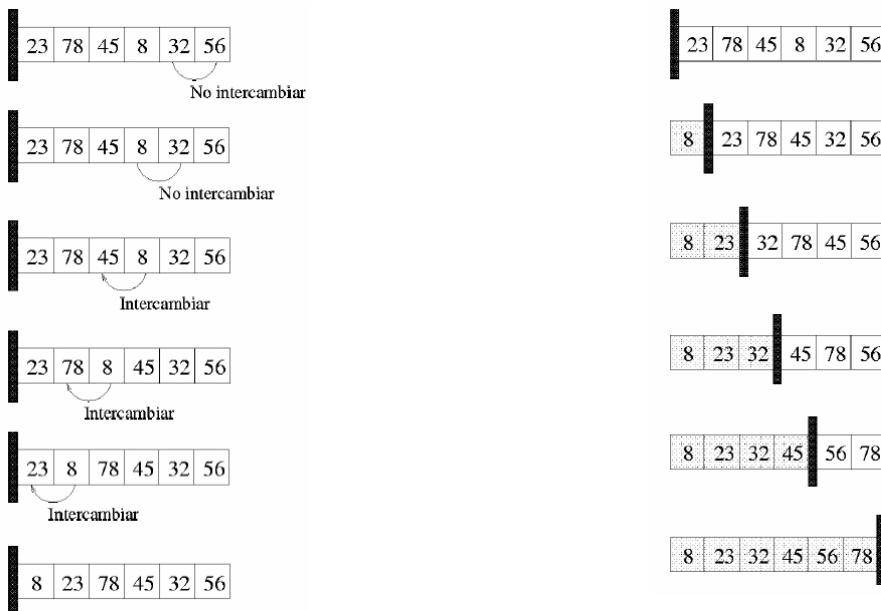
```

static void ordenarBurbuja (double v[ ] )
{
    double tmp;
    int i, j;
    int N = v.length;

    for (i=1; i<N; i++)
        for (j=N-1; j>=i; j--)
            if (v[ j ] < v[ j-1 ] ) {
                tmp = v[ j ];
                v[ j ] = v[ j-1 ];
                v[ j-1 ] = tmp;
            }
}
    
```

En cada iteración

Estado del vector
tras cada iteración:



Ordenación rápida (QuickSort)

1. Se toma un elemento arbitrario del vector, al que denominaremos pivote (p).
2. Se divide el vector de tal forma que todos los elementos a la izquierda del pivote sean menores que él, mientras que los que quedan a la derecha son mayores que él.
3. Ordenamos, por separado, las dos zonas delimitadas por el pivote.

```
static void quicksort
    (double v[], int izda, int dcha)
{
    int pivote; // Posición del pivote

    if (izda<dcha) {

        pivote = partir (v, izda, dcha);

        quicksort (v, izda, pivote-1);

        quicksort (v, pivote+1, dcha);
    }
}
```

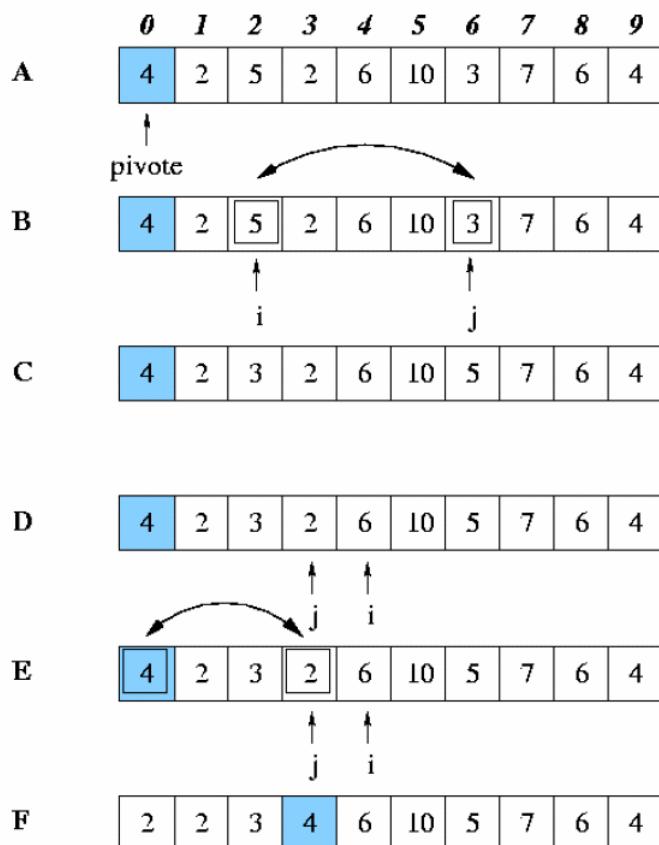
Uso:

```
quicksort (vector, 0, vector.length-1);
```

Obtención del pivote

Mientras queden elementos mal colocados respecto al pivote:

- Se recorre el vector, de izquierda a derecha, hasta encontrar un elemento situado en una posición i tal que $v[i] > p$.
- Se recorre el vector, de derecha a izquierda, hasta encontrar otro elemento situado en una posición j tal que $v[j] < p$.
- Se intercambian los elementos situados en las casillas i y j (de modo que, ahora, $v[i] < p < v[j]$).



```

/**
 * División del vector en dos partes
 * @see quicksort
 *
 * @param primero Índice del primer elemento
 * @param ultimo Índice del último elemento
 * @return Posición del pivote
 *
 * @pre (primero>=0)
 *      && (primero<=ultimo)
 *      && (ultimo<v.length)
 */

private static int partir
    (double v[], int primero, int ultimo)
{
    double pivote = v[primero]; // Valor del pivote
    double temporal;           // Variable auxiliar
    int izda = primero+1;
    int dcha = ultimo;

    do { // Pivotear...

        while ((izda<=dcha) && (v[izda]<=pivote))
            izda++;

        while ((izda<=dcha) && (v[dcha]>pivote))
            dcha--;

        if (izda < dcha) {
            temporal = v[izda];
            v[izda] = v[dcha];
            v[dcha] = temporal;
            dcha--;
            izda++;
        }
    } while (izda <= dcha);

    // Colocar el pivote en su sitio
    temporal = v[primero];
    v[primero] = v[dcha];
    v[dcha] = temporal;

    return dcha; // Posición del pivote
}

```

Algoritmos de búsqueda

Búsqueda lineal = Búsqueda secuencial

```
// Búsqueda lineal de un elemento en un vector
// - Devuelve la posición de "dato" en el vector
// - Si "dato" no está en el vector, devuelve -1

static int buscar (double vector[], double dato)
{
    int i;
    int N = vector.length;
    int pos = -1;

    for (i=0; i<N; i++)
        if (vector[i]==dato)
            pos = i;

    return pos;
}
```

Versión mejorada

```
// Búsqueda lineal de un elemento en un vector
// - Devuelve la posición de "dato" en el vector
// - Si "dato" no está en el vector, devuelve -1

static int buscar (double vector[], double dato)
{
    int i;
    int N = vector.length;
    int pos = -1;

    for (i=0; (i<N) && (pos== -1); i++)
        if (vector[i]==dato)
            pos = i;

    return pos;
}
```

Búsqueda binaria

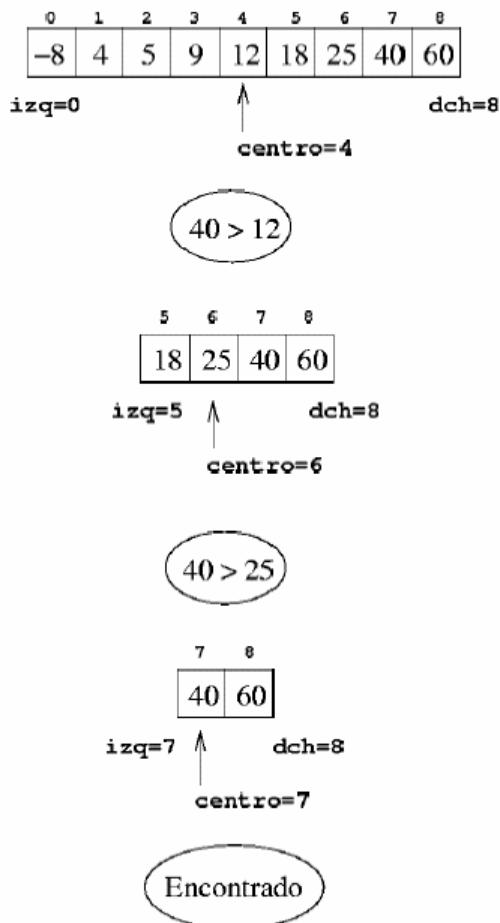
Precondición

El vector ha de estar ordenado

Algoritmo

Se compara el dato buscado con el elemento en el centro del vector:

- Si coinciden, hemos encontrado el dato buscado.
- Si el dato es mayor que el elemento central del vector, tenemos que buscar el dato en segunda mitad del vector.
- Si el dato es menor que el elemento central del vector, tenemos que buscar el dato en la primera mitad del vector.



```

// Búsqueda binaria de un elemento en un vector
// - Devuelve la posición de "dato" en el vector
// - Si "dato" no está en el vector, devuelve -1

// Implementación recursiva
// Uso: binSearch(vector,0,vector.length-1,dato)

static int binSearch
    (double v[], int izq, int der, double buscado)
{
    int centro = (izq+der)/2;

    if (izq>der)
        return -1;
    else if (buscado==v[centro])
        return centro;
    else if (buscado<v[centro])
        return binSearch(v, izq, centro-1, buscado);
    else
        return binSearch(v, centro+1, der, buscado);
}

// Implementación iterativa
// Uso: binSearch (vector, dato)

static int binSearch (double v[], double buscado)
{
    int izq = 0;
    int der = v.length-1;
    int centro = (izq+der)/2;

    while ((izq<=der) && (v[centro]!=buscado)) {
        if (buscado<v[centro])
            der = centro - 1;
        else
            izq = centro + 1;

        centro = (izq+der)/2;
    }

    if (izq>der)
        return -1;
    else
        return centro;
}

```

Apéndice: Cadenas de caracteres

Una cadena de caracteres no es más que un vector de caracteres.

La clase `java.lang.String`, que se emplea para representar cadenas de caracteres en Java, incluye distintos métodos que nos facilitan algunas de las operaciones que se suelen realizar con cadenas de caracteres:

- El método `substring` nos permite obtener una subcadena:

```
String java="Java";
String s = java.substring(0,3);
System.out.println(s); // Jav
```

- El método `charAt(n)` nos devuelve el carácter que se encuentra en la posición n de la cadena:

```
String java="Java";
char c = java.charAt(2);
System.out.println(c); // v
```

- El método `indexOf(s)` nos devuelve la posición de una subcadena dentro de la cadena:

```
String java="Java";
int p = java.indexOf("av");
System.out.println(p); // 1
```

- El método `replace(old,new)` reemplaza subcadenas:

```
String java="Java";
java.replace("ava","ini");
System.out.println(java); // Jini
```

- El método `equals(s)` se usa para comprobar si dos cadenas son iguales:

```
if (s.equals("Hola")) {  
    ...  
}
```

RECORDATORIO: el operador `==` no debe utilizarse para comparar objetos.

- El método `startsWith(s)` nos dice si una cadena empieza con un prefijo determinado:

```
if (s.startsWith("get")) {  
    ...  
}
```

- El método `endsWith(s)` nos dice si una cadena termina con un sufijo determinado:

```
if (s.endsWith(".html")) {  
    ...  
}
```

- El método `length()` devuelve la longitud de la cadena.

...

La clase `java.lang.String` incluye decenas de métodos.

La lista completa de métodos y los detalles de utilización de cada método se pueden consultar en la ayuda del JDK.

Vectores y matrices

Relación de ejercicios

1. Dado un vector de números reales:
 - a. Escriba un método `max` que nos devuelva el máximo de los valores incluidos en el vector.
 - b. Escriba un método `min` que nos devuelva el mínimo de los valores incluidos en el vector.
 - c. Escriba un método `media` que nos devuelva la media de los valores incluidos en el vector.
 - d. Escriba un método `varianza` que nos devuelva la varianza de los valores incluidos en el vector.
 - e. Escriba un método `mediana` que nos devuelva la mediana de los valores incluidos en el vector.
 - f. Escriba un método `moda` que nos devuelva la moda de los valores incluidos en el vector
 - g. Escriba un método `percentil(n)` que nos devuelva el valor correspondiente al percentil n en el conjunto de valores del vector.
2. Implemente una clase en Java, llamada `Serie`, que encapsule un vector de números reales e incluya métodos (no estáticos) que nos permitan calcular todos los valores mencionados en el ejercicio anterior a partir de los datos encapsulados por un objeto de tipo `Serie`.
3. Dado un vector de números reales, escriba un método que nos devuelva el máximo y el mínimo de los valores incluidos en el vector.
4. Dado un vector, implemente un método que inserte un elemento en una posición dada del vector.

NOTA: Insertar un elemento en el vector desplaza una posición hacia la derecha a los elementos del vector que han de quedar detrás del elemento insertado. Además, la inserción ocasiona la “desaparición” del último elemento del vector.

5. Implemente un método llamado secuencia que realice la búsqueda de la secuencia en orden creciente más larga dentro de un vector de enteros. El método ha de devolver tanto la posición de la primera componente de la secuencia como el tamaño de la misma.
6. Una cadena de ADN se representa como una secuencia circular de bases (adenina, timina, citosina y guanina) que es única para cada ser vivo, por ejemplo:

| | | |
|---|---|---|
| A | T | G |
| T | | C |
| A | T | G |

Dicha cadena se puede representar como un vector de caracteres recorriéndola en sentido horario desde la parte superior izquierda:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | T | G | C | G | T | A | T |
|---|---|---|---|---|---|---|---|

Se pide diseñar una clase que represente una secuencia de ADN e incluya un método booleano que nos devuelva `true` si dos cadenas de ADN coinciden.

MUY IMPORTANTE: La secuencia de ADN es cíclica, por lo que puede comenzar en cualquier posición. Por ejemplo, las dos secuencias siguientes coinciden:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | T | G | C | G | T | A | T |
| A | T | A | T | G | C | G | T |

7. Dado un vector de números reales, escriba un método que ordene los elementos del vector **de mayor a menor**.
8. Dado un vector de números reales, escriba un método que ordene los elementos del vector de tal forma que los números pares aparezcan antes que los números impares. Además, los números pares deberán estar ordenados de forma ascendente, mientras que los números impares deberán estar ordenados de forma descendente. Esto es, el vector {1,2,3,4,5,6} quedará como {2,4,6,5,3,1}.
9. Crear una clase `Matriz` para manipular matrices que encapsule un array bidimensional de números reales.
 - a. Incluya en la clase métodos que nos permitan acceder y modificar de forma segura los elementos de la matriz (esto es, las variables de instancia deben ser privadas y los métodos han de comprobar la validez de sus parámetros).
 - b. Escriba un método que nos permita sumar matrices.
 - c. Implemente un método que nos permita multiplicar matrices.
 - d. Cree un método con el que se obtenga la traspuesta de una matriz.

10. En Java, para generar números pseudoaleatorios, se puede utilizar la función `Math.random()` definida en la clase `java.lang.Math`. Dicha función genera una secuencia de números pseudoaleatorios que se supone sigue una distribución uniforme (esto es, todos los valores aparecerán con la misma probabilidad). Escriba un programa que compruebe si el generador de números pseudoaleatorios de Java genera realmente números aleatorios con una distribución uniforme.

Sugerencia: Genere un gran número de números aleatorios (entre 0 y 100, por ejemplo) y compruebe que la distribución resultante del número de veces que aparece cada número es (más o menos) uniforme. Por ejemplo, mida la dispersión de la distribución resultante utilizando una medida como la varianza (y reutilice la clase `Serie` del ejercicio 2).

11. Realizar una simulación de Monte Carlo para aproximar el valor del área bajo una curva $f(x)$, en $x \in [a, b]$.

Algoritmo: Generar puntos aleatorios en el rectángulo de extremos $(a, 0)$ y (b, m) , con $m = \max_{a \leq x \leq b} f(x)$, y contar el número de puntos que caen por debajo de la curva.

NOTA: Este método permite generar números pseudoaleatorios que sigan cualquier distribución que nosotros deseemos. Por ejemplo, probar con una distribución normal.

12. Crear un programa modular para jugar a las **7 y media**. Se trata de un juego de cartas (con baraja española) en el que el objetivo es alcanzar una puntuación de 7.5. Cada carta del 1 al 7 tiene su valor nominal y cada figura (sota, caballo y rey) vale 0.5 puntos.

NOTA: Para barajar, mezcle los elementos de un vector de cartas intercambiando en repetidas ocasiones cartas elegidas al azar con la ayuda de la función `Math.random()`

Recursividad

Preliminares

Uso de memoria en tiempo de ejecución
Demostraciones por inducción

Concepto de recursividad

Utilidad
Funcionamiento de un algoritmo recursivo
Diseño de algoritmos recursivos
Recursividad frente a iteración

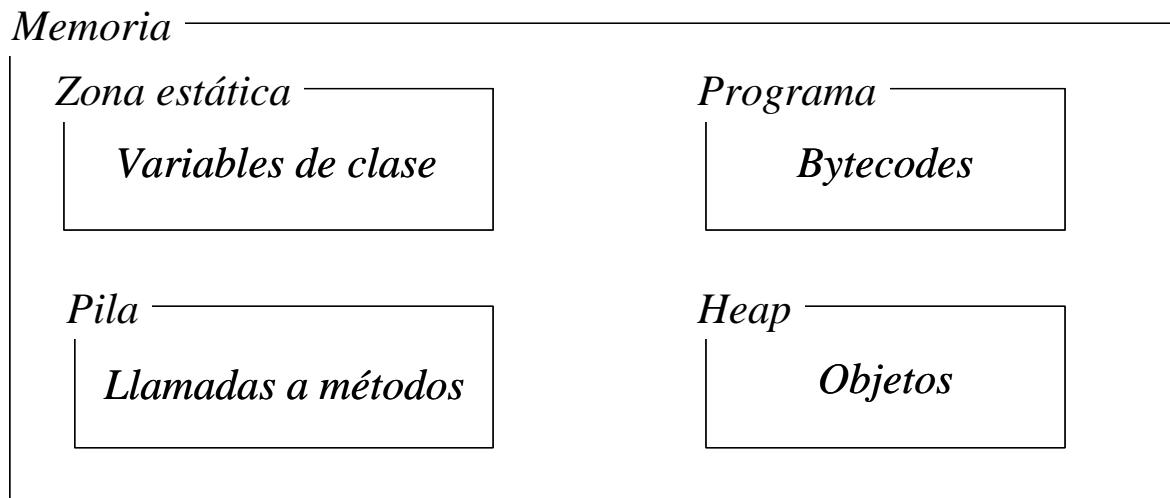
Ejemplos

Sucesión de Fibonacci
Combinaciones
Las torres de Hanoi

Preliminares

Uso de memoria en tiempo de ejecución

Al ejecutar una aplicación Java,
la memoria usada por la aplicación se divide en distintas zonas:



Programa

En una zona de memoria se cargan los bytecodes correspondientes a las clases que forman parte de la aplicación.

NOTA: Para que se puedan encontrar los bytecodes correspondientes a las distintas clases, los ficheros .class deben estar en un directorio o en un fichero (.zip o .jar) que figure en la variable de entorno CLASSPATH.

Zona estática

Donde se almacenan las variables de clase (declaradas con static)

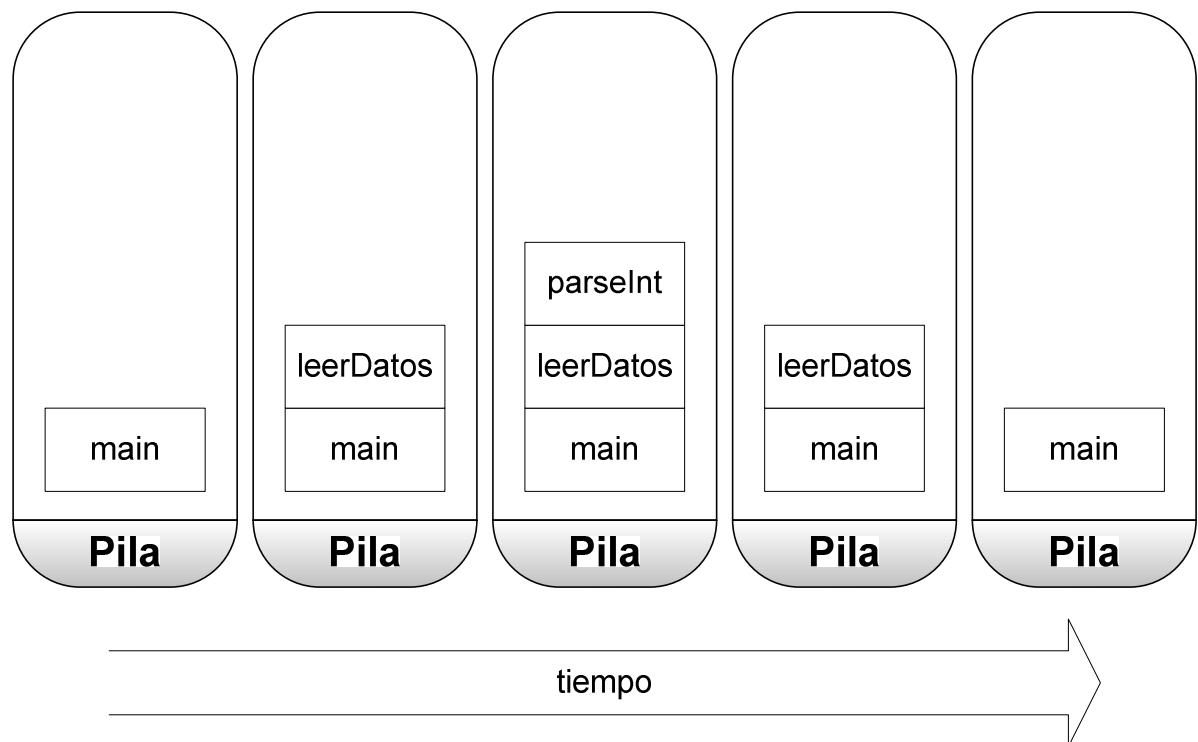
- Datos compartidos (datos globales).
- Datos que han de mantenerse más allá de la duración de la invocación a un método o de la vida de un objeto.

Pila

Donde se almacenan las variables locales (y parámetros) de los métodos que se invocan.

- Cada llamada a un método provoca que se reserve espacio en la pila para almacenar sus variables locales (y los valores de sus parámetros).
- Al finalizar la ejecución del método, se libera el espacio ocupado en la pila por las variables locales del método.

Esta zona de memoria se denomina pila por la forma en que evoluciona su estado:



Inducción

Una estrategia matemática de demostración.

Ejemplo

Sea $s : \text{int} \rightarrow \text{int}$ una función
que nos da la suma de los n primeros números naturales

De forma iterativa, podemos escribir

$$S(n) = 1 + 2 + 3 + \dots + n$$

La expresión anterior es equivalente a $S'(n) = \frac{n(n+1)}{2}$

¿Cómo demostramos que, para cualquier valor de n ,
la expresión anterior es correcta?

1. Vemos que, para $n=1$,
la expresión anterior es válida: $S(1) = 1 = S'(1)$
2. Suponemos que la expresión funciona bien para $n=k$
y comprobamos si también es válida para $n=k+1$

$$S(k+1) = S(k) + (k+1) = \frac{k(k+1)}{2} + (k+1) = \frac{k^2 + 3k + 2}{2} = \frac{(k+1)(k+2)}{2} = S'(k+1)$$

Por tanto, por inducción, $S(n)=S'(n)$ para todo $n \geq 1$

Para realizar la demostración, escribimos el término $S(k+1)$
en función del término anterior $S(k)$ de **forma recursiva**.

Para poder utilizar esta definición recursiva de la función,
lo único que necesitamos es un **caso base** (en este caso, $S(1)=1$)

¿Cómo se evalúa una función recursiva?

$$\begin{aligned} S(4) &= S(3) + 4 = (S(2) + 3) + 4 = ((S(1) + 2) + 3) + 4 \\ &= (((1) + 2) + 3) + 4 = ((3) + 3) + 4 = (6) + 4 = 10 \end{aligned}$$

Recursividad

Una función que se llama a sí misma se denomina **recursiva**

$$n! = \left\{ \begin{array}{ll} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{array} \right|$$

$$x^n = \left\{ \begin{array}{ll} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{array} \right|$$

Definición recursiva: Véase *definición recursiva*.
[FOLDOC: Free On-line Dictionary of Computing]

Utilidad

Cuando la solución de un problema se puede expresar en términos de la resolución de un problema de la misma naturaleza, aunque de menor complejidad.

**Divide y vencerás:
Un problema complejo se divide en otros problemas más sencillos
(del mismo tipo)**

Sólo tenemos que conocer la solución no recursiva para algún caso sencillo (denominado caso base) y hacer que la división de nuestro problema acabe recurriendo a los casos base que hayamos definido.

Como en las demostraciones por inducción, podemos considerar que “tenemos resuelto” el problema más simple para resolver el problema más complejo (sin tener que definir la secuencia exacta de pasos necesarios para resolver el problema).

Ejemplo

¿Cuántas formas hay de colocar n objetos en orden?

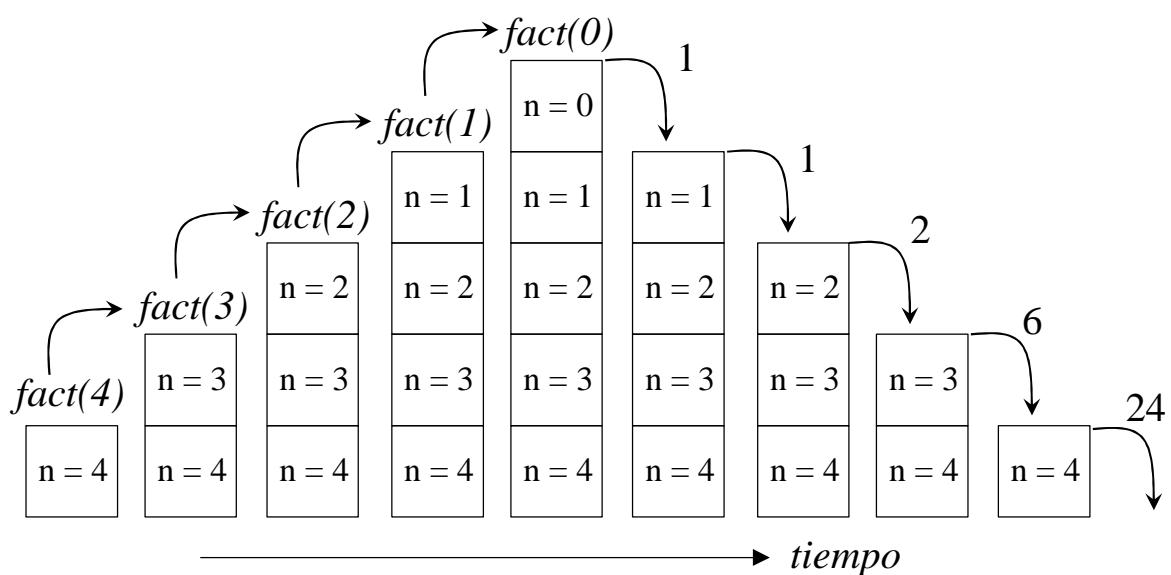
- Podemos colocar cualquiera de los n objetos en la primera posición.
- A continuación, colocamos los (n-1) objetos restantes.
- Por tanto: $P(n) = n P(n-1) = n!$

Factorial calculado de forma recursiva:

```
static int factorial (int n)
{
    int resultado;

    if (n==0)                                // Caso base
        resultado = 1;
    else                                       // Caso general
        resultado = n*factorial(n-1);

    return resultado;
}
```

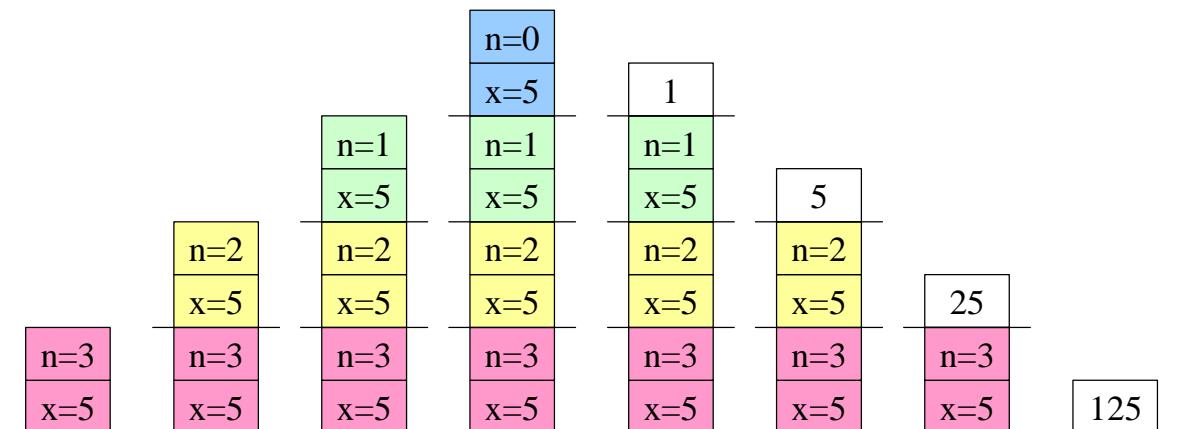


Funcionamiento de un algoritmo recursivo

- Se descompone el problema en problemas de menor complejidad (algunos de ellos de la misma naturaleza que el problema original).
- Se resuelve el problema para, al menos, un caso base.
- Se compone la solución final a partir de las soluciones parciales que se van obteniendo.

Ejemplo

```
static int potencia (int x, int n)
{
    if (n==0)                                // Caso base
        return 1;
    else                                       // Caso general
        return x * potencia(x,n-1);
}
```



Cálculo de 5^3

Diseño de algoritmos recursivos

1. Resolución de problema para los casos base:

- Sin emplear recursividad.
- Siempre debe existir algún caso base.

2. Solución para el caso general:

- Expresión de forma recursiva.
- Pueden incluirse pasos adicionales
(para combinar las soluciones parciales).

Siempre se debe avanzar hacia un caso base:
Las llamadas recursivas simplifican el problema y, en última instancia,
los casos base nos sirven para obtener la solución.

- Los casos base corresponden a situaciones que se pueden resolver con facilidad.
- Los demás casos se resuelven recurriendo, antes o después, a alguno(s) de los casos base.

De esta forma, podemos resolver problemas complejos que serían muy difíciles de resolver directamente.

Recursividad vs. iteración

Aspectos que hay que considerar al decidir cómo implementar la solución a un problema (de forma iterativa o de forma recursiva):

- **La carga computacional**
(tiempo de CPU y espacio en memoria) asociada a las llamadas recursivas.
- **La redundancia**
(algunas soluciones recursivas resuelven el mismo problema en repetidas ocasiones).
- **La complejidad de la solución**
(en ocasiones, la solución iterativa es muy difícil de encontrar).
- **La concisión, legibilidad y elegancia del código resultante**
(la solución recursiva del problema puede ser más sencilla).

Ejemplo: Versión mejorada del cálculo de x^n

```
static int potencia (int x, int n)
{
    int aux;

    if (n==0) {
        return 1;
    } else {
        aux = potencia(x, n/2);
        if (n%2 == 0)
            return aux * aux;
        else
            return x * aux * aux;
    }
}
```

Ejemplo: Sucesión de Fibonacci

$$\begin{aligned} Fib(0) &= Fib(1) = 1 \\ Fib(n) &= Fib(n - 1) + Fib(n - 2) \end{aligned}$$

Solución recursiva

```
static int fibonacci (int n)
{
    if ((n == 0) || (n == 1))
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

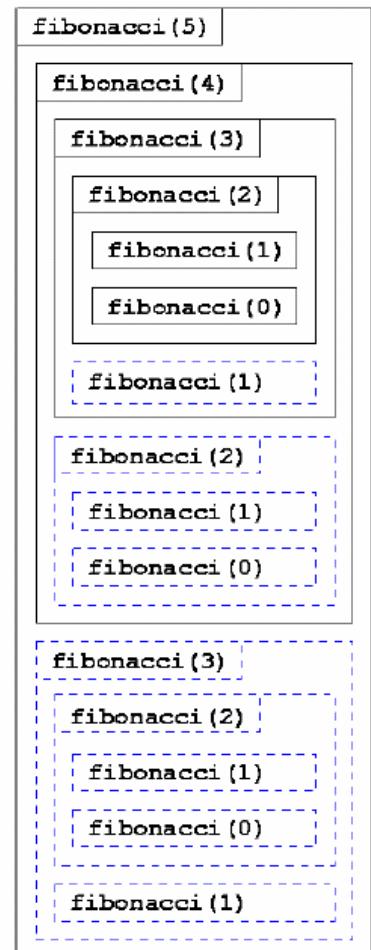
Solución iterativa

```
static int fibonacci (int n)
{
    int actual, ant1, ant2;

    ant1 = ant2 = 1;

    if ((n == 0) || (n == 1)) {
        actual = 1;
    } else
        for (i=2; i<=n; i++) {
            actual = ant1 + ant2;
            ant2 = ant1;
            ant1 = actual;
        }
    }

    return actual;
}
```



Cálculo recursivo de fibonacci(5)

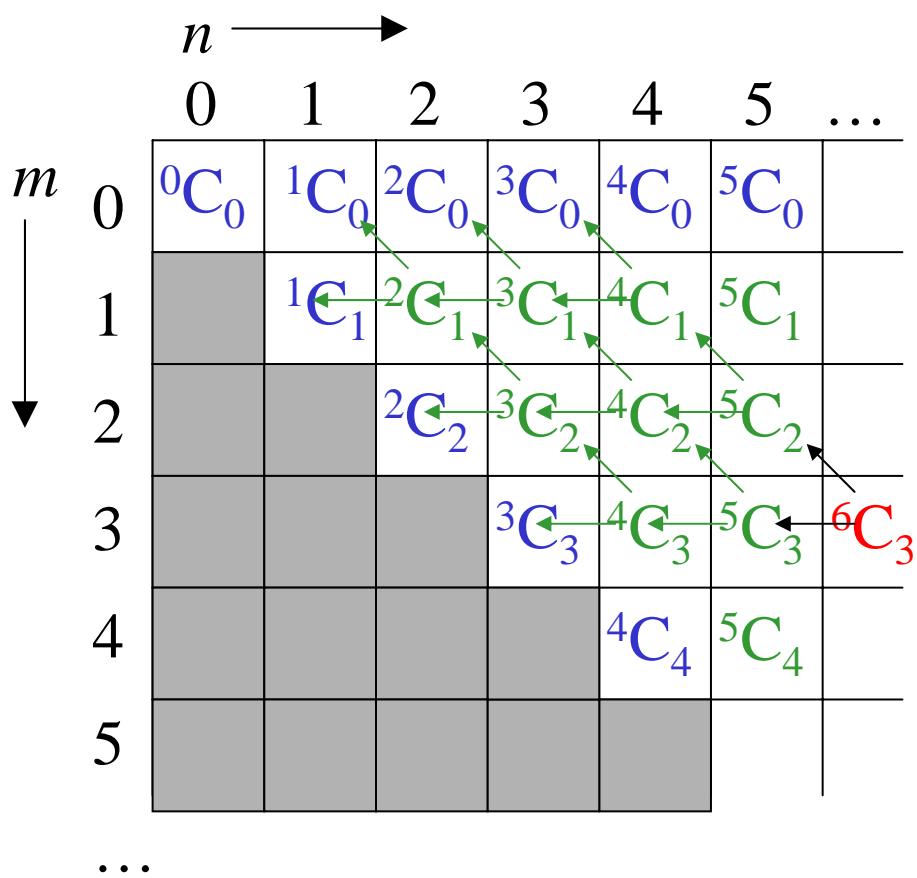
Ejemplo: Combinaciones

¿De cuántas formas se pueden seleccionar m elementos de un conjunto de n?

Si cogemos dos letras del conjunto {A,B,C,D,E} podemos obtener 10 parejas diferentes:

$$\begin{array}{cccc} \{A, B\} & \{A, C\} & \{A, D\} & \{A, E\} \\ & \{B, C\} & \{B, D\} & \{B, E\} \\ & & \{C, D\} & \{C, E\} \\ & & & \{D, E\} \end{array}$$

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$



Implementación recursiva:

```
static int combinaciones (int n, int m)
{
    if ((m == 0) || (m == n))
        return 1;
    else
        return combinaciones(n-1, m)
            + combinaciones(n-1, m-1);
}
```

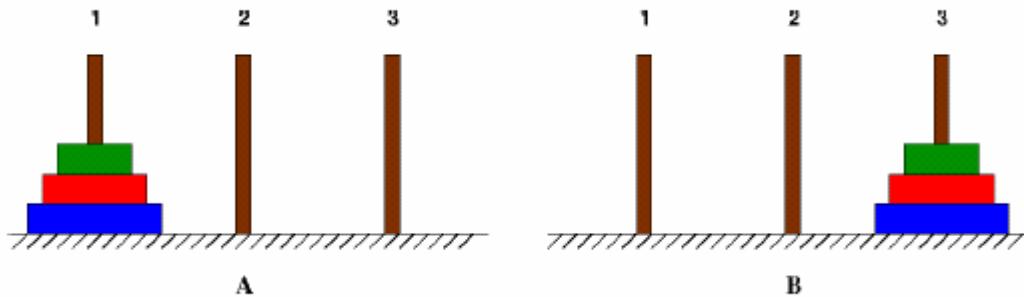
El algoritmo recursivo anterior es muy ineficiente porque realiza muchas veces los mismos cálculos.

De hecho, se realizan $\binom{n}{m}$ llamadas recursivas.

Para mejorar la eficiencia del algoritmo recursivo podemos buscar un algoritmo iterativo equivalente:

- Usando una matriz en la que iremos almacenando los valores que vayamos calculando para no repetir dos veces el mismo trabajo.
- Aplicando directamente la expresión $\binom{n}{m} = \frac{n!}{m!(n-m)!}$

Ejemplo: Las torres de Hanoi



Mover n discos del poste 1 al poste 3
(utilizando el poste 2 como auxiliar):

```
hanoi (n, 1, 2, 3);
```

Solución recursiva:

```
static void hanoi
    (int n, int inic, int tmp, int fin)
{
    if (n > 0) {

        // Mover n-1 discos de "inic" a "tmp".
        // El temporal es "fin".

        hanoi (n-1, inic, fin, tmp);

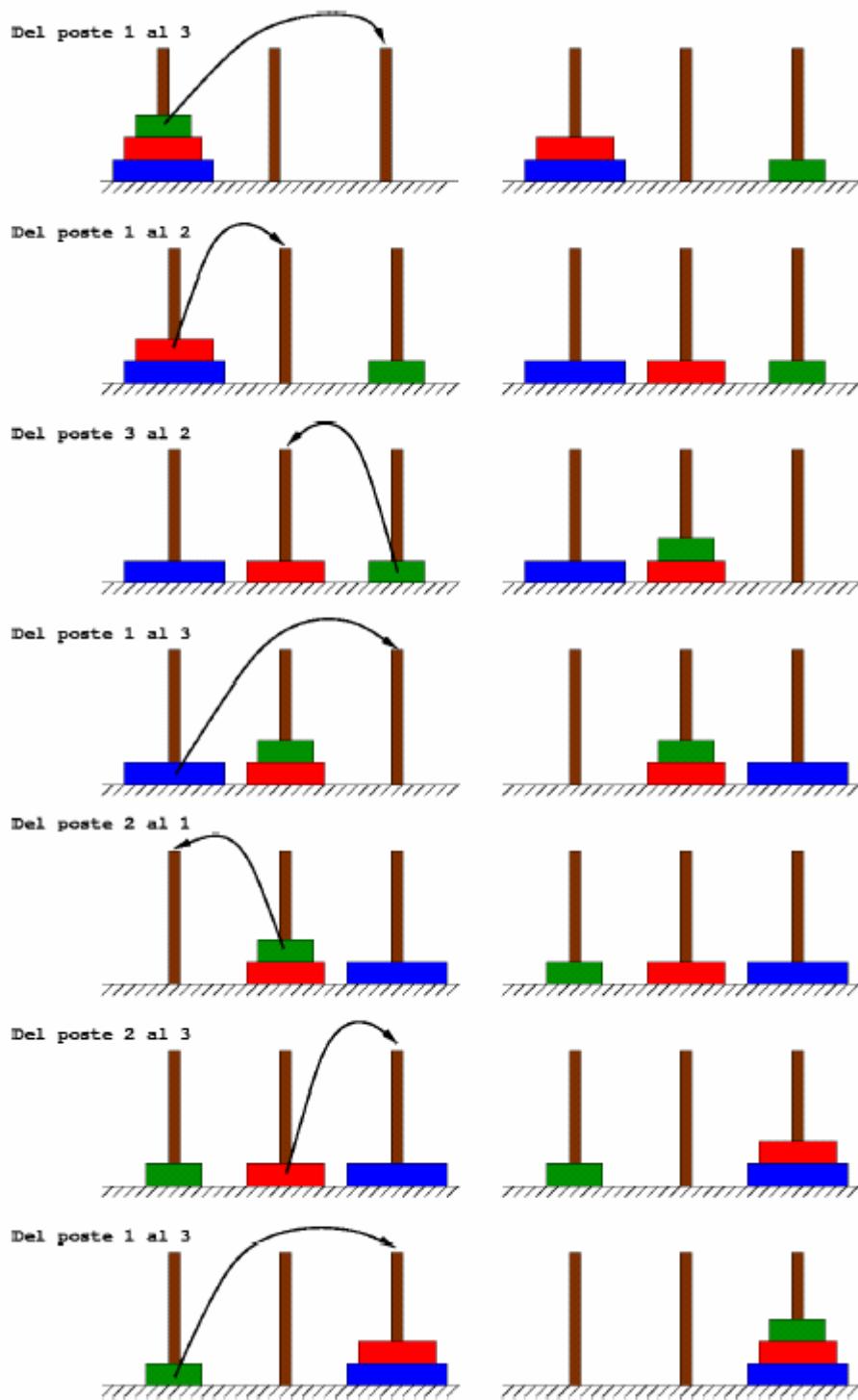
        // Mover el que queda en "inic" a "fin"

        System.out.println(inic+"->"+fin);

        // Mover n-1 discos de "tmp" a "fin".
        // El temporal es "inic".

        hanoi (n-1, tmp, inic, fin);
    }
}
```

Solución para 3 discos



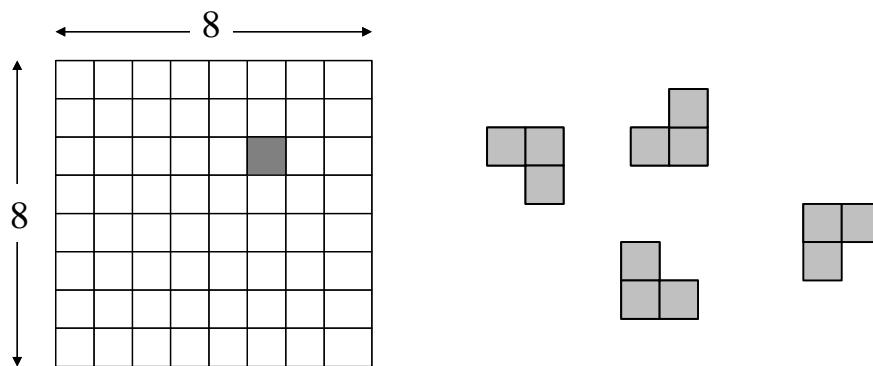
Según la leyenda, los monjes de un templo tenían que mover una pila de 64 discos sagrados de un sitio a otro. Sólo podían mover un disco al día y, en el templo, sólo había otro sitio en el que podían dejarlos, siempre ordenados de forma que los mayores quedasen en la base.

El día en que los monjes realizasen el último movimiento, el final del mundo habría llegado... ¿en cuántos días?

Recursividad

Relación de ejercicios

1. Demuestre por inducción que la función $Q(n)=1^2+2^2+3^2+\dots+n^2$ puede expresarse como $Q(n) = n(n+1)(2n+1) / 6$
2. Demuestre por inducción que, para todo n mayor o igual que 1, 133 divide a $11^{n+1} + 12^{2n-1}$
3. Demuestre por inducción que, para todo n mayor o igual que 4, $n! > 2^n$
4. Dado un tablero de ajedrez (de tamaño 8x8) al que le falta una casilla, ¿podemos llenar las demás casillas utilizando únicamente teselas con forma de L?



PISTA: Realice una demostración por inducción

- ¿Cuántas casillas faltan por llenar?
- ¿Cuántas teselas se han colocado ya?
- ¿Cuántas casillas quedan tras colocar otra tesela?

Generalice la demostración para un tablero de tamaño $2^n \times 2^n$

5. Demuestre que, con sellos de 4 y 5 céntimos, se puede franquear cualquier carta que requiera sellos por valor de 12 o más céntimos.
6. Un granjero ha comprado una pareja de conejos para criarlos y luego venderlos. Si la pareja de conejos produce una nueva pareja cada mes y la nueva pareja tarda un mes más en ser también productiva, ¿cuántos pares de conejos podrá poner a la venta el granjero al cabo de un año?

FUENTE: *Liber abaci*, sección III

7. Dado el siguiente fragmento de código:

```
static final double N = 2;
static final double PREC = 1e-6;

static double f (double x)
{
    return x*x-N;
}

static double bisect (double min, double max)
{
    double med = (min+max)/2;

    if (max-min<PREC) {
        return med;
    } else if (f(min)*f(med)<0) {
        return bisect (min,med);
    } else {
        return bisect (med,max);
    }
}
```

- a) ¿Qué calcula la llamada a la función recursiva `bisect(0,N)`? Si cambiamos el valor de `N`, ¿qué estaríamos calculando? ¿Y si cambiásemos la función `f(x)`?
- b) Implemente un algoritmo iterativo equivalente.

8. Dado el siguiente algoritmo recursivo:

```
void f(int num, int div)
{
    if (num>1) {
        if ((num%div) == 0) {
            System.out.println(div);
            f(num/div,div);
        } else {
            f(num,div+1);
        }
    }
}
```

- a) Dado un número cualquiera `x`, ¿qué nos muestra por pantalla la llamada a la función recursiva `f(x, 2)`? ¿Cuál sería un nombre más adecuado para la función `f`?
- b) Implemente un algoritmo iterativo equivalente.

9. ¿Cuál es el resultado de esta función para distintos valores de x?

```
static int f (int x)
{
    if (x > 100)
        return (x - 10);
    else
        return (f (f (x+11)));
}
```

10. Construya una función que convierta un número decimal en una cadena que represente el valor del número en hexadecimal (base 16). A continuación, generalice la función para convertir un número decimal en un número en base B (con B<10).

Recordatorio: El cambio de base se realiza mediante divisiones sucesivas por 16 en las cuales los restos determinan los dígitos hexadecimales del número según la siguiente correspondencia:

| | | | | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| Resto | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Dígito | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

Por ejemplo:

$$\begin{array}{r}
 65029 \quad |_{16} \\
 5 \quad 4064 \quad |_{16} \\
 \downarrow \qquad \downarrow \\
 0 \quad 254 \quad |_{16} \\
 \downarrow \qquad \downarrow \\
 14 \quad 15 \\
 \downarrow \qquad \downarrow \\
 5 \quad 0 \quad E \quad F \longrightarrow FE05 \\
 \end{array}$$

$65029_{10} = FE05_{16}$

11. Implemente, tanto de forma recursiva como de forma iterativa, una función que nos diga si una cadena de caracteres es simétrica (un palíndromo). Por ejemplo, “DABALEARROZALAZORRAELABAD” es un palíndromo.
12. Implemente, tanto de forma recursiva como de forma iterativa, una función que le dé la vuelta a una cadena de caracteres.

NOTA:

Obviamente, si la cadena es un palíndromo, la cadena y su inversa coincidirán.

13. Implemente, tanto de forma recursiva como de forma iterativa, una función que permitan calcular el número de combinaciones de n elementos tomados de m en m. Realice dos versiones de la implementación iterativa, una aplicando la fórmula y otra utilizando una matriz auxiliar (en la que se vaya construyendo el triángulo de Pascal).

14. Implemente, tanto de forma recursiva como de forma iterativa, una función que nos devuelva el máximo común divisor de dos números enteros utilizando el algoritmo de Euclides.

ALGORITMO DE EUCLIDES

Dados dos números enteros positivos m y n , tal que $m > n$,

para encontrar su máximo común divisor

(es decir, el mayor entero positivo que divide a ambos):

- Dividir m por n para obtener el resto r ($0 \leq r < n$)
- Si $r = 0$, el MCD es n .
- Si no, el máximo común divisor es $\text{MCD}(n,r)$.

15. La ordenación por mezcla (*mergesort*) es un método de ordenación que se basa en un principio muy simple: se ordenan las dos mitades de un vector y , una vez ordenadas, se mezclan. Escriba un programa que implemente este método de ordenación.

16. Diseñe e implemente un algoritmo que imprima todas las posibles descomposiciones de un número natural como suma de números menores que él (sumas con más de un sumando).

17. Diseñe e implemente un método recursivo que nos permita obtener el determinante de una matriz cuadrada de dimensión n .

18. Diseñe e implemente un programa que juegue al juego de cifras de “**Cifras y Letras**”. El juego consiste en obtener, a partir de 6 números, un número lo más cercano posible a un número de tres cifras realizando operaciones aritméticas con los 6 números.

19. **Problema de las 8 reinas:** Se trata de buscar la forma de colocar 8 reinas en un tablero de ajedrez de forma que ninguna de ellas amenace ni se vea amenazada por otra reina.

Algoritmo:

- Colocar la reina i en la primera casilla válida de la fila i
- Si una reina no puede llegar a colocarse en ninguna casilla, se vuelve atrás y se cambia la posición de la reina $i-1$
- Intentar colocar las reinas restantes en las filas que quedan

20. **Salida de un laberinto:** Se trata de encontrar un camino que nos permita salir de un laberinto definido en una matriz $N \times N$. Para movernos por el laberinto, sólo podemos pasar de una casilla a otra que sea adyacente a la primera y no esté marcada como una casilla prohibida (esto es, las casillas prohibidas determinan las paredes que forman el laberinto).

Algoritmo:

- Se comienza en la casilla $(0,0)$ y se termina en la casilla $(N-1, N-1)$
- Nos movemos a una celda adyacente si esto es posible.
- Cuando llegamos a una situación en la que no podemos realizar ningún movimiento que nos lleve a una celda que no hayamos visitado ya, retrocedemos sobre nuestros pasos y buscamos un camino alternativo.

Refactorización

Definición

Refactorización (n)

Cambio realizado a la estructura interna del software para hacerlo... más fácil de comprender
y más fácil de modificar
sin cambiar su comportamiento observable.

Refactorizar (v)

Reestructurar el software aplicando una secuencia de refactorizaciones.

¿Por qué se “refactoriza” el software?

1. Para mejorar su diseño
 - a. Conforme se modifica, el software pierde su estructura.
 - b. Eliminar código duplicado simplificar su mantenimiento.
2. Para hacerlo más fácil de entender

p.ej. La legibilidad del código facilita su mantenimiento
3. Para encontrar errores

p.ej. Al reorganizar un programa, se pueden apreciar con mayor facilidad las suposiciones que hayamos podido hacer.
4. Para programar más rápido

Al mejorar el diseño del código, mejorar su legibilidad y reducir los errores que se cometan al programar, se mejora la productividad de los programadores.

Ejemplo

Generación de números primos

© Robert C. Martin, “The Craftsman” column,
Software Development magazine, julio-septiembre 2002

Para conseguir un trabajo nos plantean el siguiente problema...

Implementar una clase que sirva para calcular todos los números primos de 1 a N utilizando la criba de Eratóstenes

Una primera solución

Necesitamos crear un método que reciba como parámetro un valor máximo y devuelva como resultado un vector con los números primos.

Para intentar lucirnos, escribimos...

```
/**  
 * Clase para generar todos los números primos de 1 hasta  
 * un número máximo especificado por el usuario. Como  
 * algoritmo se utiliza la criba de Eratóstenes.  
 * <p>  
 * Eratóstenes de Cirene (276 a.C., Cirene, Libia - 194  
 * a.C., Alejandría, Egipto) fue el primer hombre que  
 * calculó la circunferencia de la Tierra. También  
 * se le conoce por su trabajo con calendarios que ya  
 * incluían años bisiestos y por dirigir la mítica  
 * biblioteca de Alejandría.  
 * <p>  
 * El algoritmo es bastante simple: Dado un vector de  
 * enteros empezando en 2, se tachan todos los múltiplos  
 * de 2. A continuación, se encuentra el siguiente  
 * entero no tachado y se tachan todos sus múltiplos. El  
 * proceso se repite hasta que se pasa de la raíz cuadrada  
 * del valor máximo. Todos los números que queden sin  
 * tachar son números primos.  
 *  
 * @author Fernando Berzal  
 * @version 1.0 Enero'2005 (FB)  
 */
```

El código lo escribimos todo en un único (y extenso) método, que procuramos comentar correctamente:

```
public class Criba
{
    /**
     * Generar números primos de 1 a max
     * @param max es el valor máximo
     * @return Vector de números primos
     */
    public static int[] generarPrimos (int max)
    {
        int i,j;

        if (max >= 2) {

            // Declaraciones
            int dim = max + 1; // Tamaño del array
            boolean[] esPrimo = new boolean[dim];

            // Inicializar el array
            for (i=0; i<dim; i++)
                esPrimo[i] = true;

            // Eliminar el 0 y el 1, que no son primos
            esPrimo[0] = esPrimo[1] = false;

            // Criba
            for (i=2; i<Math.sqrt(dim)+1; i++) {
                if (esPrimo[i]) {
                    // Eliminar los múltiplos de i
                    for (j=2*i; j<dim; j+=i)
                        esPrimo[j] = false;
                }
            }

            // ¿Cuántos primos hay?
            int cuenta = 0;

            for (i=0; i<dim; i++) {
                if (esPrimo[i])
                    cuenta++;
            }
        }
    }
}
```

```

        // Rellenar el vector de números primos
        int[] primos = new int[cuenta];

        for (i=0, j=0; i<dim; i++) {
            if (esPrimo[i])
                primos[j++] = i;
        }

        return primos;

    } else { // max < 2

        return new int[0]; // Vector vacío
    }
}
}

```

Para comprobar que nuestro código funciona bien, decidimos probarlo con un programa que incluye distintos casos de prueba (para lo que usaremos la herramienta **JUnit**, disponible en <http://www.junit.org/>):

```

import junit.framework.*; // JUnit
import java.util.*;

// Clase con casos de prueba para Criba

public class CribaTest extends TestCase
{
    // Programa principal (usa un componente de JUnit)

    public static void main(String args[])
    {
        junit.swingui.TestRunner.main (
            new String[] {"CribaTest"});
    }

    // Constructor

    public CribaTest (String nombre)
    {
        super(nombre);
    }
}

```

```

// Casos de prueba

public void testPrimos()
{
    int[] nullArray = Criba.generarPrimos(0);
    assertEquals(nullArray.length, 0);

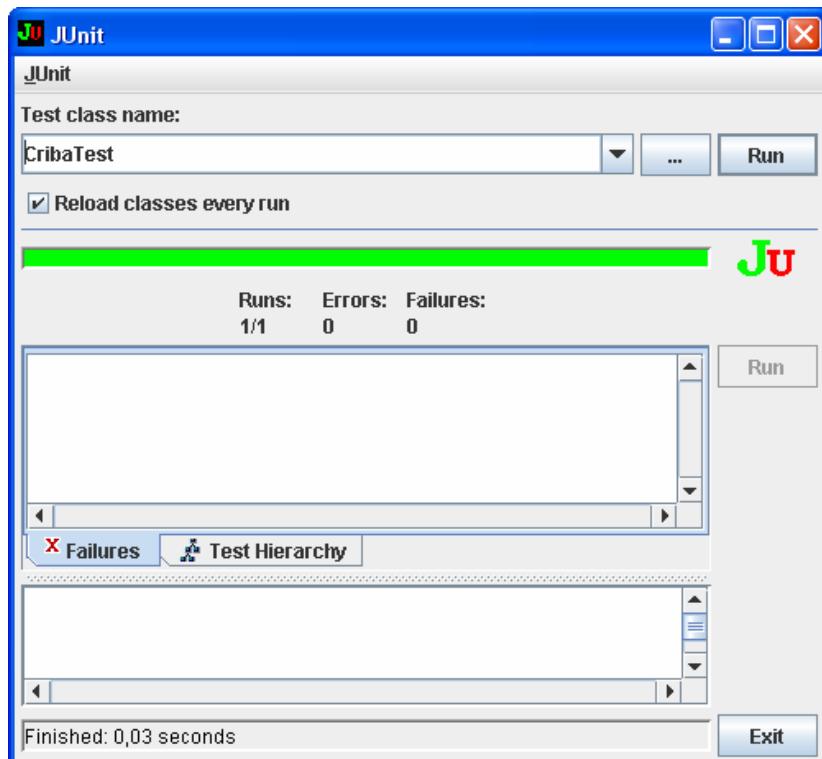
    int[] minArray = Criba.generarPrimos(2);
    assertEquals(minArray.length, 1);
    assertEquals(minArray[0], 2);

    int[] threeArray = Criba.generarPrimos(3);
    assertEquals(threeArray.length, 2);
    assertEquals(threeArray[0], 2);
    assertEquals(threeArray[1], 3);

    int[] centArray = Criba.generarPrimos(100);
    assertEquals(centArray.length, 25);
    assertEquals(centArray[24], 97);
}
}

```

Al ejecutar los casos de prueba, conseguimos tener ciertas garantías de que el programa funciona correctamente:



Después de eso, vamos orgullosos a enseñar nuestro programa y... un programador no dice que, si queremos el trabajo, mejor no le enseñemos eso al jefe de proyecto (que no tiene demasiada paciencia)

Veamos qué cosas hemos de mejorar...

Primeras mejoras

Parece evidente que nuestro método `generarPrimos` realiza tres funciones diferentes, por lo que de `generarPrimos` extraemos tres métodos diferentes. Además, buscamos un nombre más adecuado para la clase y eliminamos todos los comentarios innecesarios.

```
/*
 * Esta clase genera todos los números primos de 1 hasta un
 * número máximo especificado por el usuario utilizando la
 * criba de Eratóstenes
 * <p>
 * Dado un vector de enteros empezando en 2, se tachan todos
 * los múltiplos de 2. A continuación, se encuentra el
 * siguiente entero no tachado y se tachan sus múltiplos.
 * Cuando se llega a la raíz cuadrada del valor máximo, los
 * números que queden sin tachar son los números primos
 *
 * @author Fernando Berzal
 * @version 2.0 Enero'2005 (FB)
 */

public class GeneradorDePrimos
{
    private static int dim;
    private static boolean esPrimo[];
    private static int primos[];

    public static int[] generarPrimos (int max)
    {
        if (max < 2) {
            return new int[0]; // Vector vacío
        } else {
            inicializarCriba(max);
            cribar();
            llenarPrimos();
            return primos;
        }
    }
}
```

```

private static void inicializarCriba (int max)
{
    int i;

    dim = max + 1;
    esPrimo = new boolean[dim];

    for (i=0; i<dim; i++)
        esPrimo[i] = true;

    esPrimo[0] = esPrimo[1] = false;
}

private static void cribar ( )
{
    int i,j;

    for (i=2; i<Math.sqrt(dim)+1; i++) {
        if (esPrimo[i]) {
            // Eliminar los múltiplos de i
            for (j=2*i; j<dim; j+=i)
                esPrimo[j] = false;
        }
    }
}

private static void rellenarPrimos ( )
{
    int i, j, cuenta;

    // Contar primos
    cuenta = 0;

    for (i=0; i<dim; i++)
        if (esPrimo[i])
            cuenta++;

    // Rellenar el vector de números primos
    primos = new int[cuenta];

    for (i=0, j=0; i<dim; i++)
        if (esPrimo[i])
            primos[j++] = i;
}
}

```

Los mismos casos de prueba de antes nos permiten comprobar que, tras la refactorización, el programa sigue funcionando correctamente.

Un segundo intento

El código ha mejorado pero aún es algo más enrevesado de la cuenta:

eliminamos la variable `dim` (nos vale `esPrimo.length`),
elegimos identificadores más adecuados para los métodos y
reorganizamos el interior del método `inicializarCandidatos`
(el antiguo `inicializarCriba`).

```
public class GeneradorDePrimos
{
    private static boolean esPrimo[];
    private static int primos[];

    public static int[] generarPrimos (int max)
    {
        if (max < 2) {
            return new int[0];
        } else {
            inicializarCandidatos(max);
            eliminarMultiplos();
            obtenerCandidatosNoEliminados();
            return primos;
        }
    }

    private static void inicializarCandidatos (int max)
    {
        int i;

        esPrimo = new boolean[max+1];
        esPrimo[0] = esPrimo[1] = false;
        for (i=2; i<esPrimo.length; i++)
            esPrimo[i] = true;
    }

    private static void eliminarMultiplos ()
    ... // Código del antiguo método cribar()

    private static void obtenerCandidatosNoEliminados ()
    ... // Código del antiguo método rellenarPrimos()
}
```

El código resulta más fácil de leer tras la refactorización.

Mejoras adicionales

El bucle anidado de `eliminarMultiplos` podía eliminarse si usamos un método auxiliar para eliminar los múltiplos de un número concreto.

Por otro lado, la raíz cuadrada que aparece en `eliminarMultiplos` no queda muy claro de dónde proviene (en realidad, es el valor máximo que puede tener el menor factor de un número no primo menor o igual que N). Además, el +1 resulta innecesario.

```
private static void eliminarMultiplos ()  
{  
    int i;  
  
    for (i=2; i<maxFactor(); i++)  
        if (esPrimo[i])  
            eliminarMultiplosDe(i);  
}  
  
private static int maxFactor ()  
{  
    return (int) Math.sqrt(esPrimo.length) + 1;  
}  
  
private static void eliminarMultiplosDe (int i)  
{  
    int multiplo;  
  
    for (multiplo=2*i;  
         multiplo<esPrimo.length;  
         multiplo+=i)  
        esPrimo[multiplo] = false;  
}
```

De forma análoga, el método `obtenerCandidatosNoEliminados` tiene dos partes bien definidas, por lo que podemos extraer un método que se limite a contar el número de primos obtenidos...

Hemos ido realizando cambios que mejoran la implementación sin modificar su comportamiento externo (su interfaz), algo que verificamos tras cada refactorización volviendo a ejecutar los casos de prueba.

¿Cuándo hay que refactorizar?

Cuando se está escribiendo nuevo código

Al añadir nueva funcionalidad a un programa (o modificar su funcionalidad existente), puede resultar conveniente refactorizar:

- para que éste resulte más fácil de entender, o
- para simplificar la implementación de las nuevas funciones cuando el diseño no estaba inicialmente pensado para lo que ahora tenemos que hacer.

Cuando se intenta corregir un error

La mayor dificultad de la depuración de programas radica en que hemos de entender exactamente cómo funciona el programa para encontrar el error. Cualquier refactorización que mejore la calidad del código tendrá efectos positivos en la búsqueda del error.

De hecho, si el error “se coló” en el código es porque no era lo suficientemente claro cuando lo escribimos.

Cuando se revisa el código

Una de las actividades más productivas desde el punto de vista de la calidad del software es la realización de revisiones del código (recorridos e inspecciones). Llevada a su extremo, la programación siempre se realiza por parejas (*pair programming*, una de las técnicas de la programación extrema, XP [eXtreme Programming]).

¿Por qué es importante la refactorización?

Cuando se corrige un error o se añade una nueva función, el valor actual de un programa aumenta. Sin embargo, para que un programa siga teniendo valor, debe ajustarse a nuevas necesidades (mantenerse), que puede que no sepamos prever con antelación. La refactorización, precisamente, facilita la adaptación del código a nuevas necesidades.

¿Qué síntomas indican que debería refactorizar?

El código es más difícil de entender (y, por tanto, de cambiar) cuando:

- usa identificadores mal escogidos,
- incluye fragmentos de código duplicados,
- incluye lógica condicional compleja,
- los métodos usan un número elevado de parámetros,
- incluye fragmentos de código secuencial muy extensos,
- está dividido en módulos enormes (estructura monolítica),
- los módulos en los que se divide no resultan razonables desde el punto de vista lógico (cohesión baja),
- los distintos módulos de un sistema están relacionados con otros muchos módulos de un sistema (acoplamiento fuerte),
- un método accede continuamente a los datos de un objeto de una clase diferente a la clase en la que está definida (posiblemente, el método debería pertenecer a la otra clase),
- una clase incluye variables de instancia que deberían ser variables locales de alguno(s) de sus métodos,
- métodos que realizan funciones análogas se usan de forma diferente (tienen nombres distintos y/o reciben los parámetros en distinto orden),
- un comentario se hace imprescindible para poder entender un fragmento de código (deberíamos re-escribir el código de forma que podamos entender su significado).

NOTA: Esto no quiere decir que dejemos de comentar el código.

Algunas refactorizaciones comunes

Algunos IDEs (p.ej. Eclipse) permiten realizarlas de forma automática.

Rrenombrar método [*rename method*]

Cuando el nombre de un método no refleja su propósito

1. Declarar un método nuevo con el nuevo nombre.
2. Copiar el cuerpo del antiguo método al nuevo método
(y realizar cualquier modificación que resulte necesaria).
3. Compilar
(para verificar que no hemos introducido errores sintácticos)
4. Reemplazar el cuerpo del antiguo método por una llamada al nuevo método (este paso se puede omitir si no se hace referencia al antiguo método desde muchos lugares diferentes).
5. Compilar y probar.
6. Encontrar todas las referencias al antiguo método y cambiarlas por invocaciones al nuevo método.
7. Eliminar el antiguo método.
8. Compilar y probar.

NOTA:

Si el antiguo método era un método público usado por otros componentes o aplicaciones y no podemos eliminarlo, el antiguo método se deja en su lugar (como una llamada al nuevo método) y se marca como “*deprecated*” con Javadoc (@deprecated).

Extraer método [*extract method*]

Convertir un fragmento de código en un método cuyo identificador explique el propósito del fragmento de código.

1. Crear un nuevo método y buscarle un identificador adecuado.
2. Copiar el fragmento de código en el cuerpo del método.
3. Buscar en el código extraído referencias a variables locales del método original (estas variables se convertirán en los parámetros, variables locales y resultado del nuevo método):
 - a. Si una variable se usa sólo en el fragmento de código extraído, se declara en el nuevo método como variable local de éste.
 - b. Si el valor de una variable sólo se lee en el fragmento de código extraído, la variable será un parámetro del nuevo método.
 - c. Si una variable se modifica en el fragmento de código extraído, se intenta convertir el nuevo método en una función que da como resultado el valor que hay que asignarle a la variable modificada.
 - d. Compilar el código para comprobar que todas las referencias a variables son válidas
4. Reemplazar el fragmento de código en el método original por una llamada al nuevo método.
5. Eliminar las declaraciones del método original correspondientes a las variables que ahora son variables locales del nuevo método.
6. Compilar y probar.

Pruebas de unidad con JUnit

Cuando se implementa software, resulta recomendable comprobar que el código que hemos escrito funciona correctamente.

Para ello, implementamos pruebas que verifican que nuestro programa genera los resultados que de él esperamos.

Conforme vamos añadiéndole nueva funcionalidad a un programa, creamos nuevas pruebas con las que podemos medir nuestro progreso y comprobar que lo que antes funcionaba sigue funcionando tras haber realizado cambios en el código (*test de regresión*).

Las pruebas también son de vital importancia cuando refactorizamos (aunque no añadamos nueva funcionalidad, estamos modificando la estructura interna de nuestro programa y debemos comprobar que no introducimos errores al refactorizar).

Automatización de las pruebas

Para agilizar la realización de las pruebas resulta práctico que un test sea completamente automático y compruebe los resultados esperados.

- ✗ No es muy apropiado llamar a una función, guardar el resultado en algún sitio y después tener que comprobar manualmente si el resultado era el deseado.
- ✓ Mantener automatizado un conjunto amplio de tests permite reducir el tiempo que se tarda en depurar errores y en verificar la corrección del código.

JUnit

Herramienta especialmente diseñada para implementar y automatizar la realización de pruebas de unidad en Java.

Dada una clase de nuestra aplicación...

- ✚ En una clase aparte definimos un conjunto de casos de prueba
 - La clase hereda de `junit.framework.TestCase`
 - Cada caso de prueba se implementa en un método aparte.
 - El nombre de los casos de prueba siempre comienza por `test`.

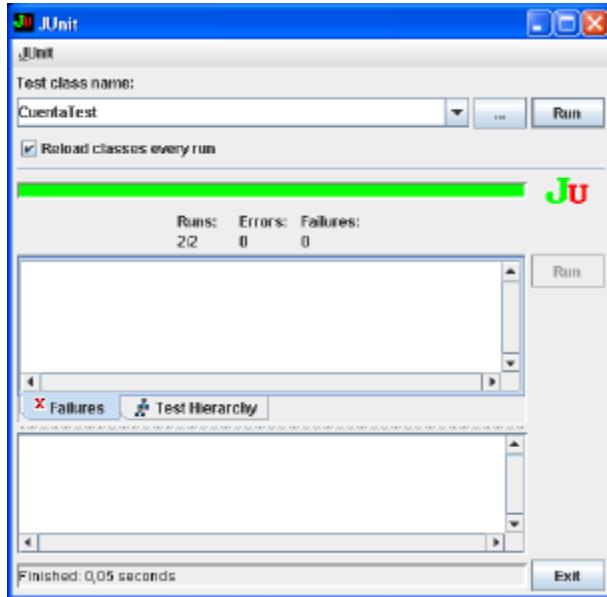
```
import junit.framework.*;  
  
public class CuentaTest extends TestCase  
{  
    ...  
}
```

- ✚ Cada caso de prueba invoca a una serie de métodos de nuestra clase y comprueba los resultados que se obtienen tras invocarlos.
 - Creamos uno o varios objetos de nuestra clase con `new`
 - Realizamos operaciones con ellos.
 - Definimos aserciones (condiciones que han de cumplirse).

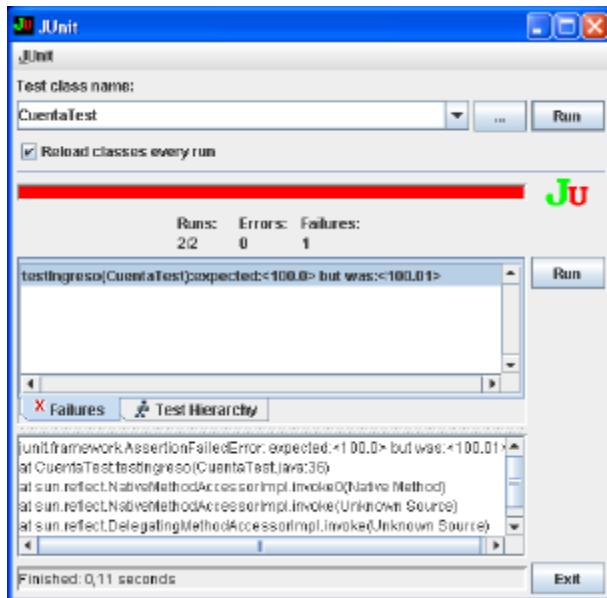
```
public void testCuentaNueva ()  
{  
    Cuenta cuenta = new Cuenta();  
    assertEquals(cuenta.getSaldo(), 0.00);  
}  
  
public void testIngreso ()  
{  
    Cuenta cuenta = new Cuenta();  
    cuenta.ingresar(100.00);  
    assertEquals(cuenta.getSaldo(), 100.00);  
}
```

Finalmente, ejecutamos los casos de prueba con **JUnit**:

- Si todos los casos de prueba funcionan correctamente...



- Si algún caso de prueba falla...



Tendremos que localizar el error y corregirlo con ayuda de los mensajes que nos muestra JUnit.

MUY IMPORTANTE: Que nuestra implementación supere todos los casos de prueba no quiere decir que sea correcta; sólo quiere decir que funciona correctamente para los casos de prueba que hemos diseñado.

Apéndice: Cómo ejecutar JUnit desde nuestro propio código

Para lanzar JUnit desde nuestro propio código, sin tener que ejecutar la herramienta a mano, hemos de implementar el método `main` en nuestra clase y definir un sencillo constructor.

```
import junit.framework.*;

public class CuentaTest extends TestCase
{
    public static void main(String args[])
    {
        junit.swingui.TestRunner.main (
            new String[] {"CuentaTest"});
    }

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

    // Casos de prueba
    ...
}
```

Ejemplo: La clase Money

Basado en “Test-Driven Development by Example”, pp. 1-87 © Kent Beck

Vamos a definir una clase, denominada `Money`, para representar cantidades de dinero que pueden estar expresadas en distintas monedas

Por ejemplo, queremos usar esta clase para generar informes como...

| Empresa | Acciones | Precio | Total |
|------------|----------|--------|----------|
| Telefónica | 200 | 10 EUR | 2000 EUR |
| Vodafone | 100 | 50 EUR | 5000 EUR |
| 7000 EUR | | | |

El problema es que también nos podemos encontrar con situaciones como la siguiente ...

| Empresa | Acciones | Precio | Total |
|-----------|----------|--------|----------|
| Microsoft | 200 | 13 USD | 2600 USD |
| Indra | 100 | 50 EUR | 5000 EUR |
| 7000 EUR | | | |

donde hemos tenido que utilizar el tipo de cambio actual ($1\text{€}=\$1.30$)

Comenzamos creando una clase para representar cantidades de dinero:

```
public class Money
{
    private int cantidad;
    private String moneda;

    public Money (int cantidad, String moneda)
    {
        this.cantidad = cantidad;
        this.moneda = moneda;
    }

    public int getCantidad() { return cantidad; }
    public String getMoneda() { return moneda; }
}
```

Una de las cosas que tendremos que hacer es *sumar cantidades*, por lo que podemos idear un caso de prueba como el siguiente:

```
import junit.framework.*;  
  
public class MoneyTest extends TestCase  
{  
    public void testSumaSimple()  
    {  
        Money m10 = new Money(10, "EUR");  
        Money m20 = new Money(20, "EUR");  
  
        Money esperado = new Money(30, "EUR");  
        Money resultado = m10.add(m20);  
  
        Assert.assertEquals(resultado, esperado);  
    }  
}
```

Al idear el caso de prueba, nos estamos fijando en cómo tendremos que usar nuestra clase en la práctica, lo que nos es extremadamente útil para definir su interfaz.

En este caso, nos hace falta añadir un método add a la clase Money para poder compilar y ejecutar el caso de prueba...

Creamos una implementación inicial de este método:

```
public Money add(Money m)  
{  
    int total = getCantidad() + m.getCantidad();  
    return new Money(total, getMoneda());  
}
```

Compilamos y ejecutamos el test para llevarnos una sorpresa...



El caso de prueba falla porque la comparación de objetos en Java, por defecto, se limita a comparar referencias (no compara el estado de los objetos, que es lo que podríamos pensar [erróneamente]).

La comparación de objetos en Java se realiza con el método `equals`, que puede recibir como parámetro un objeto cualquiera.

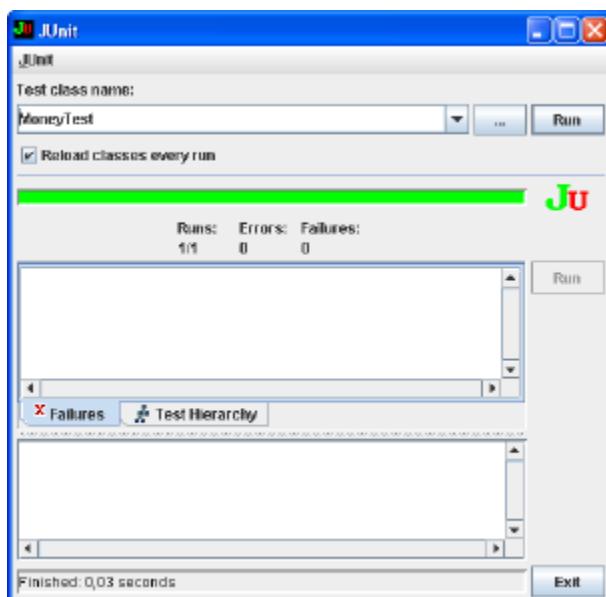
Por tanto, hemos de definir el método `equals` en la clase `Money`:

```
public boolean equals (Object obj)
{
    Money aux;
    boolean iguales;

    if (obj instanceof Money) {
        aux = (Money) obj;
        iguales = aux.getMoneda().equals(getMoneda())
                    && (aux.getCantidad() == getCantidad());
    } else {
        iguales = false;
    }

    return iguales;
}
```

Volvemos a ejecutar el caso de prueba...



... y ahora sí podemos seguir avanzando

De todas formas, para asegurarnos de que todo va bien, creamos un caso de prueba específico que verifique el funcionamiento de `equals`

```
public void testEquals()
{
    Money m10 = new Money (10, "EUR");
    Money m20 = new Money (20, "EUR");

    Assert.assertEquals(m10,m10);
    Assert.assertEquals(m20,m20);
    Assert.assertTrue(!m10.equals(m20));
    Assert.assertTrue(!m20.equals(m10));
    Assert.assertTrue(!m10.equals(null));
}
```

Al ejecutar nuestros dos casos de prueba con JUNIT vemos que todo marcha como esperábamos.



Sin embargo, comenzamos a ver que existe código duplicado (y ya sabemos que eso no es una buena señal), por lo que utilizamos la posibilidad que nos ofrece JUNIT de definir variables de instancia en la clase que hereda de `TestCase` (variables que hemos de inicializar en el método `setUp()`)

```
public class MoneyTest extends TestCase
{
    Money m10;
    Money m20;

    public void setUp()
    {
        m10 = new Money (10, "EUR");
        m20 = new Money (20, "EUR");
    }
    ...
}
```

Aparte de sumar cantidades de dinero, también tenemos que ser capaces de *multiplicar una cantidad por un número entero* (p.ej. número de acciones por precio de cada acción)

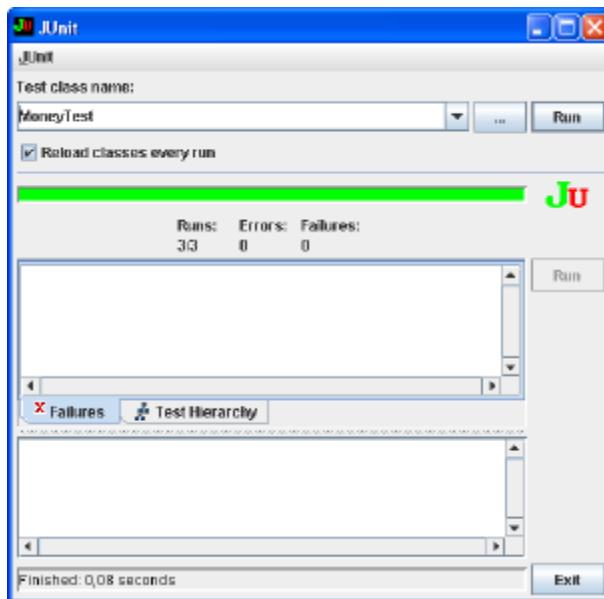
Podemos añadir un nuevo caso de prueba que utilice esta función:

```
public void testMultiplicar ()  
{  
    Assert.assertEquals ( m10.times(2) , m20 );  
    Assert.assertEquals ( m10.times(2) , m20 );  
    Assert.assertEquals ( m10.times(10) , m20.times(5) );  
}
```

Obviamente, también tendremos que definir `times` en `Money`:

```
public Money times (int n)  
{  
    return new Money ( n*getCantidad() , getMoneda() );  
}
```

Compilamos y ejecutamos los casos de prueba con JUnit:



Poco a poco, vamos añadiéndole funcionalidad a nuestra clase:

Cada vez que hacemos cambios, volvemos a ejecutar todos los casos de prueba para confirmar que no hemos estropeado nada.

Una vez que hemos comprobado que ya somos capaces de hacer operaciones cuando todo se expresa en la misma moneda, tenemos que comenzar a trabajar con distintas monedas.

Por ejemplo:

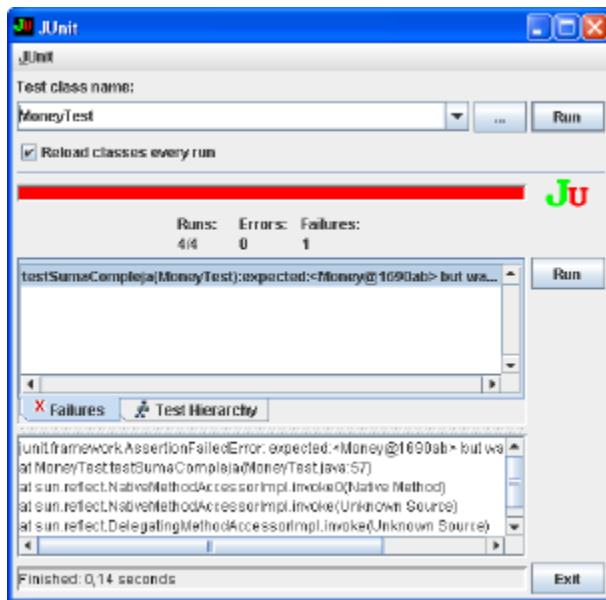
```
public void testSumaCompleja ()  
{  
    Money euros      = new Money(100, "EUR");  
    Money dollars    = new Money(130, "USD");  
    Money resultado = euros.add(dollars);  
    Money banco     = Bank.exchange(dollars, "EUR")  
    Money esperado   = euros.add(banco);  
  
    Assert.assertEquals ( resultado, esperado );  
}
```

Para hacer el cambio de moneda, suponemos que tenemos acceso a un banco que se encarga de hacer la conversión. Hemos de crear una clase auxiliar `Bank` que se va a encargar de consultar los tipos de cambio y aplicar la conversión correspondiente:

```
public class Bank  
{  
    public static Money exchange  
        (Money dinero, String moneda)  
    {  
        int cantidad = 0;  
  
        if (dinero.getMoneda().equals(moneda)) {  
            cantidad = dinero.getCantidad();  
        } else if ( dinero.getMoneda().equals("EUR")  
                    && moneda.equals("USD")) {  
            cantidad = (130*dinero.getCantidad())/100;  
        } else if ( dinero.getMoneda().equals("USD")  
                    && moneda.equals("EUR")) {  
            cantidad = (100*dinero.getCantidad())/130;  
        }  
  
        return new Money(cantidad,moneda);  
    }  
}
```

Por ahora, nos hemos limitado a realizar una conversión fija para probar el funcionamiento de nuestra clase Money (en una aplicación real tendríamos que conectarnos realmente con el banco).

Obviamente, al ejecutar nuestros casos de prueba se produce un error:



Hemos de corregir la implementación interna del método add de la clase Money para que tenga en cuenta el caso de que las cantidades correspondan a monedas diferentes:

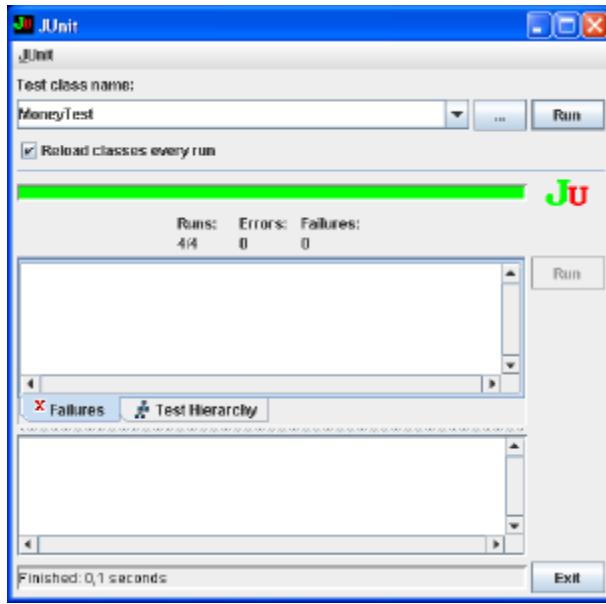
```
public Money add (Money dinero)
{
    Money convertido;
    int total;

    if (getMoneda().equals(dinero.getMoneda()))
        convertido = dinero;
    else
        convertido = Bank.exchange(dinero, getMoneda());

    total = getCantidad() + convertido.getCantidad();

    return new Money( total, getMoneda());
}
```

Volvemos a ejecutar los casos de prueba y comprobamos que, ahora sí, las sumas se hacen correctamente:



Si todavía no las tuviésemos todas con nosotros, podríamos seguir añadiendo casos de prueba para adquirir más confianza en la implementación que acabamos de realizar.

Por ejemplo, el siguiente caso de prueba comprueba el funcionamiento del banco al realizar conversiones de divisas:

```
public void testBank ( )
{
    Money euros      = new Money(10, "EUR");
    Money dollars    = new Money(13, "USD");

    Assert.assertEquals (
        Bank.exchange(dollars, "EUR"), euros );
    Assert.assertEquals (
        Bank.exchange(dollars, "USD"), dollars );
    Assert.assertEquals (
        Bank.exchange(euros,   "EUR"), euros );
    Assert.assertEquals (
        Bank.exchange(euros, "USD"), dollars );
}
```

Comentarios finales: El método `toString`

Para que resulte más fácil interpretar los mensajes generados por JUNIT, resulta recomendable definir el método `toString()` en todas las clases que definamos. Por ejemplo:

```
public class Money
{
    ...
    public String toString ()
    {
        return getCantidad() + " " + getMoneda();
    }
}
```

RECORDATORIO: `toString()` es un método que se emplea en Java para convertir un objeto cualquiera en una cadena de caracteres.

Teniendo definido el método anterior, en nuestras aplicaciones podríamos escribir directamente...

```
...
Money share = new Money(13, "USD");
Money investment = share.times(200);
Money euros = Bank.exchange(investment, "EUR");

System.out.println(investment + " (" + euros + ")");
```

y obtener como resultado en pantalla:

2600 USD (2000 EUR)

TDD

[Test-Driven Development]

Consiste en implementar las pruebas de unidad antes incluso de comenzar a escribir el código de un módulo.

Las pruebas de unidad consisten en comprobaciones (manuales o automatizadas) que se realizan para verificar que el código correspondiente a un módulo concreto de un sistema software funciona de acuerdo con los requisitos del sistema.

Tradicionalmente, las pruebas se realizan a posteriori

Los casos de prueba se suelen escribir después de implementar el módulo cuyo funcionamiento pretenden verificar.

Como mucho, se preparan en paralelo si el programador y la persona que realiza las pruebas [*tester*] no son la misma persona.

En TDD, las pruebas se preparan antes de comenzar a escribir el código.

Primero escribimos un caso de prueba y sólo después implementamos el código necesario para que el caso de prueba se pase con éxito

Aunque pueda parecer extraño, TDD ofrece algunas ventajas:

- Al escribir primero los casos de prueba, definimos de manera formal los requisitos que esperamos que cumpla nuestra aplicación.

Los casos de prueba sirven de **documentación** del sistema.

- Al escribir una prueba de unidad, pensamos en la forma correcta de utilizar un módulo que aún no existe.

Hacemos hincapié en el diseño de la **interfaz** de un módulo antes de centrarnos en su implementación (algo siempre bueno: “la interfaz debe determinar la implementación, y no al revés”).

- La **ejecución** de los casos de prueba se realiza de forma **automatizada** (por ejemplo, con ayuda de JUNIT)

Al ejecutar los casos de prueba detectamos si hemos introducido algún error al tocar el código para realizar cualquier cambio en nuestra aplicación (ya sea para añadirle nuevas funciones o para reorganizar su estructura interna [refactorización]).

- Los casos de prueba nos permiten **perder el miedo** a realizar modificaciones en el código

Tras realizar pequeñas modificaciones sobre el código, volveremos a ejecutar los casos de prueba para comprobar inmediatamente si hemos cometido algún error o no.

- Los casos de prueba definen claramente cuándo termina nuestro **trabajo** (cuando se pasan con éxito todas los casos de prueba).

El proceso de construcción de software se convierte en un ciclo:

1. Añadir un nuevo caso de prueba
que recoja algo que nuestro módulo debe realizar correctamente.



2. Ejecutar los casos de prueba
para comprobar que el caso recién añadido falla.



3. Realizar pequeños cambios en la implementación
(en función de lo que queremos que haga nuestra aplicación).



4. Ejecutar los casos de prueba
hasta que todos se vuelven a pasar con éxito.



5. Refactorizar el código para mejorar su diseño (eliminar código duplicado, extraer métodos, renombrar identificadores...)



6. Ejecutar los casos de prueba
para comprobar que todo sigue funcionando correctamente.



7. Volver al paso inicial

Caso práctico: Bolera

| | | | | | | | | | | | | |
|---|----|---|----|----|----|----|---|----|----|-----|---|-----|
| 1 | 4 | 4 | 5 | 6 | 5 | 5 | 0 | 1 | 7 | 6 | 2 | 6 |
| 5 | 14 | | 29 | 49 | 60 | 61 | | 77 | 97 | 117 | | 133 |

El problema consiste en...

obtener la puntuación de un jugador en una partida de bolos.

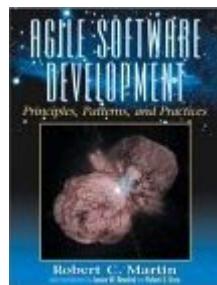
Una posible solución...

y el proceso seguido para obtenerla en

“*The Bowling Game. An example of test-first pair programming*”

© Robert C. Martin & Robert S. Koss, 2001

<http://www.objectmentor.com/resources/articles/xpepisode.htm>



Robert C. Martin:

“*Agile Software Development: Principles, Patterns, and Practices*”.
Prentice Hall, 2003. ISBN 0-13-597444-5.

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:

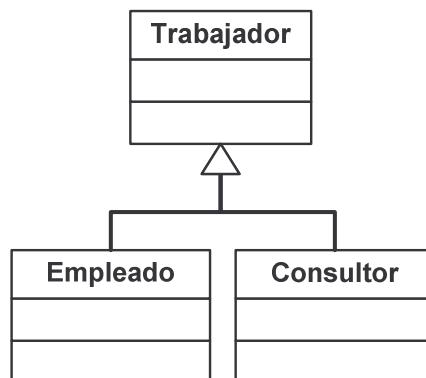
Se crean tantos objetos de la clase como nos haga falta.
Cada objeto proporcionará un servicio que podrá ser
utilizado por otros objetos de nuestro sistema.

Herencia

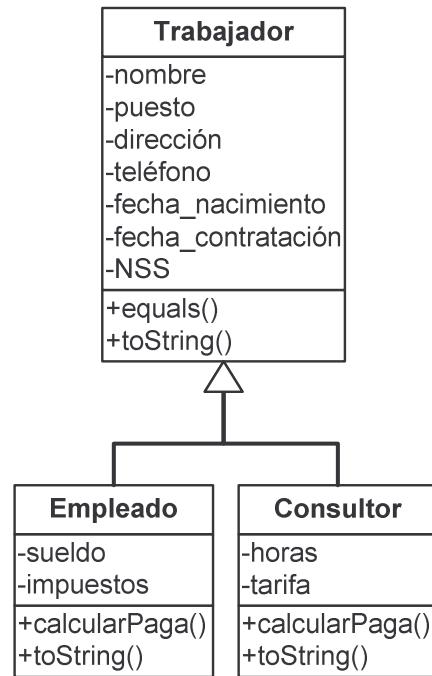
Hay clases que comparten gran parte de sus características.

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.

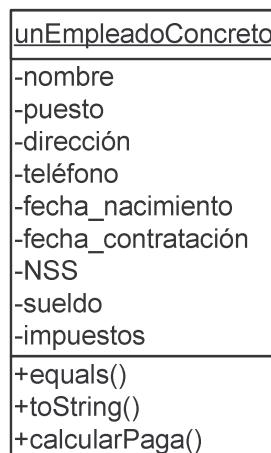
- La nueva clase, a la que se denomina **subclase**, puede poseer atributos y métodos que no existan en la clase original.
- Los objetos de la nueva clase **heredan** los atributos y los métodos **de la** clase original, que se denomina **superclase**.



- § 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 + ")";
    }
}
```

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.

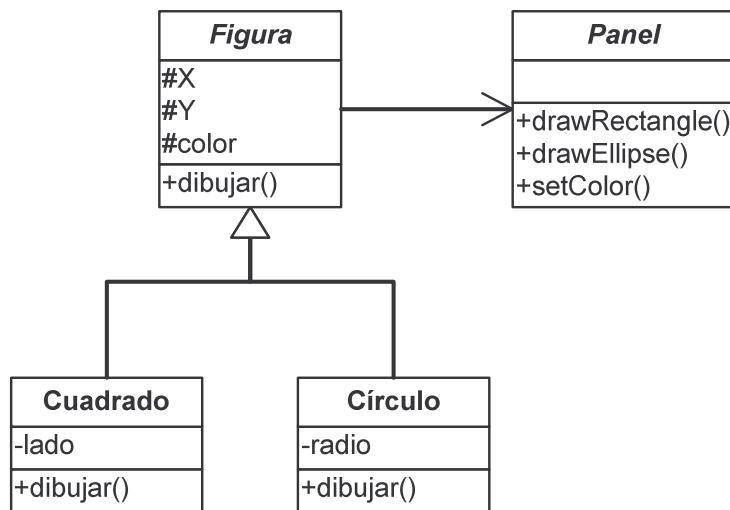
Acerca de la sobrecarga de métodos

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;
    }

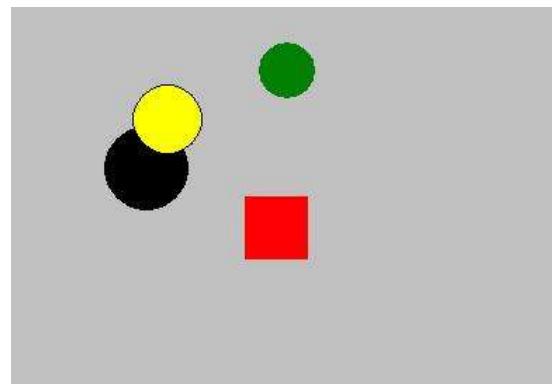
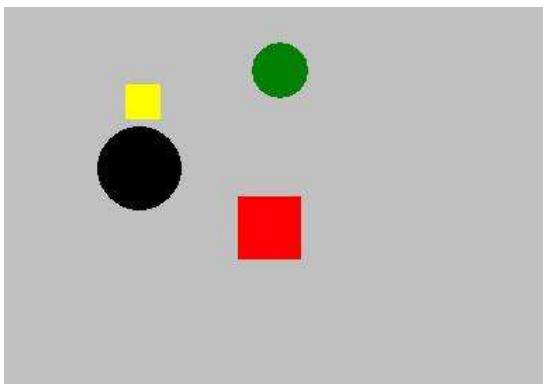
    public void dibujar ()
    {
        panel.drawEllipse(x,y, x+2*radio, y+2*radio);
    }
}

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 creamos 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.

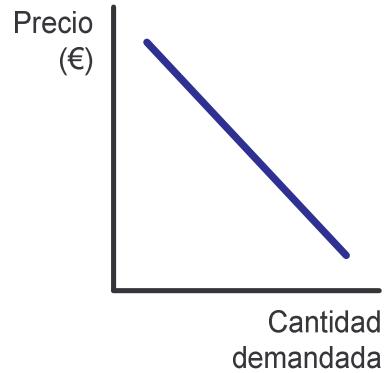
```
package economics;

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 <code>import</code> 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 [<i>Remote Method Invocation</i>]. |
| 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*

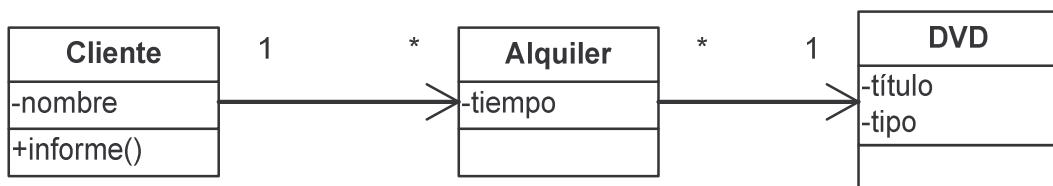
Adaptado de “Refactoring” © Martin Fowler, 2000



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 XIIII", 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);
}

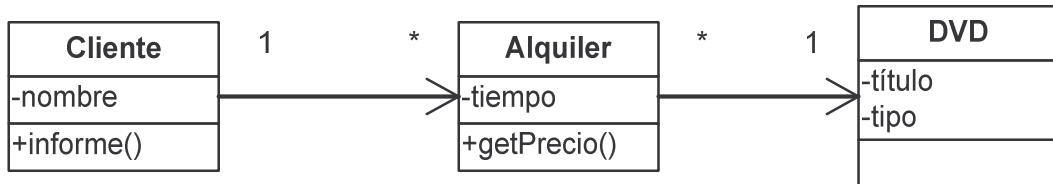
public void testInfantil4 ()
{
    alquiler = new Alquiler(shrek,4);
    assertEquals( cliente.precio(alquiler), 3.0, 0.001);
}

public void testInfantil7 ()
{
    alquiler = new Alquiler(shrek,7);
    assertEquals( cliente.precio(alquiler), 7.5, 0.001);
}
}

```

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:

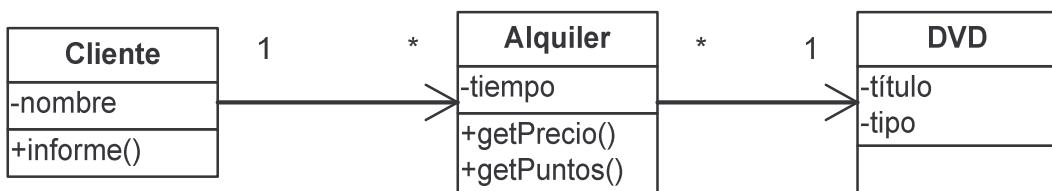
p.ej.

```
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.

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;
    }
}
  
```

```

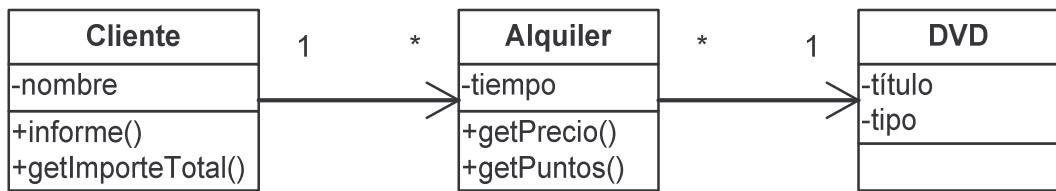
public class Cliente
{
    ...
    public String informe()
    {
        ...
        for (i=0; i<_alquileres.size(); i++) {
            alquiler = (Alquiler) _alquileres.get(i);
            importe = alquiler.getPrecio();
            puntos += alquiler.getPuntos();

            ...
        }
        ...
    }
}
  
```

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...

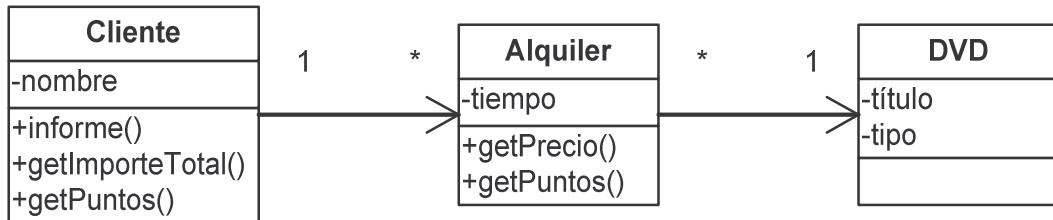
public double getImporteTotal ()
{
    int      i;
    double   total;
    Alquiler alquiler;

    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>";

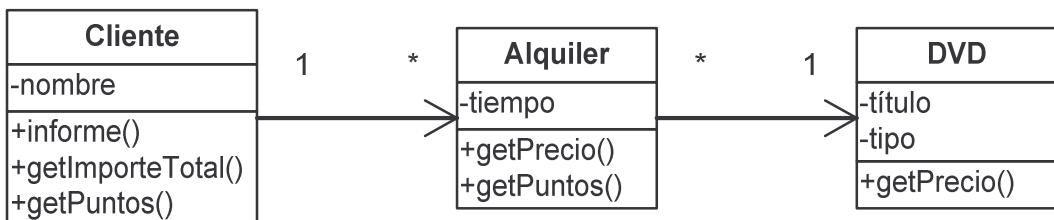
    salida += "<P>IMPORTE TOTAL = "
              + "<B>" + getImporteTotal() + " €</B>\n";

    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;

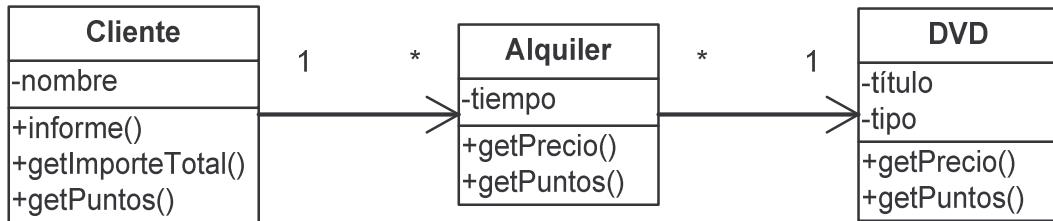
        case DVD.INFANTIL:
            importe += 1.5;
            if (tiempo>3)
                importe += (tiempo-3) * 1.5;
            break;
    }
    return importe;
}
```

```
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;
}
```

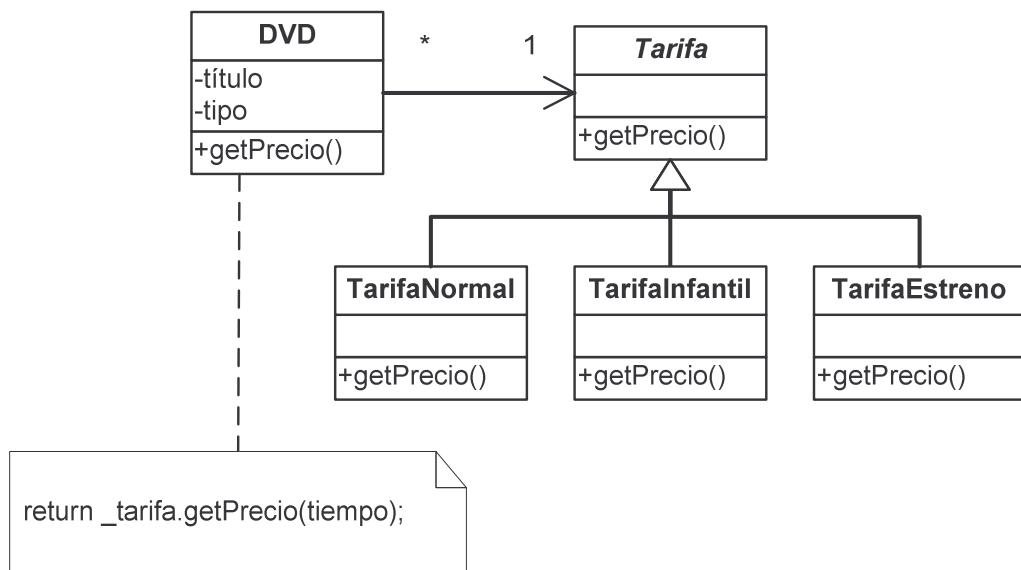
```
public class Alquiler...
```

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

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

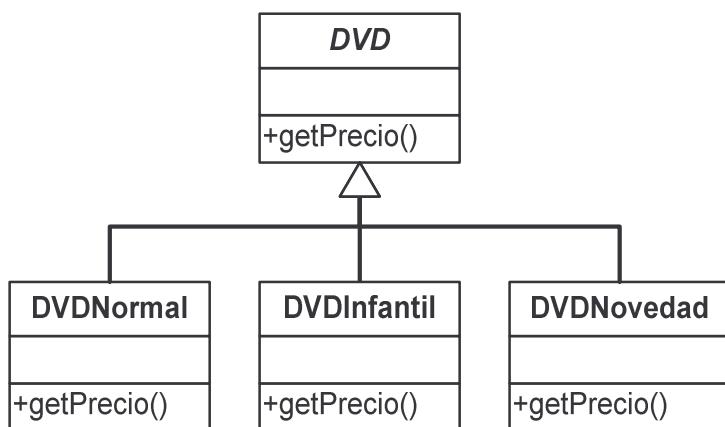
Paso 8: Reemplazar lógica condicional con polimorfismo

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;

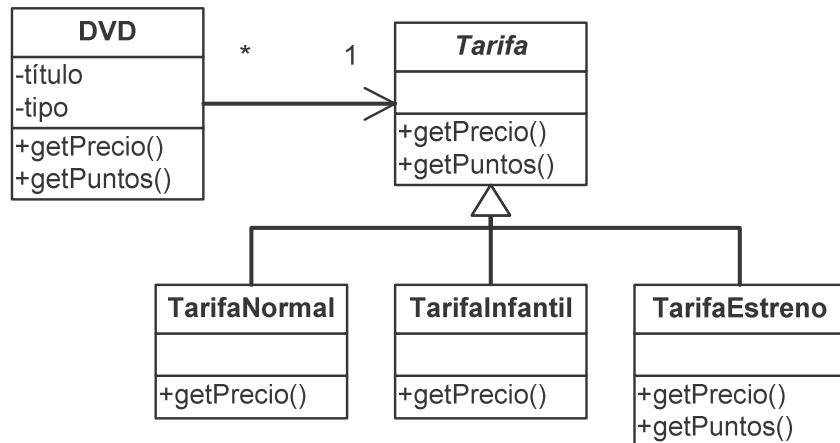
        return importe;
    }

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...



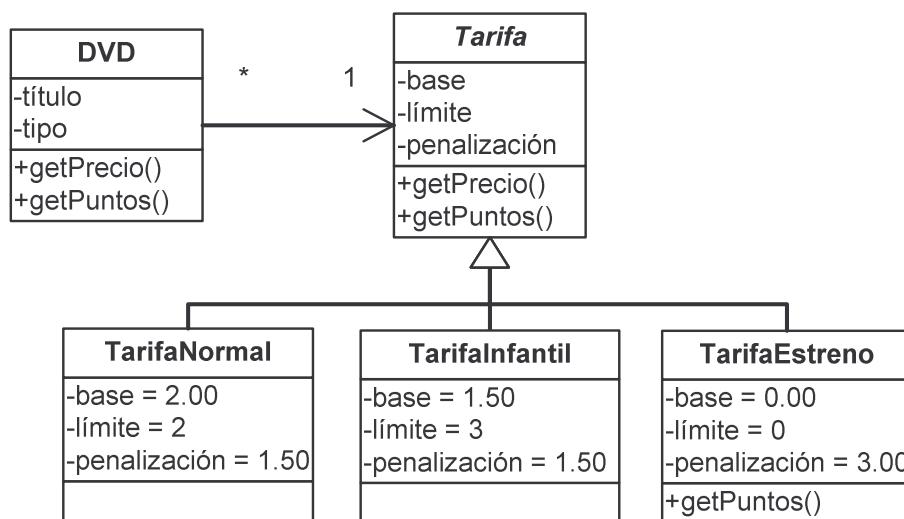
Las constantes simbólicas definidas en la clase DVD son, en realidad, meros identificadores que utilizamos para las diferenciar las distintas políticas de precios, por lo que deberíamos pasarlas a la clase genérica Tarifa.



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).



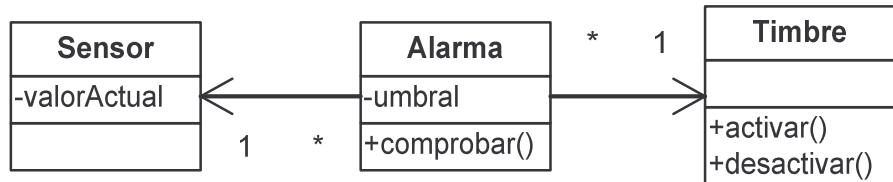
Clases y objetos

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:

- 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).
- 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.
- 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:

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

Principios de programación orientada a objetos

Diseño de clases

- Síntomas de un mal diseño
- El principio de responsabilidad única
- El principio abierto-cerrado
- El principio de sustitución de Liskov
- El principio de inversión de dependencias

Clases abstractas e interfaces

- El principio de segregación de interfaces
- Ejemplo: Banca electrónica

Diseño de paquetes

- Granularidad
- Estabilidad

Bibliografía

Robert C. Martin:

“*Agile Software Development: Principles, Patterns, and Practices*”.
Prentice Hall, 2003. ISBN 0-13-597444-5.
<http://www.objectmentor.com/resources>

Diseño de clases

¿Cómo sabemos si nuestro diseño es correcto?

Existen algunos síntomas que nos indican que el diseño de un sistema es bastante pobre:

- **RIGIDEZ** (las clases son difíciles de cambiar)
- **FRAGILIDAD** (es fácil que las clases dejen de funcionar)
- **INMOVILIDAD** (las clases son difíciles de reutilizar)
- **VISCOSIDAD** (resulta difícil usar las clases correctamente)
- **COMPLEJIDAD INNECESARIA** (sistema “sobrediseñado”)
- **REPETICIÓN INNECESARIA** (abuso de “copiar y pegar”)
- **OPACIDAD** (aparente desorganización)

Afortunadamente, también existen algunos principios heurísticos que nos ayudan a eliminar los síntomas arriba enumerados:

 **Principio de responsabilidad única**

 **Principio abierto-cerrado**

 **Principio de sustitución de Liskov**

 **Principio de inversión de dependencias**

 **Principio de segregación de interfaces**

Descripción de los síntomas de un diseño “mejorable”

Rigidez

El sistema es difícil de cambiar porque cualquier cambio, por simple que sea, fuerza otros muchos cambios en cascada.

¿Por qué es un problema?

“Es más complicado de lo que pensé”

Cuando nos dicen que realicemos un cambio, puede que nos encontremos con que hay que hacer más cosas de las que, en principio, habíamos pensado que harían falta.

Cuantos más módulos haya que tocar para hacer un cambio, más rígido es el sistema.

Fragilidad

Los cambios hacen que el sistema deje de funcionar (incluso en lugares que, aparentemente, no tienen nada que ver con lo que hemos tocado).

¿Por qué es un problema?

Porque la solución de un problema genera nuevos problemas...

Conforme la fragilidad de un sistema aumenta, la probabilidad de que un cambio ocasione nuevos quebraderos de cabeza también aumenta.

Inmovilidad

La reutilización de componentes requiere demasiado esfuerzo.

¿Por qué es un problema?

Porque la reutilización es siempre la solución más rápida.

Aunque un diseño incorpore elementos potencialmente útiles, su inmovilidad hace que sea demasiado difícil extraerlos.

Viscosidad

Es más difícil utilizar correctamente lo que ya hay implementado que hacerlo de forma incorrecta (e, incluso, que reimplementarlo).

¿Por qué es un problema?

Cuando hay varias formas de hacer algo, no siempre se elige la mejor forma de hacerlo y el diseño tiende a degenerarse.

La alternativa más evidente y fácil de realizar ha de ser aquélla que preserve el estilo del diseño.

Complejidad innecesaria

El diseño contiene infraestructura que no proporciona beneficios.

¿Por qué es un problema?

A veces se anticipan cambios que luego no se producen, lo que conduce a *más código* (que hay que depurar y mantener)

La complejidad innecesaria hace que el software sea más difícil de entender.

Repetición innecesaria

“Copiar y pegar” resulta perjudicial a la larga.

¿Por qué es un problema?

Porque el mantenimiento puede convertirse en una pesadilla.

El código duplicado debería unificarse bajo una única abstracción.

Opacidad

Código enrevesado difícil de entender.

¿Por qué es un problema? Porque la “entropía” del código tiende a aumentar si no se toman las medidas oportunas.

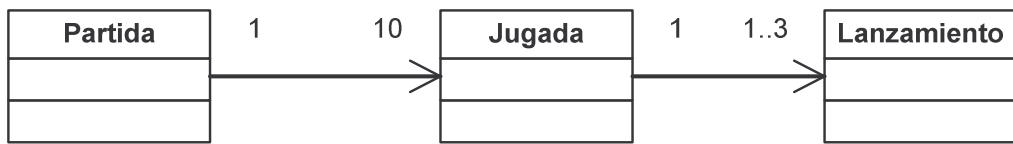
Escribimos código para que otros puedan leerlo (y entenderlo).

El principio de responsabilidad única

Tom DeMarco, 1979: “*Structured Analysis and System Specification*”

Una clase debe tener un único motivo para cambiar.

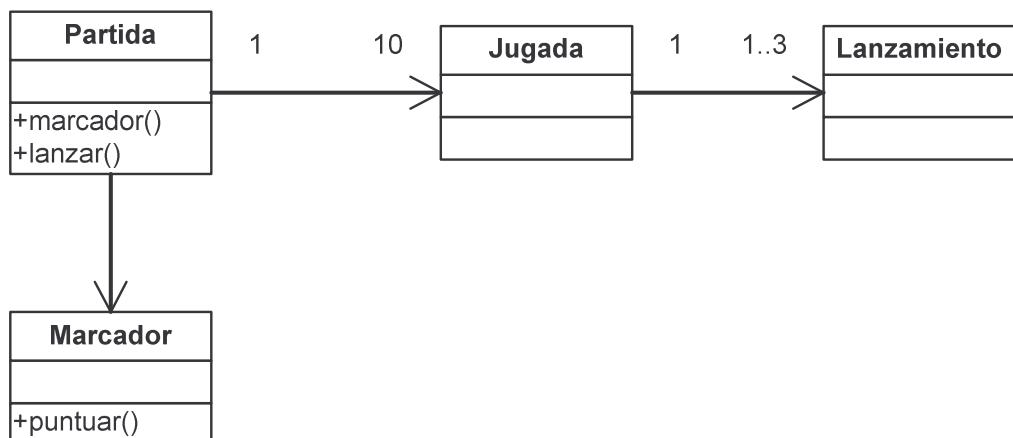
Por ejemplo, podemos crear una clase Partida para representar una partida de bolos (que consta de 10 jugadas):



La clase Partida tiene aquí dos responsabilidades diferentes:

- Mantener el estado actual de la partida (esto es, cuántos lanzamientos llevamos realizados y cuántos nos quedan)
- Calcular nuestra puntuación (siguiendo las reglas oficiales del juego: 10 puntos por pleno [*strike*], etc.).

En una situación así, puede interesarnos separar el cálculo de la puntuación del mantenimiento del estado de la partida:



De esta forma, podríamos utilizar nuestra clase Partida para distintas variantes del juego (si es que existiesen) o incluso para otros juegos (¿la petanca?).

En este contexto,
una responsabilidad equivale a “una razón para cambiar”.

Si se puede pensar en más de un motivo por el que cambiar una clase,
entonces la clase tiene más de una responsabilidad.

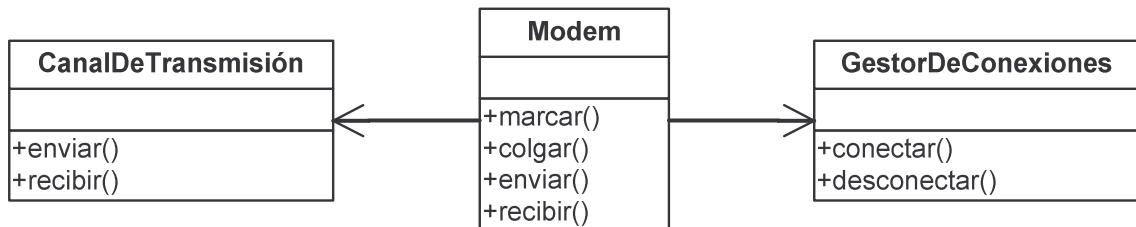
Ejemplo

```
public class Modem
{
    public void marcar (String número) ...
    public void colgar () ...

    public void enviar (String datos) ...
    public String recibir () ...
}
```

La clase Modem tiene aquí dos responsabilidades:

- Encargarse de la gestión de las conexiones.
- Enviar y recibir datos.



Modem sigue teniendo dos responsabilidades y sigue siendo necesaria en nuestra aplicación, pero ahora nada depende de Modem:
Sólo el programa principal ha de conocer la existencia de Modem.

Si la forma de hacerse cargo de una de las responsabilidades sabemos que nunca varía, no hay necesidad de separarla (se trataría de complejidad innecesaria en nuestro diseño):

Una razón de cambio sólo es una razón de cambio
si el cambio llega a producirse realmente.

El principio abierto-cerrado

Bertrand Meyer, 1997: “Construcción de software orientado a objetos”

**Los módulos deben ser a la vez abiertos y cerrados:
abiertos para ser extendidos y cerrados para ser usados.**

1. Debe ser posible extender un módulo para ampliar su conjunto de operaciones y añadir nuevos atributos a sus estructuras de datos.
2. Un módulo ha de tener un interfaz estable y bien definido (que no se modificará para no afectar a otros módulos que dependen de él).

Si se aplica correctamente este principio, los cambios en un sistema se realizan siempre añadiendo código, sin tener que modificar la parte del código que ya funciona.

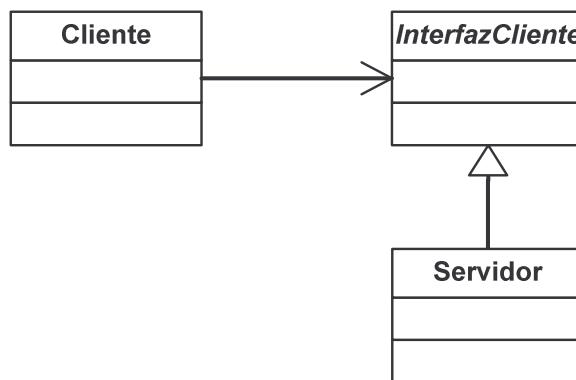
El sencillo diseño siguiente no se ajusta al principio abierto-cerrado:



- ⌘ El Cliente usa los servicios de un Servidor concreto, al cual está ligado. Si queremos cambiar el tipo de servidor, hemos de cambiar el código del cliente.

Aunque pueda resultar paradójico, se puede cambiar lo que hace un módulo sin cambiar el módulo: la clave es la **abstracción**.

El siguiente diseño sí se ajusta al principio abierto-cerrado:



- ü *InterfazCliente* es una clase base utilizada por el cliente para acceder a un servidor concreto: Los clientes acceden a servidores concretos que implementen el interfaz definida por la clase *InterfazCliente*.
- ü Si queremos que un cliente utilice un servidor diferente, basta con crear una clase nueva que también implemente la interfaz de *InterfazCliente* (esto es, la clase *Cliente* no habrá que modificarla).
- ü El trabajo que tenga que realizar el *Cliente* puede describirse en función del interfaz genérico expuesto por *InterfazCliente*, sin recurrir a detalles de servidores concretos.

¿Por qué *InterfazCliente* y no *ServidorGenérico*?

Porque las clases abstractas (más sobre ellas más adelante) están más íntimamente relacionadas a sus clientes que a las clases concretas que las implementan.

El principio abierto-cerrado
es clave en el diseño orientado a objetos.

Si nos ajustamos a él, obtenemos los mayores beneficios
que se suelen atribuir a la orientación a objetos
(flexibilidad, mantenibilidad, reutilización).

No obstante,
pese a que la abstracción y el polimorfismo lo hacen posible,

el principio abierto-cerrado
no se garantiza simplemente con utilizar
un lenguaje de programación orientado a objetos.

De la misma forma,
tampoco tenemos que crear abstracciones arbitrariamente:

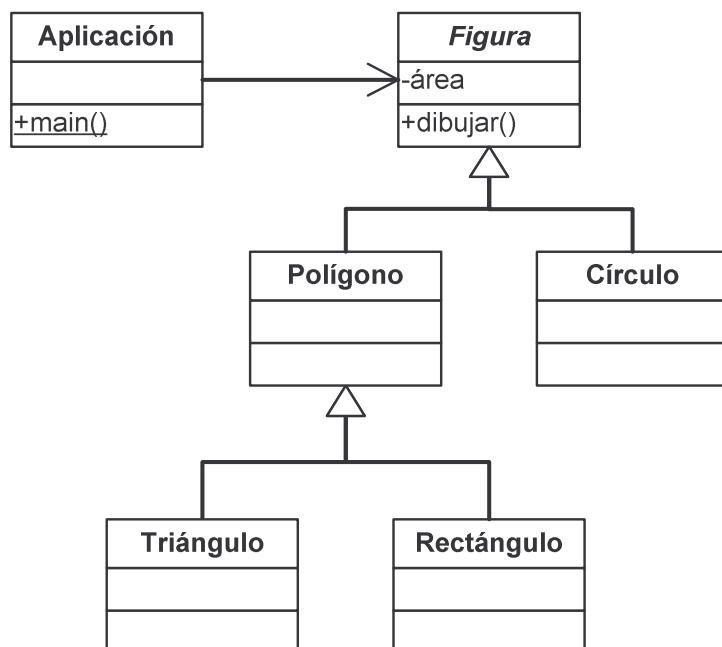
- ü No todo tiene que ser abstracto:
El código debe acabar haciendo algo concreto.
- ü Las abstracciones se han de utilizar
en las zonas que exhiban cierta propensión a sufrir cambios.

El principio de sustitución de Liskov

Barbara Liskov: “*Data abstraction and hierarchy*”. SIGPLAN Notices, 1988

Los tipos base siempre se pueden sustituir por sus subtipos.

*Si S es un subtipo de T ,
cualquier programa definido en función de T
debe comportarse de la misma forma con objetos de tipo S .*



La importancia de este principio se hace evidente cuando deja de cumplirse:

```
public class Figura
{
    ...
    public void dibujar ()
    {
        if (this instanceof Polígono)
            dibujarPolígono();
        else if (this instanceof Círculo)
            dibujarCírculo();
    }
    ...
}
```

El método `dibujar` viola el principio de sustitución de Liskov, porque ha de ser consciente de cualquier subtipo de `Figura` que creemos en nuestro sistema (p.ej. ¿qué sucede si también tenemos que trabajar con elipses?).

De hecho, también se viola el *principio abierto-cerrado*.

La solución: Utilizar polimorfismo
Cada tipo de figura será responsable de implementar el método `dibujar`.

```
public class Polígono extends Figura
...
public void dibujar ()
...

public class Círculo extends Figura
...
public void dibujar ()
...
```

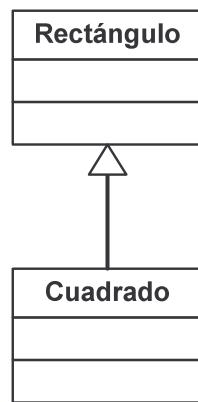
Veamos un caso algo más sutil:
Inicialmente trabajamos con rectángulos...

```
public class Rectángulo
{
    private double anchura;
    private double altura;

    public double getAnchura () ...
    public double getAltura () ...

    public void setAnchura () ...
    public void setAltura () ...
}
```

Por lo que sea, también hemos de manipular cuadrados...



La implementación empieza a crear “dificultades”...

```
public class Cuadrado extends Rectángulo
{
    public void setAnchura (double x)
    {
        super.setAnchura(x);
        super.setAltura(x);
    }

    public void setAltura (double x)
    {
        super.setAnchura(x);
        super.setAltura(x);
    }
}
```

Ahora, consideremos el siguiente método:

```
...
void f (Rectángulo rect)
{
    rect.setAnchura(32);
}
...
```

¿Qué sucede si llamamos a este método con un cuadrado?

La llamada a `setAnchura` establece también la altura del “rectángulo” (algo aparentemente correcto).

Pero, ¿y si el método hace algo diferente?

```
...
void g (Rectángulo rect)
{
    rect.setAnchura(4);
    rect.setAltura(5);
    Assert.assertEquals( rect.getArea() , 20 );
}
...
```

Obviamente, el código anterior fallará si el método recibe un cuadrado en vez de un triángulo L

¿Por qué falla la implementación?

Aunque el modelo del cuadrado como un tipo particular de rectángulo parezca razonable, desde el punto de vista del programador del método `g`, ¡un cuadrado no es un rectángulo!

Las relaciones de herencia se han de utilizar para heredar comportamiento (el comportamiento que los clientes de una clase pueden asumir y en el que confían).

Formalmente,
la interfaz de una clase define su contrato.

El contrato se especifica declarando las precondiciones y las postcondiciones asociadas a cada método (esto es, lo que verificamos con las pruebas de unidad con JUNIT):

- Las **precondiciones** son las condiciones que se han de cumplir antes de la llamada al método.
- Las **postcondiciones** son las condiciones que se verifican tras la ejecución del método.

En el caso de la clase Rectángulo, las postcondición implícita de setAnchura es

```
(anchura == x) && (altura == old.altura)
```

donde old hace referencia al estado del rectángulo antes de la llamada.

Para la clase Cuadrado, no obstante, la postcondición asociada a setAnchura es

```
(anchura == x) && (altura == x)
```

Nuestra implementación de setAnchura para la clase Cuadrado es incorrecta porque impone un postcondición distinta a la impuesta por setAnchura en su clase base Rectángulo.

La redefinición de un método en una clase derivada sólo puede reemplazar la precondición original por una más débil y la postcondición original por otra más fuerte (restrictiva).

¿Por qué? Porque cuando se usa una clase base, sólo se conocen las precondiciones y postcondiciones asociadas a la clase base.

El principio de inversión de dependencias

Robert C. Martin: *C++Report*, 1996

- a) Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
- b) Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

La programación estructurada
tiende a crear estructuras jerárquicas en las que

el comportamiento de los módulos de alto nivel
(p.ej. el programa principal)
depende de detalles de módulos de un nivel inferior
(p.ej. la opción del menú seleccionada por el usuario).

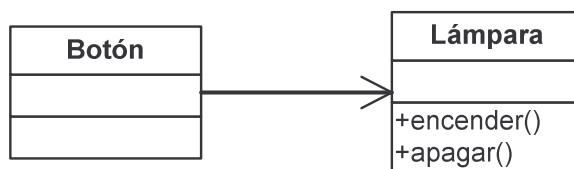
La estructura de dependencias en una aplicación orientada a objetos bien diseñada suele ser la inversa.

Si se mantuviese la estructura tradicional:

- ⌘ Cuando se cambiaseen los módulos de nivel inferior, habría que modificar los módulos de nivel superior.
- ⌘ Además, los módulos de nivel superior resultarían difíciles de reutilizar si dependiesen de módulos inferiores.

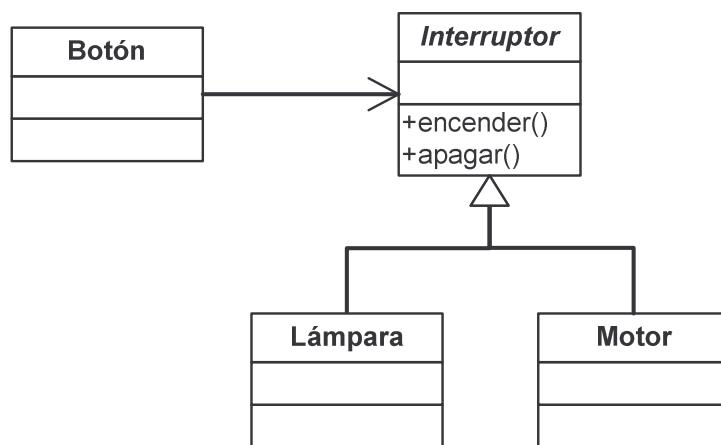
Ejemplo

La inversión de dependencias se puede realizar siempre que una clase envía un mensaje a otra y deseamos eliminar la dependencia



Tal como está diseñado, no será posible utilizar el botón para encender, por ejemplo, un motor (ya que el botón depende directamente de la lámpara que enciende y apaga).

Para solucionar el problema,
volvemos a hacer uso de nuestra capacidad de abstracción:



Ahora podremos utilizar el botón
para cualquier cosa que se pueda encender y apagar.

Clases abstractas e interfaces

Clases abstractas

Una clase abstracta...

es una clase que no se puede instanciar

se usa únicamente para definir subclases

¿Cuándo es una clase abstracta?

En cuanto uno de sus métodos no tiene implementación (en Java, el método abstracto se etiqueta con la palabra reservada `abstract`).

¿Cuándo se utilizan clases abstractas?

Cuando deseamos definir una abstracción que englobe objetos de distintos tipos y queremos hacer uso del polimorfismo.

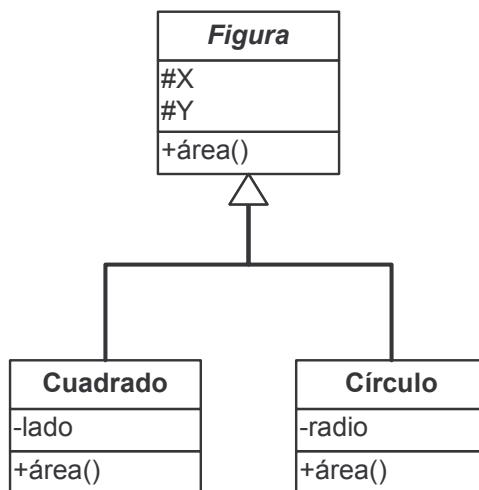


Figura es una clase abstracta (nombre en cursiva en UML) porque no tiene sentido calcular su área, pero sí la de un cuadrado o un círculo. Si una subclase de Figura no redefine `area()`, deberá declararse también como clase abstracta.

```

public abstract class Figura
{
    protected double x;
    protected double y;

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

    public abstract double area ();
}

public class Circulo extends Figura
{
    private double radio;

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

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

public class Cuadrado extends Figura
{
    private double lado;

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

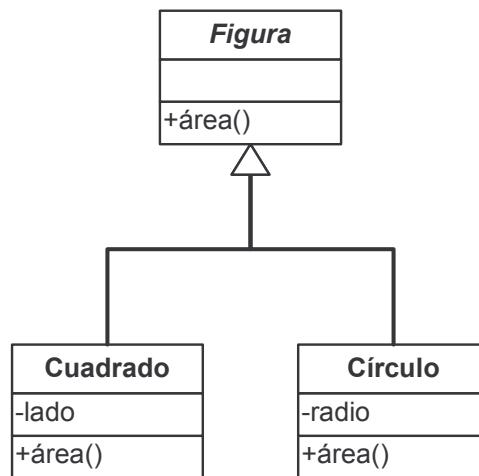
    public double area ()
    {
        return lado*lado;
    }
}

```

Interfaces

**Una interfaz es una clase completamente abstracta
(una clase sin implementación)**

En el ejemplo anterior, si no estuviésemos interesados en conocer la posición de una Figura, podríamos eliminar por completo su implementación y convertir Figura en una interfaz:



```
public interface Figura
{
    public double area ();
}
```

- En Java, las interfaces se declaran con la palabra reservada **interface** de manera similar a como se declaran las clases abstractas.
- En la declaración de una interfaz, lo único que puede aparecer son declaraciones de métodos (su nombre y signatura, sin su implementación) y definiciones de constantes simbólicas.
- Una interfaz no encapsula datos, sólo define cuáles son los métodos que han de implementar los objetos de aquellas clases que implementen la interfaz.

```

public class Circulo implements Figura
{
    private double radio;

    public Circulo (double radio)
    {
        this.radio = radio;
    }

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

public class Cuadrado implements Figura
{
    private double lado;

    public Cuadrado (double lado)
    {
        this.lado = lado;
    }

    public double area ()
    {
        return lado*lado;
    }
}

```

- En Java, para indicar que una clase implementa una interfaz se utiliza la palabra reservada **implements**.
- La clase debe entonces implementar **todos** los métodos definidos por la interfaz o declararse, a su vez, como una clase abstracta (lo que no suele ser especialmente útil):

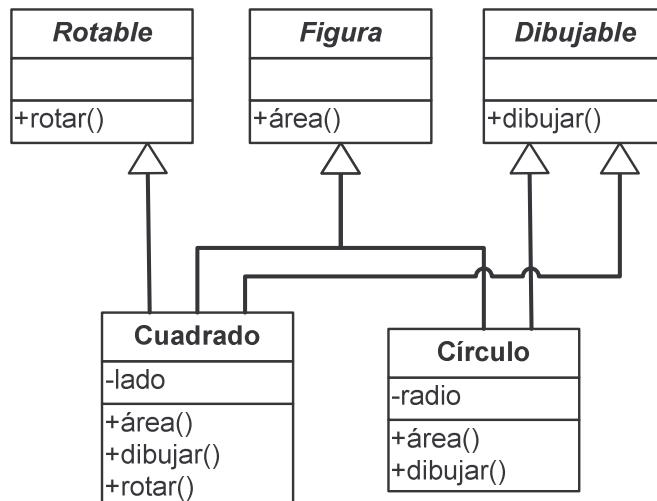
```

abstract class SinArea implements Figura
{
}

```

Herencia múltiple de interfaces

Una clase puede implementar varios interfaces simultáneamente, pese a que, en Java, una clase sólo puede heredar de otra clase (herencia simple de implementación, múltiple de interfaces).



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

public interface Dibujable
{
    public void dibujar ();
}

public interface Rotable
{
    public void rotar (double grados);
}

public class Circulo extends Figura
    implements Dibujable
...
}

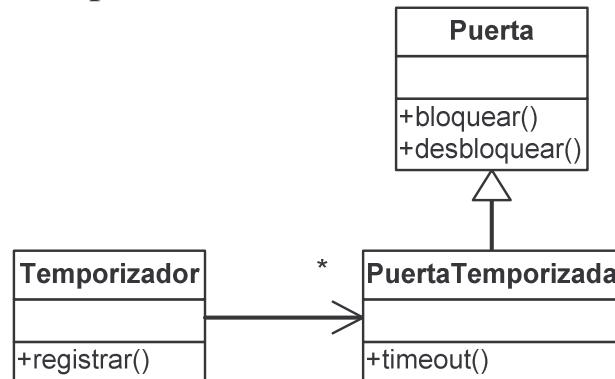
public class Cuadrado extends Figura
    implements Dibujable, Rotable
...
}
```

El principio de segregación de interfaces

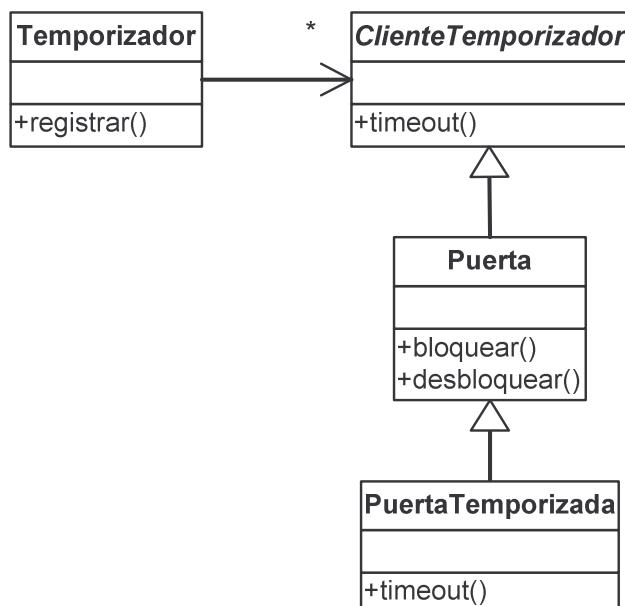
Robert C. Martin: *C++Report*, 1996

Los clientes de una clase no deberían depender de interfaces que no utilizan.

PROBLEMA: Estamos implementando un sistema de seguridad en el cual controlamos varias puertas. Algunas de esas puertas van conectadas a un programa temporizador que las bloquea automáticamente pasado cierto tiempo:



Ahora bien, deseamos que el temporizador controle también otros dispositivos (como una alarma que se dispara automáticamente), por lo que aplicamos el principio de inversión de dependencias:



Como Java sólo permite herencia simple (de implementación),
hemos hecho que Puerta
herede de la clase abstracta ClienteTemporizador
para que PuertaTemporizada
pueda funcionar con un temporizador.

Como consecuencia,
todas las puertas, necesiten o no temporizador,
dependen del interfaz que utilizan los clientes de Temporizador.

Contaminación de la interfaz:

Cuando se añade un método a una clase base
simplemente porque una de sus clases derivadas lo necesita.

- ⌘ La interfaz de la clase base es más compleja de lo que tiene que ser, al incluir métodos que no están directamente relacionados con ella [*complejidad innecesaria*]
- ⌘ Lo que se añade a la clase base tiene que estar disponible en todas sus clases derivadas (y funcionar correctamente, si queremos respetar el *principio de sustitución de Liskov*).
- ⌘ Si tuviésemos que actualizar la interfaz que han de usar los clientes de Temporizador, tendríamos que modificar también la implementación de Puerta [*fragilidad*].

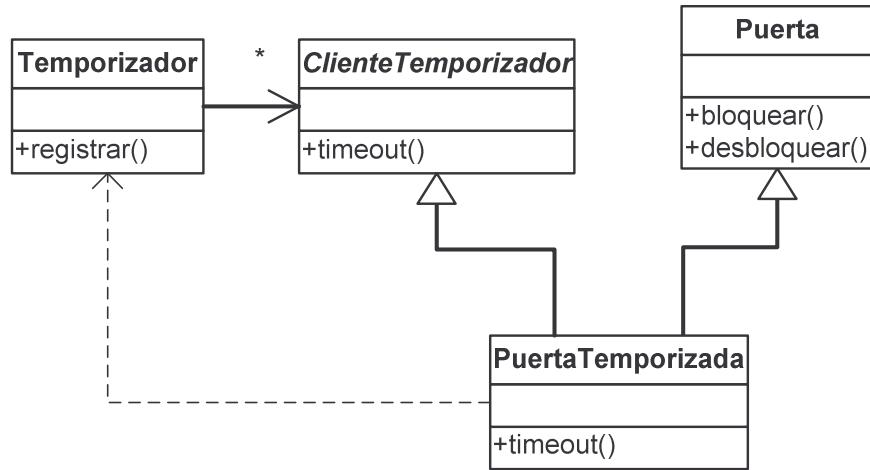
La clave:

Los clientes de un objeto de tipo T
no necesitan una referencia al objeto de tipo T para acceder a él:

- ü basta con una referencia a uno de sus tipos base, o bien
- ü una referencia a un objeto auxiliar que delegue las llamadas necesarias en el objeto original de tipo T.

Separación por herencia múltiple

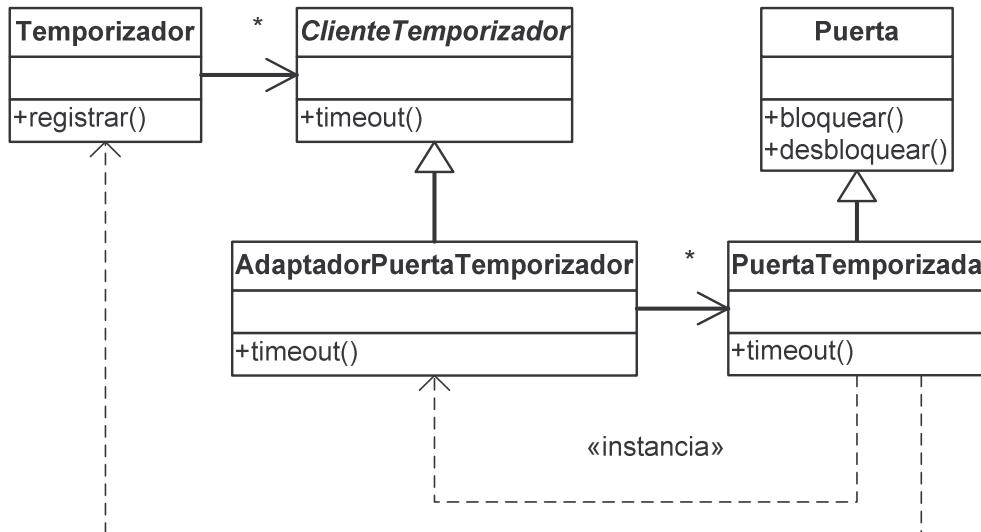
(cuando todas las clases base menos una son interfaces)



```
class PuertaTemporizada extends Puerta
    implements ClienteTemporizador
...
...
```

Separación por delegación

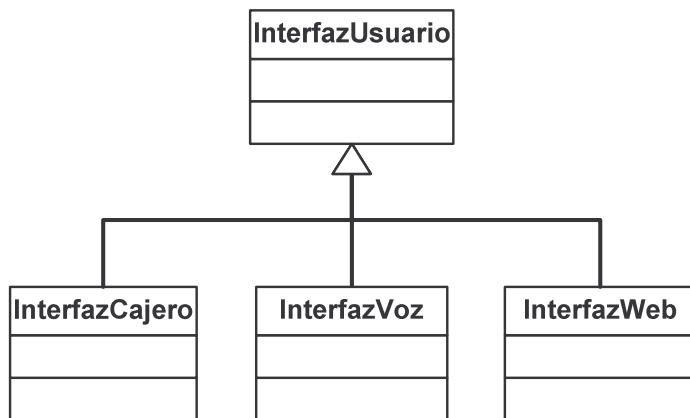
(cuando habría que heredar simultáneamente de varias clases)



```
class PuertaTemporizada extends Puerta
...
...
adaptador = new AdaptadorPT (this);
temporizador.registrar(adaptador);
```

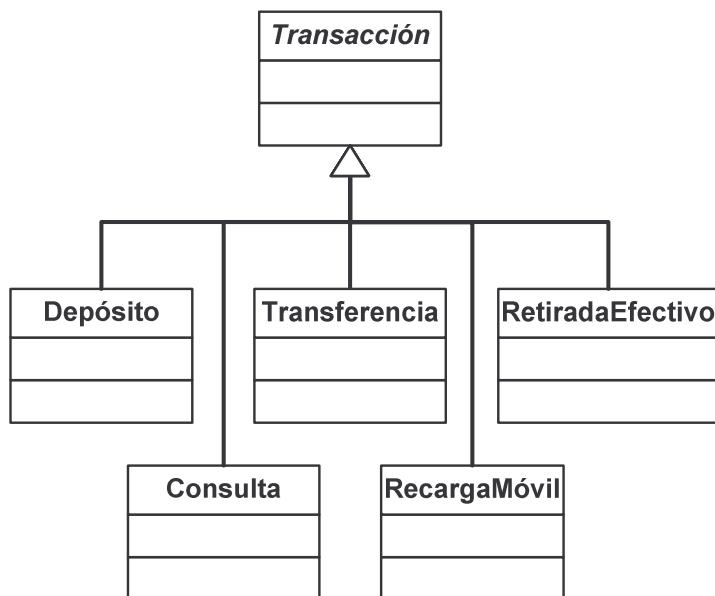
Ejemplo: Banca electrónica

Un sistema de banca electrónica debe tener varias interfaces de usuario para que los clientes del banco puedan efectuar operaciones (en un cajero automático, por teléfono y por Internet)



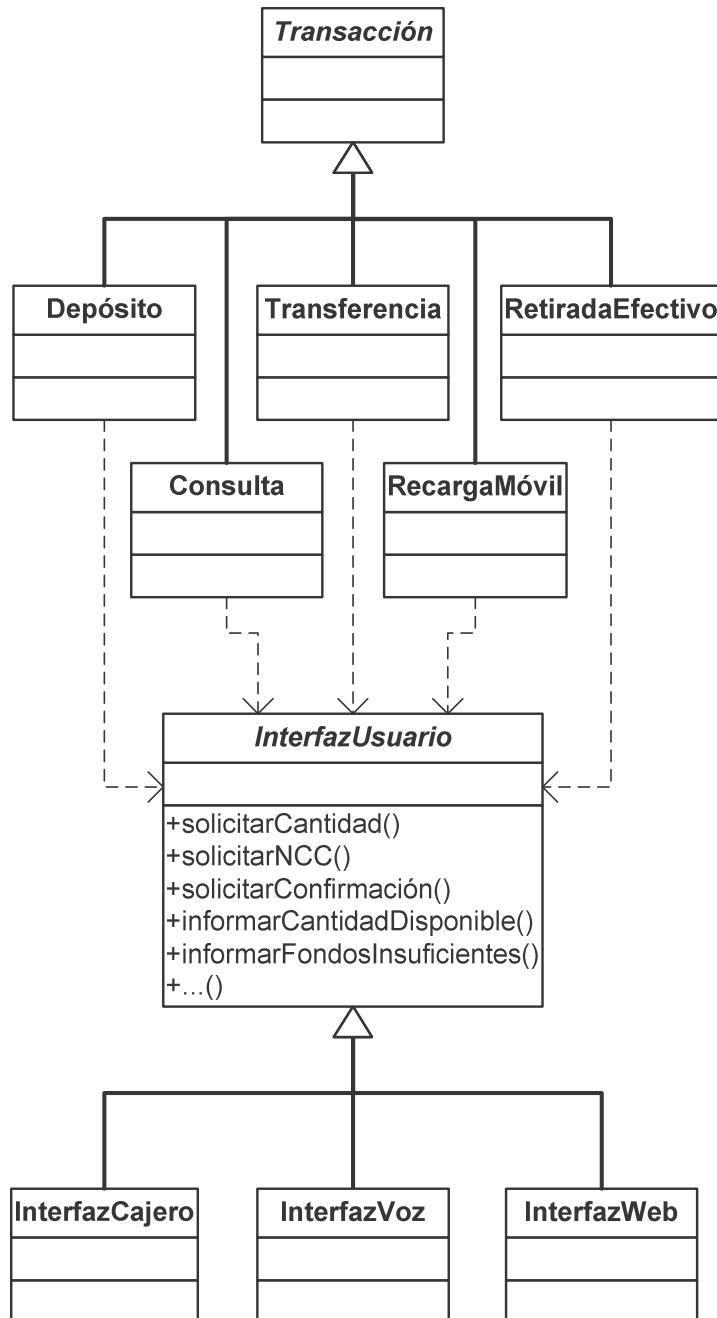
Sea cual sea el medio escogido, el cliente ha de poder ejecutar varias transacciones:

- Realizar depósitos (ingresar dinero en su cuenta).
- Retirar efectivo (sacar dinero de su cuenta).
- Realizar transferencias de dinero.
- Consultar el saldo y los últimos movimientos.
- Recargar el móvil



Solución A

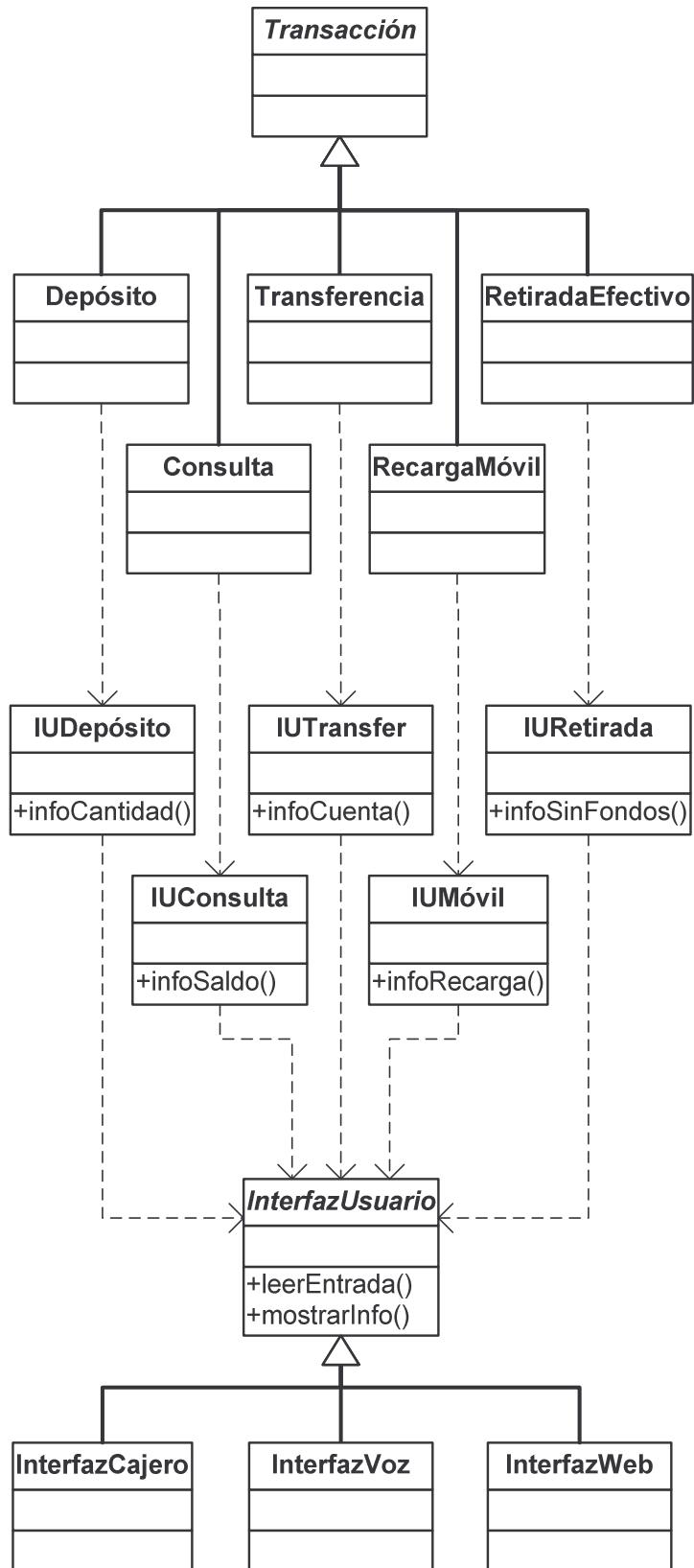
Interfaz “contaminado”



Cada tipo de transacción utiliza algunos métodos que las demás transacciones no siempre utilizan...

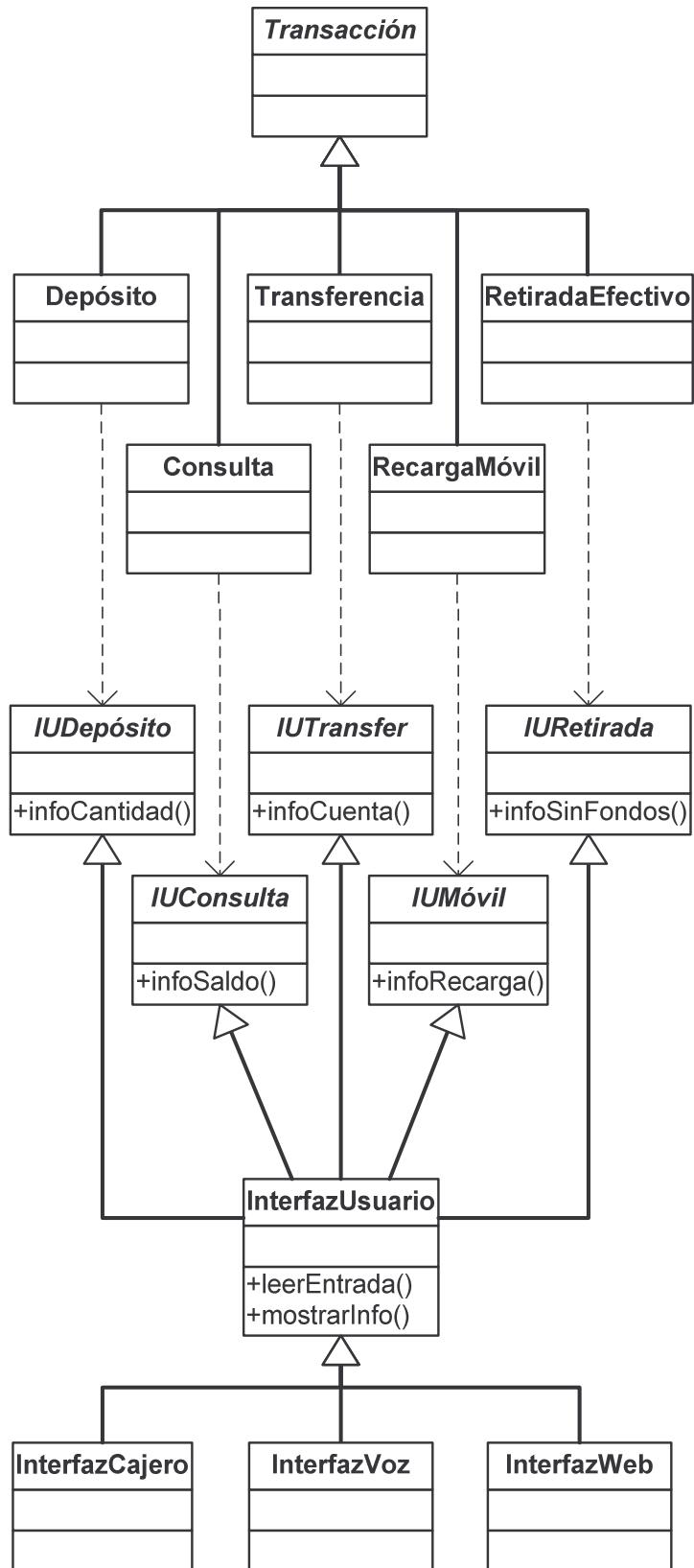
Un nuevo tipo de transacción introducirá cambios en `InterfazUsuario` que afectarán al funcionamiento de los demás tipos de transacciones.

Solución B Adaptadores



Solución C

Segregación de interfaces



Diseño de paquetes

Conforme el tamaño de las aplicaciones crece, se hace necesario algún tipo de organización a alto nivel.

Las clases son unidades demasiado pequeñas, por lo que se agrupan en paquetes.

¿Qué criterios utilizaremos para organizar las clases?

Los mismos que para organizar los miembros de las clases:

la cohesión y el acoplamiento

Granularidad:

La cohesión de los paquetes

¿Cuándo se ponen dos clases en el mismo paquete?

- Cuando para usar una, siempre es **necesario** usar la otra.
- Cuando dos clases están en un mismo paquete, es porque se usan juntas. Por tanto, todas las clases de un paquete se usan conjuntamente: si se usa una, se usan todas.

A la inversa, *¿cuándo se ponen en paquetes diferentes?*

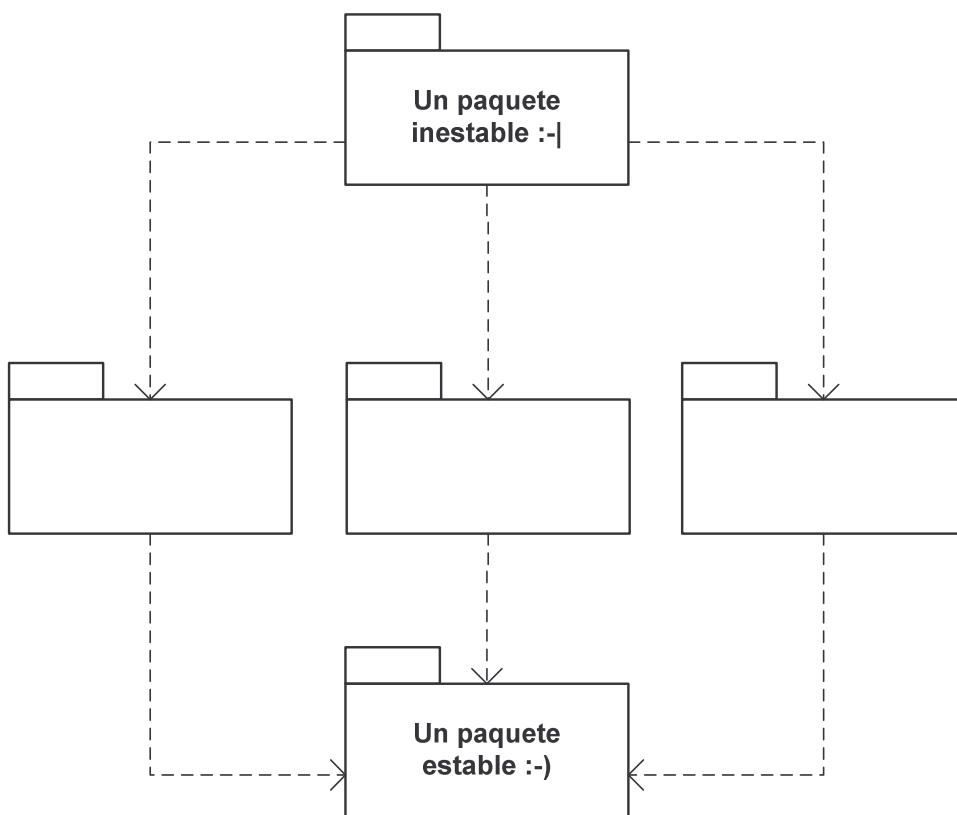
- Aquellas clases que no siempre se usen conjuntamente con las demás clases del paquete son candidatas para abandonar el paquete (quizá para ir a un subpaquete más específico).

Las clases de un paquete se verán afectadas por los mismos tipos de cambios y las modificaciones necesarias para realizar un cambio concreto deberán estar localizadas en un único paquete.

Estabilidad: El acoplamiento entre paquetes

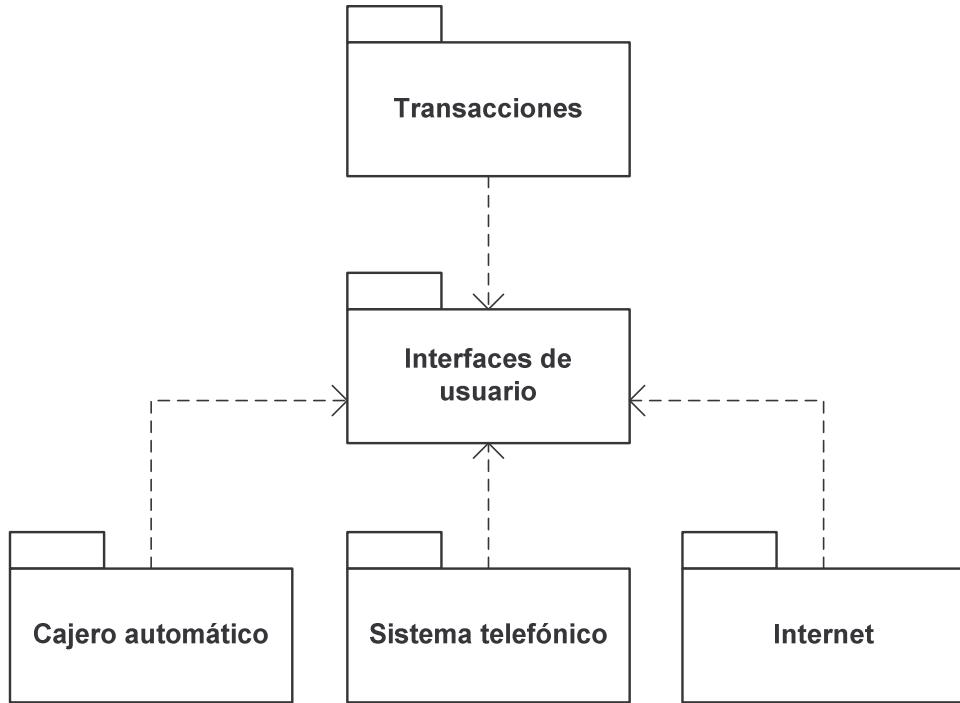
Para facilitar el desarrollo de un sistemas complejo, resulta aconsejable que las distintas partes del sistema sean lo más independientes posible:

Cuanto menos dependa un paquete de otros paquetes, mejor.



- Un paquete es más estable cuando depende de menos paquetes.
- Cuando queramos que un paquete sea flexible (esto es, fácil de cambiar), mejor si hay pocos paquetes que dependan de él.
- Un paquete debe ser más estable cuanto más abstracto sea (un paquete estable y concreto se vuelve rígido).

Si dibujamos un diagrama con las dependencias existentes entre los paquetes, el diagrama **no debe tener ciclos**:

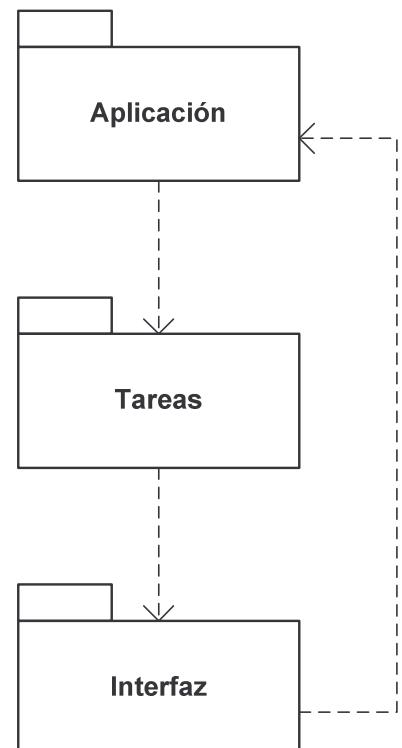


¿Qué efectos ocasiona un ciclo?

Cuando trabajamos en la construcción de la interfaz, hemos de disponer de ciertos servicios proporcionados por la aplicación.

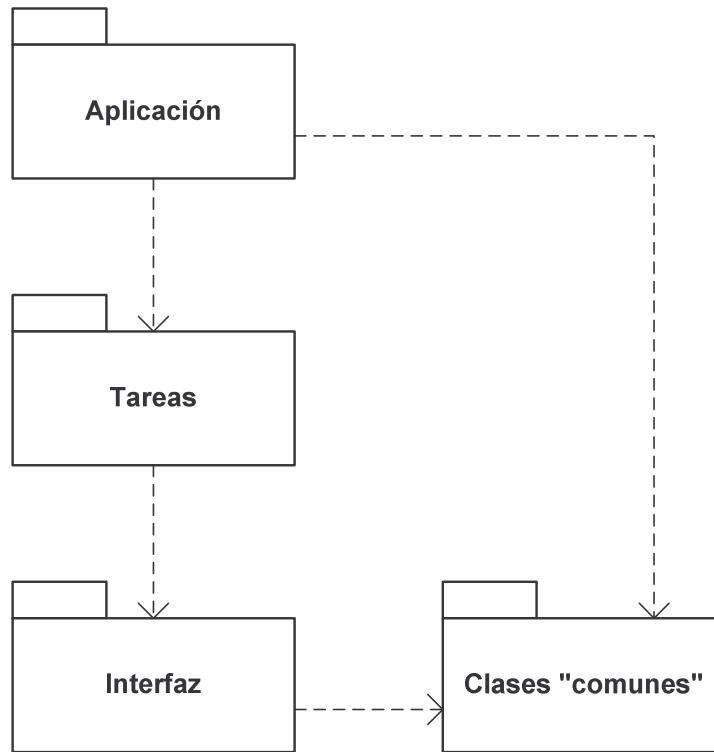
Esto hace que la interfaz dependa de todos los demás paquetes de la aplicación

Para probar el funcionamiento de la interfaz necesitamos disponer de una implementación de todos los demás paquetes y las pruebas dependerán del estado actual de esos paquetes (por lo que difícilmente se pueden considerar *pruebas de unidad*).

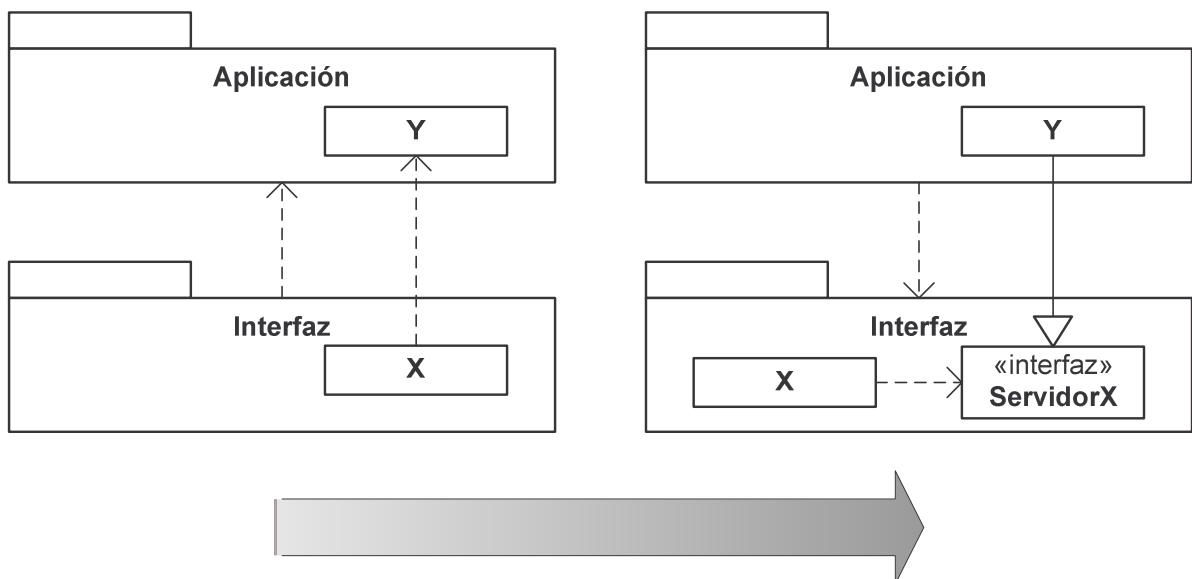


¿Cómo se rompe un ciclo entre dos paquetes?

1. Creando un nuevo paquete del que ambos dependan:



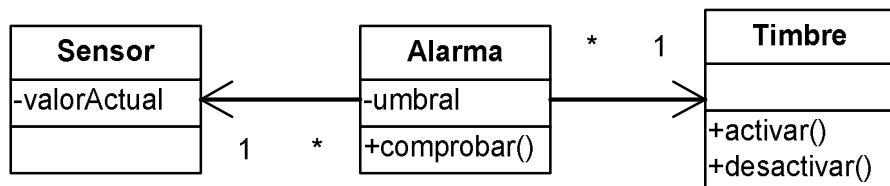
2. Aplicando el *principio de inversión de dependencias*:



Programación orientada a objetos

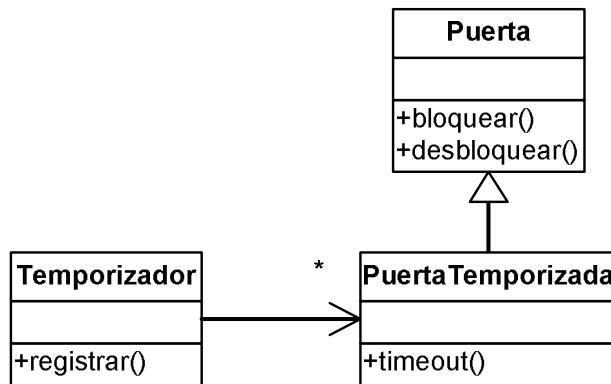
Relación de ejercicios

1. En las jerarquías de clases que haya creado hasta ahora, analice detenidamente si alguna de sus clases ha de ser abstracta. En particular, fíjese en la existencia de métodos que no deberían implementarse en una clase base y habrían de declararse, por tanto, como métodos abstractos. ¿Alguna de las clases que obtiene es completamente abstracta y debería convertirse en una interfaz?
2. Al estudiar clases y objetos, vimos cómo diseñar una alarma que conectábamos a un sensor y activaba un timbre:



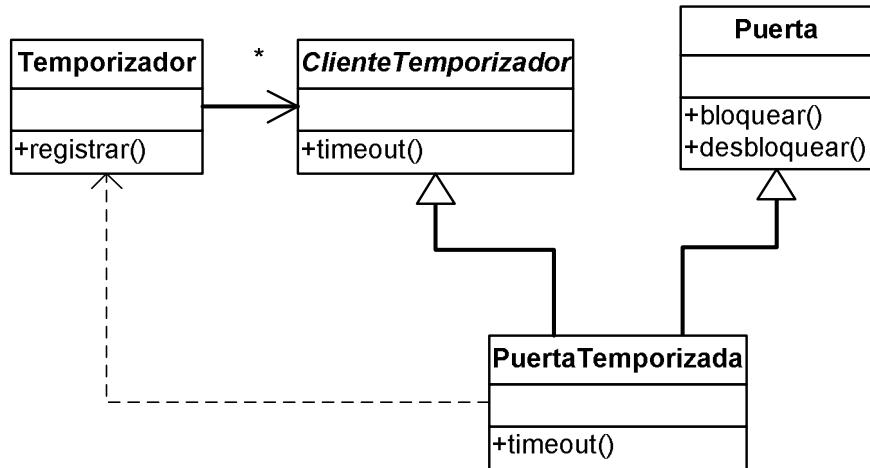
Si queríamos que la alarma también encendiese una señal luminosa, lo que hacíamos era crear una subclase de Alarma. La herencia y el polimorfismo nos permite crear distintos tipos de alarmas que comparten parte de su comportamiento con la clase base Alarma. Ahora bien, si queremos añadir nuevos dispositivos conectados a la alarma (p.ej. un teléfono que automáticamente llama a la policía) y queremos que la alarma se pueda configurar de forma flexible, ¿se le ocurre alguna forma de hacerlo usando interfaces? Implemente la solución en Java.

3. Implemente en Java el sistema de seguridad asociado a una puerta con temporizador, tal como se muestra en la siguiente figura:

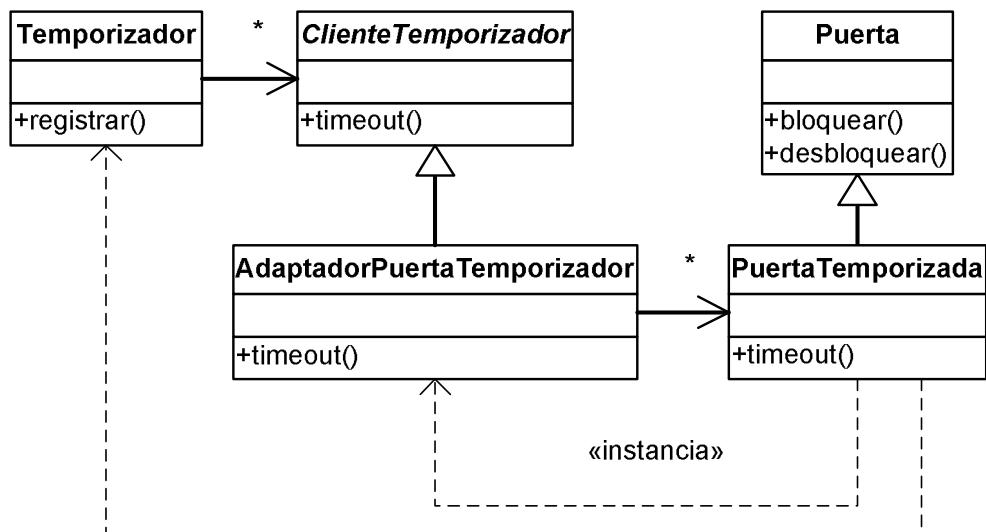


4. Generalice el diseño anterior para que el temporizador pueda activar distintos dispositivos aplicando el principio de segregación de interfaces:

- a. Usando herencia múltiple (de interfaces)



- b. Usando delegación (mediante un adaptador)

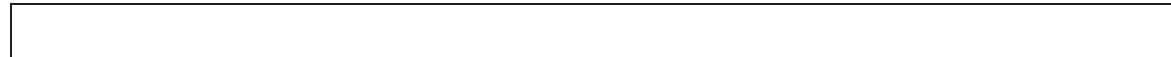


Implemente en Java todo el código asociado a los dos diseños propuestos.

A continuación, implemente en Java, utilizando las dos variantes descritas, el código necesario para que, por ejemplo, el temporizador cierre active automáticamente el sistema de riego del césped y, simultáneamente, cierre las persianas de la casa que dan al jardín...

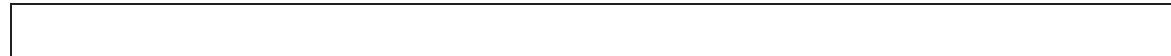
Caso práctico

Red de telefonía móvil



Vamos a analizar la arquitectura software de una red de telefonía móvil de tercera generación UMTS (Universal Mobile Telecommunications System),

NOTA: La tecnología UMTS sustituirá a la tecnología GSM (Global System for Mobile Communications)



Material adaptado de

Michael Kircher & Prashant Jain:
“*Pattern-oriented Software Architecture*”
Volume 3: Patterns for Resource Management
John Wiley & Sons, 2004
ISBN 0-470-84525-2



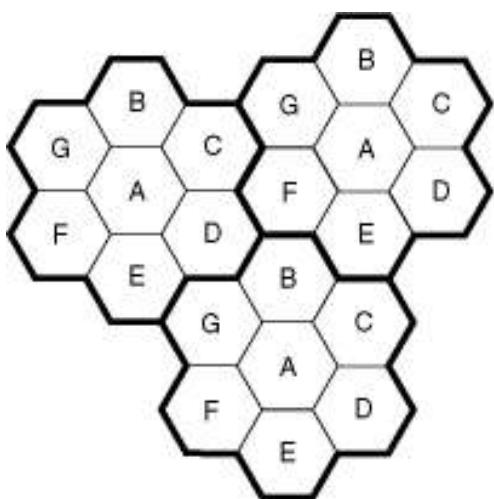
Ilustraciones y figuras cortesía de
Michael Kircher (Siemens AG Corporate Technology, Munich)

El funcionamiento de una red de telefonía móvil

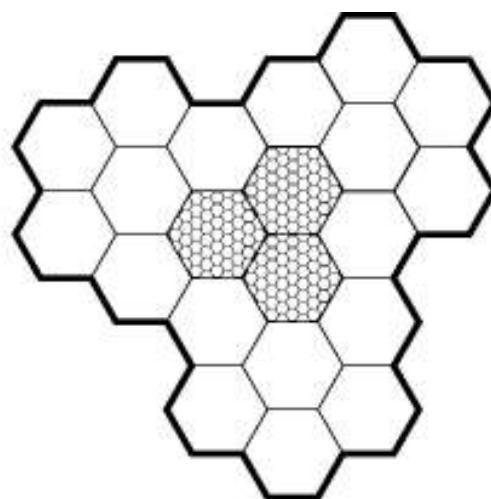
Generaciones de teléfonos móviles

| Generación | Características | Ejemplo |
|------------|------------------------------------|-----------------|
| Primera | Transmisión analógica de voz | Moviline |
| Segunda | Transmisión digital de voz | GSM900 |
| Tercera | Transmisión digital de voz y datos | UMTS |

Redes celulares de telefonía móvil



(a)



(b)

- Las frecuencias no se reutilizan en celdas adyacentes.
- Para dar servicio a más usuarios, se utilizan celdas de menor tamaño.

Los elementos de una red de telefonía móvil

1. Estaciones base (nodos B)
2. Controladores de la red de radio
(RNC: Radio Network Controller)
3. Centros de operación y mantenimiento
(OMC: Operation and Maintenance Center)

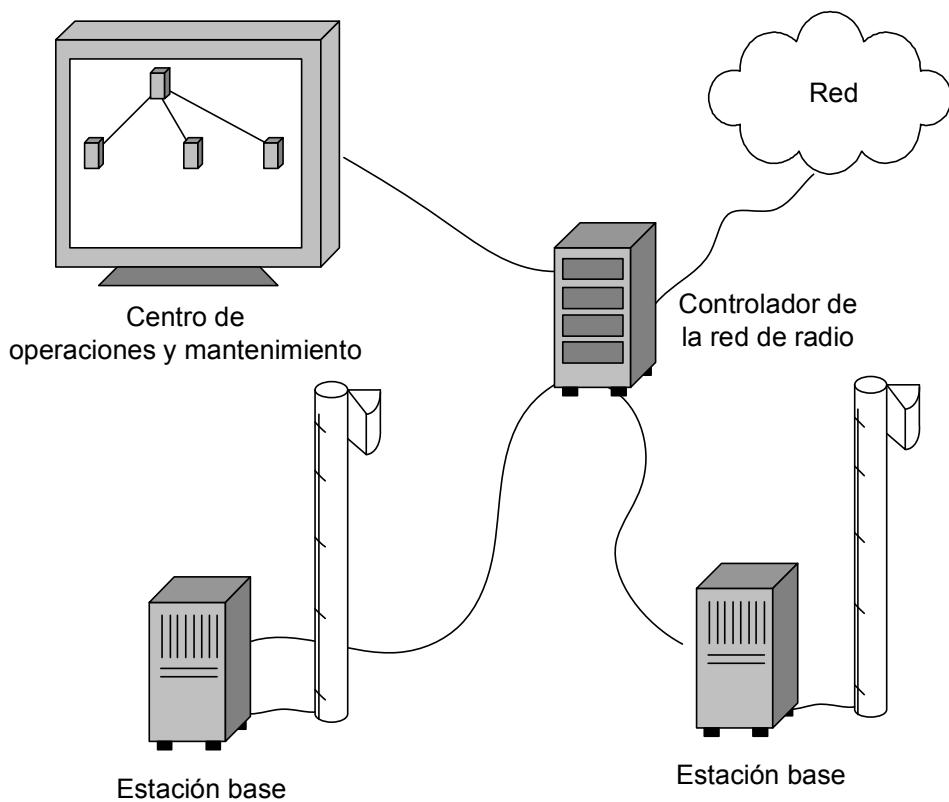


Figura original cortesía de Michael Kircher

Estaciones base

- Se comunican con los teléfonos móviles que hay en su celda a través de una o varias antenas.
- Envían los datos que reciben de un teléfono móvil al RNC, que los reenvía a otra estación base (cuando el destinatario es otro móvil) o a otra red (p.ej. red de telefonía convencional RTC).
- Se reparten por amplias zonas geográficas y han de ser muy fiables para reducir su coste de mantenimiento (suelen tener múltiples CPUs para ser capaces de procesar múltiples llamadas en paralelo y evitar que el fallo de una CPU deje inoperativa la estación base)

RNCs

- Hacen de mediadores entre las estaciones base y otras redes.
- Son ordenadores potentes que suelen funcionar con sistemas operativos convencionales.

OMC

- Controla la configuración del hardware y del software de las estaciones base y de los RNCs.
- Monitoriza el funcionamiento de la red y permite que operadores humanos intervengan cuando sea necesario.
- Es el único elemento de la red que tiene interfaz de usuario.
- Suele estar formado por un cluster de ordenadores (para repartir la carga de trabajo entre varias máquinas y evitar que el fallo de una de ellas inutilice el OMC).

Establecimiento de una llamada con un teléfono fijo:

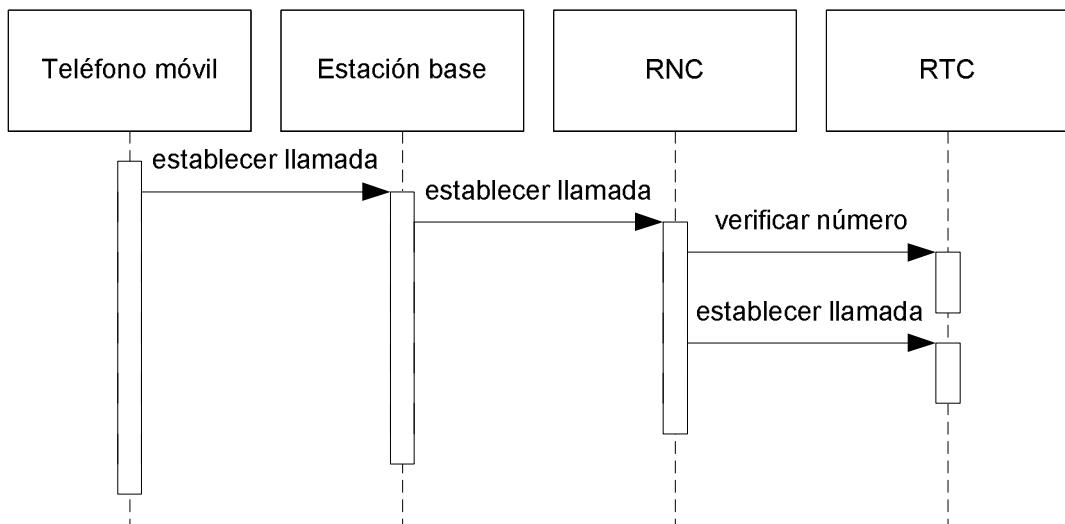


Figura original cortesía de Michael Kircher

Cuando la señal que le llega de un teléfono móvil a la estación base es demasiado débil, el RNC se encarga de asignarle otra estación base más cercana sin que se corte la conexión (proceso conocido como “call handover”):

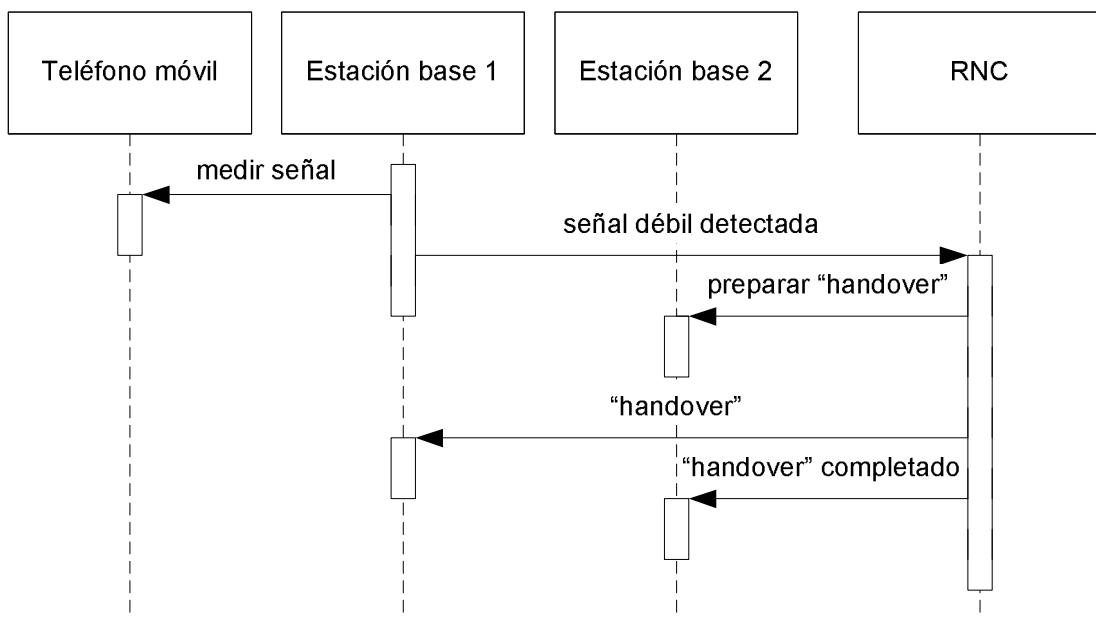
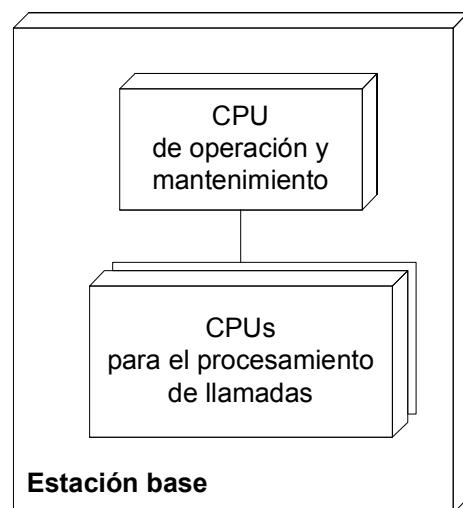


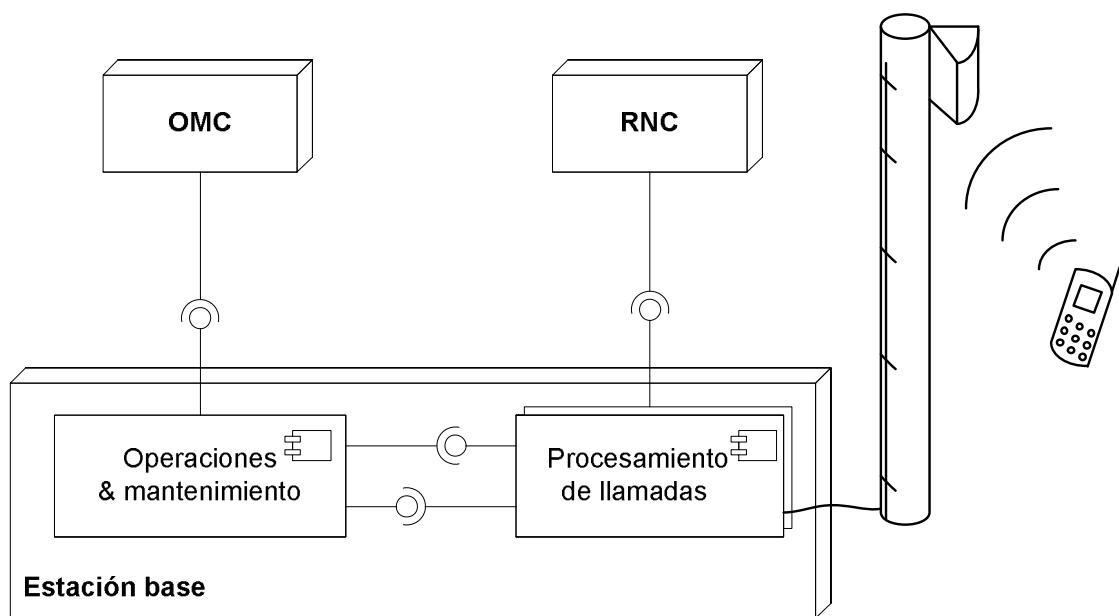
Figura original cortesía de Michael Kircher

La arquitectura de una estación base

El hardware de la estación base



El software de la estación base

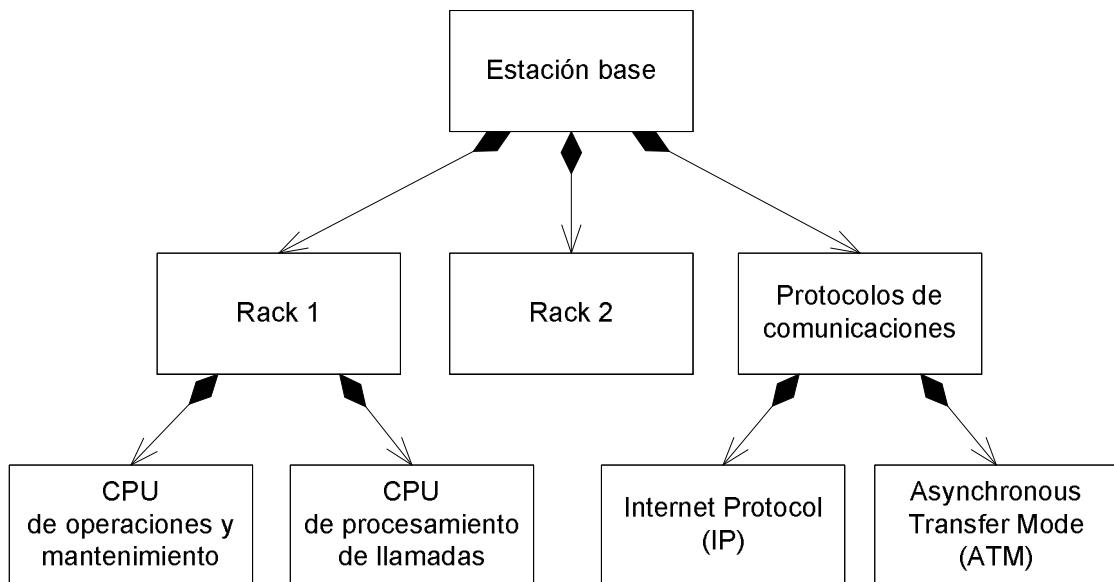


Unidad de procesamiento de llamadas...

- Establecimiento de conexiones (teléfono móvil ↔ RNC).
- Gestión de conexiones entre CPUs de la estación base.
- Monitorización de las señales (p.ej. “handover”)

Unidad de operaciones y mantenimiento...

representa todos los elementos de la estación base como un árbol



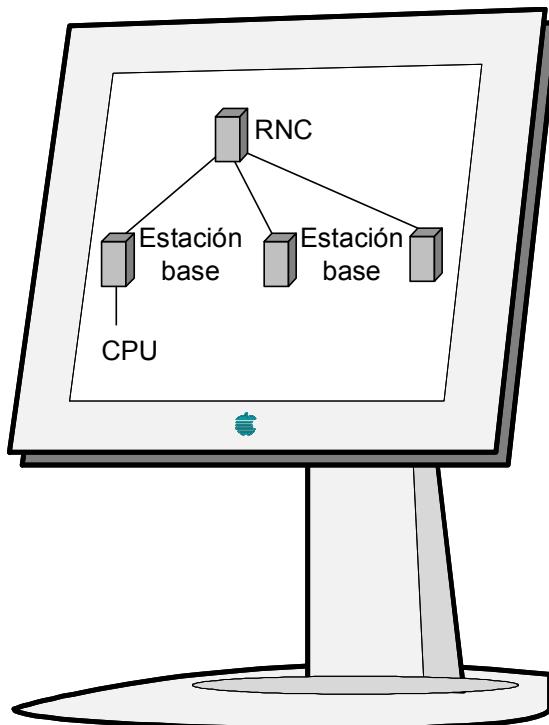
- Monitorización y configuración de los elementos hardware y software de la estación base.
- El árbol de elementos gestionados por la estación base es accesible remotamente desde el OMC para que el OMC pueda controlar la estación base.

La arquitectura del centro de operaciones y mantenimiento

Administración y monitorización de la red de telefonía móvil

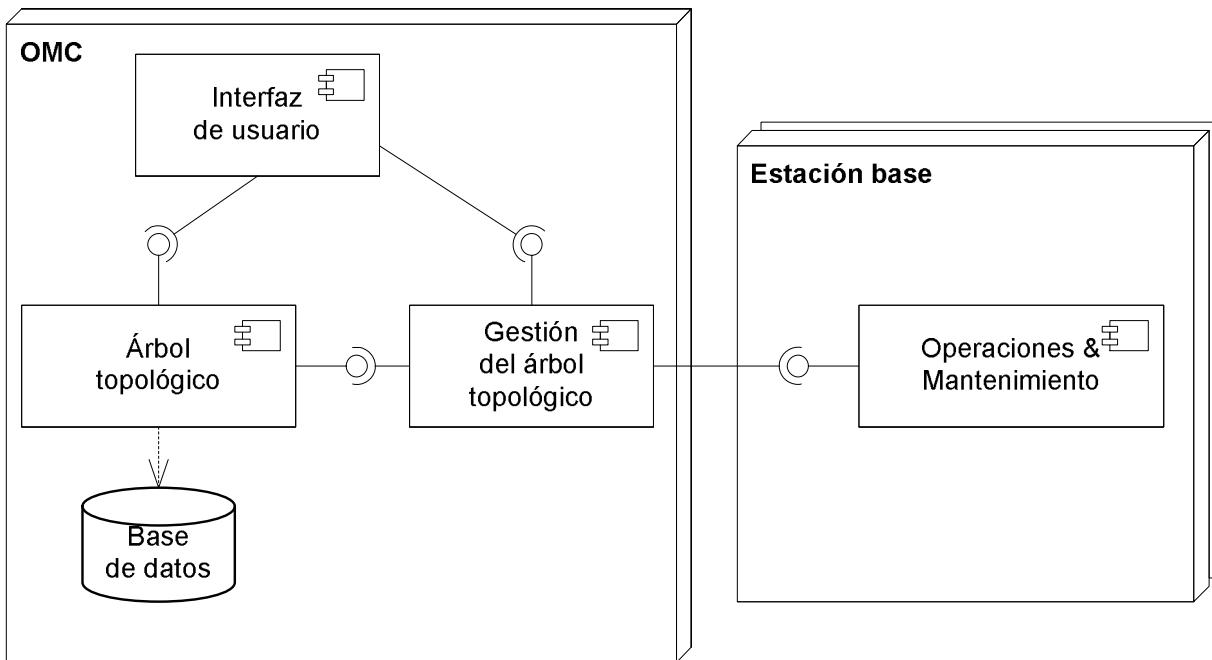
Funciones del OMC

- Comunicación con las estaciones base y los RNCs.
- Descubrimiento de estaciones base.
- Mantenimiento del estado de la red
(árbol topológico completo de toda la red).
- Configuración de los elementos de la red.
- Actualización del software de las estaciones base



Interfaz gráfica de usuario para gestionar los elementos de la red

Arquitectura del software del OMC



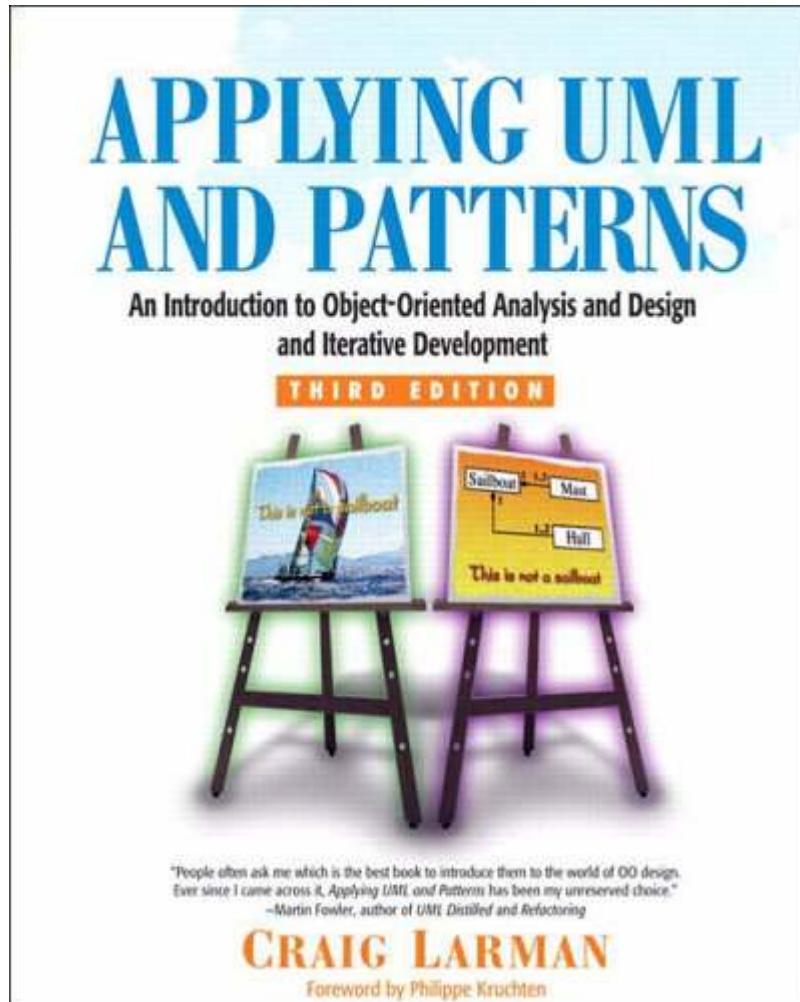
*El patrón de diseño MVC
(Model-View-Controller = Modelo-Vista-Controlador)*

El software del OMC está organizado según el patrón MVC:

- MODELO: El árbol topológico mantiene información de todos los elementos de la red.
- VISTA: La interfaz de usuario permite la interacción del operador con los elementos de la red.
- CONTROLADOR: El componente de gestión del árbol topológico se encarga de manipular el árbol y acceder a las unidades de operaciones y mantenimiento de los elementos de la red (estaciones base y RNCs).

NOTA: Los datos del estado de la red se almacenan localmente en una base de datos por cuestiones de eficiencia (para no tener que contactar con las estaciones base cada vez que queremos algo, lo que introduciría retardos que impedirían el uso cómodo de la interfaz de usuario).

Para aprender más sobre diseño orientado a objetos...



Excepciones

¿Qué es una excepción?

Una situación que, aunque puede pasar,
suele suponer algo negativo para la ejecución de nuestro programa.

Ejemplos

- El usuario escribe una palabra cuando se esperaba un número.
- El programa intenta leer un fichero que no existe.
- El programa no puede establecer una conexión de red.
- Una conexión de red se pierde
- El programa intenta realizar una división por cero.
- Se intenta calcular la raíz cuadrada de un número negativo.
- El programa se sale de los límites de un array.
- El programa accede a los miembros de un objeto inexistente.

Terminología

- Cuando se produce una excepción, la excepción “se lanza” [*throw*].
- Cuando se prevé el posible lanzamiento de una excepción y se toman medidas al respecto en el código de nuestra aplicación, se “captura” la excepción [*catch*].
- El manejo/tratamiento de excepciones consiste en capturar una excepción y tomar las medidas adecuadas al respecto.

Gestión de errores

Qué hacer cuando se pueden producir errores...

```
class EjemploSinControlDeErrores
{
    public static void main(String args[])
    {
        mostrarEntero (args, 0);
    }

    public static void mostrarEntero
                    (String args[], int n)
    {
        System.out.println( "Entero: "
                            + obtenerEntero(args, 0) );
    }

    public static int obtenerEntero
                    (String args[], int n)
    {
        return Integer.parseInt(args[n]);
    }
}
```

Idea inicial: Evitar los errores antes de que se produzcan

- Añadir comprobaciones **antes** de operaciones “peligrosas”
- No hacer nada que pueda fallar (algo imposible)

Por tanto, es necesario detectar cuándo se produce un error para tomar las medidas oportunas en cada momento...

Solución 1: Códigos de error

- Añadir comprobaciones **después** de operaciones “falibles”
- Si se detecta algún error, devolver algún tipo de código de error.

```
class EjemploConControlDeErrores
{
    public static void main(String args[])
    {
        mostrarEntero(args, 0);
    }

    public static int mostrarEntero
                    (String args[], int n)
    {
        int i = obtenerEntero(args, n);
        int error = 0;
        String salida = null;

        if (i%2 == 0)
            i = i/2;
        else
            error = -1; // Error en los argumentos

        if (error == 0) {
            if (Runtime.getRuntime().freeMemory() > (8+10)*2 )
                error = -2; // Memoria insuficiente
        }

        if (error == 0) {
            salida = "Entero: " + i;
            if (salida == null)
                error = -3; // Error al crear la salida
        }

        if (error == 0) {
            System.out.println(salida);
            if (System.out.checkError())
                error = -4; // Error al mostrar la salida
        }
    }

    return error;
}
```

```

public static int obtenerEntero
    (String args[], int n)
{
    int error = 0;

    if (args == null || args.length == 0)
        error = -1; // Vector inexistente

    if (error == 0) {
        if ((n<0) || (n>=args.length))
            error = -3;
    }

    if (error == 0) {
        if (!comprobarEntero(args[n]))
            error = -5;
    }

    // Devuelve 2*entero, -1, -3 o -5

    if (error == 0)
        return 2*Integer.parseInt(args[n]);
    else
        return error;
}

public static boolean comprobarEntero
    (String entero)
{
    ...
}

```

Inconvenientes

- ⌘ Demasiadas comprobaciones
- ⌘ Código excesivamente enrevesado (confuso y propenso a errores)

Solución 2: Excepciones

Ya que los errores son inevitables, las excepciones nos proporcionan una estructura de control que permite implementar los “casos normales” con facilidad y tratar separadamente los “casos excepcionales”

```
class EjemploConExcepciones
{
    public static void main(String args[])
    {
        try {
            mostrarEntero (args, 0);
        } catch (Exception error) {
            // Casos excepcionales...
        }
    }

    // Casos normales...

    public static void mostrarEntero
                    (String args[], int n)
    {
        System.out.println( "Entero: "
                            + obtenerEntero(args,0) );
    }

    public static int obtenerEntero
                    (String args[], int n)
    {
        return Integer.parseInt(args[n]);
    }
}
```

Además, las excepciones nos permitirán mantener información acerca de lo que falló (tipo de error, detalles relevantes y lugar en el que se produjo el error) y enviar esta información al método que queramos que se encargue de tratar el problema, todo esto sin interferir con el funcionamiento normal del programa (p.ej. sentencias `return`).

Uso de excepciones en Java

En Java, cuando se produce un error en un método,
“se lanza” un objeto Throwable.

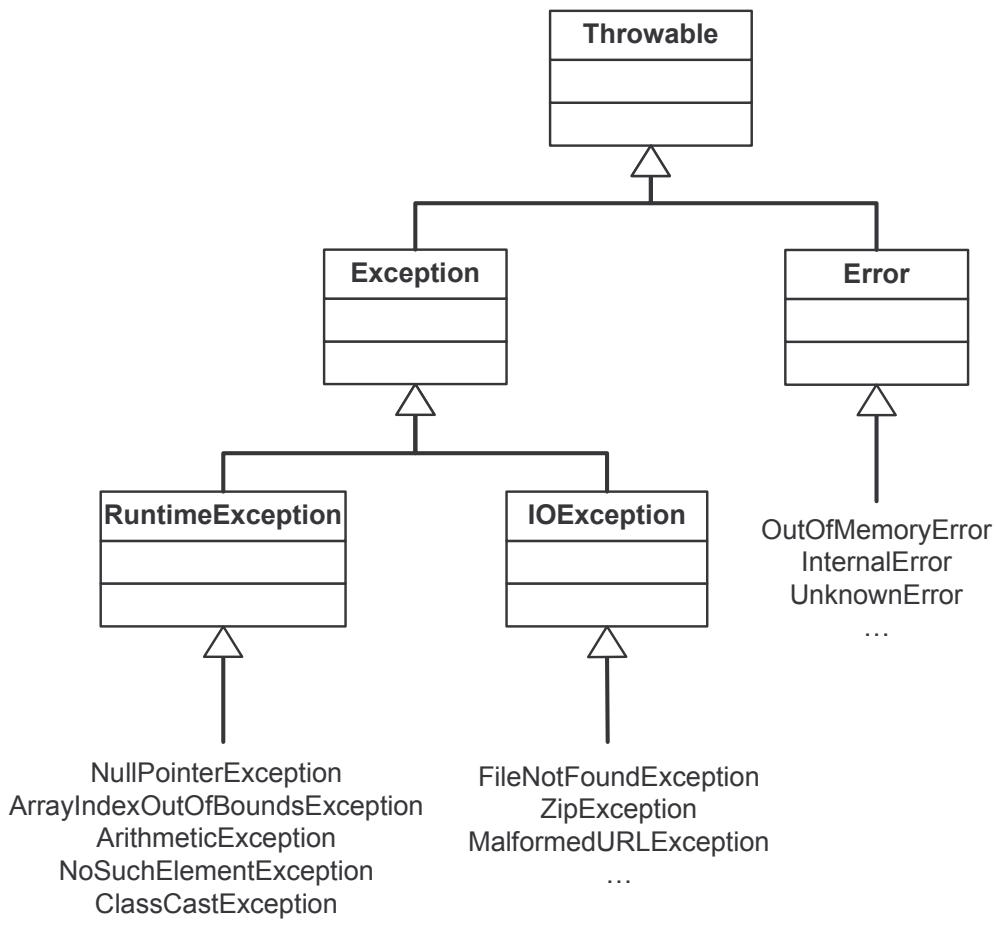
Cualquier método que haya llamado al método puede “capturar la excepción” y tomar las medidas que estime oportunas.

Tras capturar la excepción, el control no vuelve al método en el que se produjo la excepción, sino que la ejecución del programa continúa en el punto donde se haya capturado la excepción.

Consecuencia:

Nunca más tendremos que preocuparnos de “diseñar” códigos de error.

Jerarquía de clases para el manejo de excepciones en Java



Throwable

Clase base que representa todo lo que se puede “lanzar” en Java

- Contiene una instantánea del estado de la pila en el momento en el que se creó el objeto (“stack trace” o “call chain”).
- Almacena un mensaje (variable de instancia de tipo `String`) que podemos utilizar para detallar qué error se produjo.
- Puede tener una causa, también de tipo `Throwable`, que permite representar el error que causó este error.

Error

Subclase de `Throwable` que indica problemas graves que una aplicación no debería intentar solucionar (documentación de Java).

Ejemplos: Memoria agotada, error interno de la JVM...

Exception

`Exception` y sus subclases indican situaciones que una aplicación debería tratar de forma razonable.

Los dos tipos principales de excepciones son:

- `RuntimeException` (errores del programador, como una división por cero o el acceso fuera de los límites de un array)
- `IOException` (errores que no puede evitar el programador, generalmente relacionados con la entrada/salida del programa).

Captura de excepciones: Bloques try...catch

Se utilizan en Java para capturar las excepciones que se hayan podido producir en el bloque de código delimitado por `try` y `catch`.

En cuanto se produce la excepción, la ejecución del bloque `try` termina.

La cláusula `catch` recibe como argumento un objeto `Throwable`.

```
// Bloque 1
try {
    // Bloque 2
} catch (Exception error) {
    // Bloque 3
}
// Bloque 4
```

Sin excepciones: $1 \rightarrow 2 \rightarrow 4$

Con una excepción en el bloque 2: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 4$

Con una excepción en el bloque 1: 1^*

```
// Bloque 1
try {
    // Bloque 2
} catch (ArithmetricException ae) {
    // Bloque 3
} catch (NullPointerException ne) {
    // Bloque 4
}
// Bloque 5
```

Sin excepciones: $1 \rightarrow 2 \rightarrow 5$

Excepción de tipo aritmético: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 5$

Acceso a un objeto nulo (`null`): $1 \rightarrow 2^* \rightarrow 4 \rightarrow 5$

Excepción de otro tipo diferente: $1 \rightarrow 2^*$

```

// Bloque1
try {
    // Bloque 2
} catch (ArithmeticException ae) {
    // Bloque 3
} catch (Exception error) {
    // Bloque 4
}
// Bloque 5

```

Sin excepciones: $1 \rightarrow 2 \rightarrow 5$

Excepción de tipo aritmético: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 5$

Excepción de otro tipo diferente: $1 \rightarrow 2^* \rightarrow 4 \rightarrow 5$

¡Ojo! Las cláusulas check se comprueban en orden

```

// Bloque1
try {
    // Bloque 2
} catch (Exception error) {
    // Bloque 3
} catch (ArithmeticException ae) {
    // Bloque 4
}
// Bloque 5

```

Sin excepciones: $1 \rightarrow 2 \rightarrow 5$

Excepción de tipo aritmético: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 5$

Excepción de otro tipo diferente: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 5$

¡ El bloque 4 nunca se llegará a ejecutar !

La cláusula `finally`

En ocasiones, nos interesa ejecutar un fragmento de código independientemente de si se produce o no una excepción (por ejemplo, cerrar un fichero que estemos manipulando).

```
// Bloque1
try {
    // Bloque 2
} catch (ArithmetricException ae) {
    // Bloque 3
} finally {
    // Bloque 4
}
// Bloque 5
```

Sin excepciones: $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$

Excepción de tipo aritmético: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 4 \rightarrow 5$

Excepción de otro tipo diferente: $1 \rightarrow 2^* \rightarrow 4$

Si el cuerpo del bloque `try` llega a comenzar su ejecución, el bloque `finally` siempre se ejecutará...

■ Detrás del bloque `try` si no se producen excepciones

■ Despues de un bloque `catch` si éste captura la excepción.

■ Justo después de que se produzca la excepción si ninguna cláusula `catch` captura la excepción y antes de que la excepción se propague hacia arriba.

Lanzamiento de excepciones

La sentencia throw

Se utiliza en Java para lanzar objetos de tipo Throwable

```
throw new Exception("Mensaje de error...");
```

Cuando se lanza una excepción:

- Se sale inmediatamente del bloque de código actual
- Si el bloque tiene asociada una cláusula catch adecuada para el tipo de la excepción generada, se ejecuta el cuerpo de la cláusula catch.
- Si no, se sale inmediatamente del bloque (o método) dentro del cual está el bloque en el que se produjo la excepción y se busca una cláusula catch apropiada.
- El proceso continúa hasta llegar al método main de la aplicación. Si ahí tampoco existe una cláusula catch adecuada, la máquina virtual Java finaliza su ejecución con un mensaje de error.

Propagación de excepciones (throws)

Si en el cuerpo de un método se lanza una excepción (de un tipo derivado de la clase `Exception`), en la cabecera del método hay que añadir una cláusula `throws` que incluye una lista de los tipos de excepciones que se pueden producir al invocar el método.

Ejemplo

```
public String leerFichero (String nombreFichero)
                           throws IOException
...

```

Las excepciones de tipo `RuntimeException` (que son muy comunes) no es necesario declararlas en la cláusula `throws`.

Al implementar un método, hay que decidir si las excepciones se propagarán hacia arriba (`throws`) o se capturarán en el propio método (`catch`)

1. Un método que propaga una excepción:

```
public void f() throws IOException
{
    // Fragmento de código que puede
    // lanzar una excepción de tipo IOException
}
```

NOTA: Un método puede lanzar una excepción porque cree explícitamente un objeto `Throwable` y lo lance con `throw`, o bien porque llame a un método que genere la excepción y no la capture.

2. Un método equivalente que no propaga la excepción:

```
public void f()
{
    // Fragmento de código libre de excepciones

    try {
        // Fragmento de código que puede
        // lanzar una excepción de tipo IOException
        // (p.ej. Acceso a un fichero)
    } catch (IOException error) {
        // Tratamiento de la excepción
    } finally {
        // Liberar recursos (siempre se hace)
    }
}
```

Ejemplo clásico

```
...

public void transferir
    (String IDorigen, String IDdestino, int cantidad)
{
    Cuenta origen;
    Cuenta destino;

    // Comenzar transacción
    database.startTransaction();

    try {

        origen = database.find(IDorigen);

        if (origen == null)
            throw new Exception("No existe " + IDorigen);

        origen.setBalance (origen.getBalance() - cantidad);
        database.store(origen);

        destino = database.find(IDdestino);

        if (destino == null)
            throw new Exception("No existe " + IDdestino);

        destino.setBalance(destino.getBalance() + cantidad);
        database.store(destino);

        // Confirmar la transacción
        database.commit();
    } catch (Exception error) {

        // Cancelar la transacción
        database.rollback();
    }
}

...
```

Creación de nuevos tipos de excepciones

Un nuevo tipo de excepción puede crearse fácilmente: basta con definir una subclase de un tipo de excepción ya existente.

```
public DivideByZeroException  
    extends ArithmeticException  
{  
    public DivideByZeroException(String Message)  
    {  
        super(message);  
    }  
}
```

Una excepción de este tipo puede entonces lanzarse como cualquier otra excepción:

```
public double dividir(int num, int den)  
    throws DivideByZeroException  
{  
    if (den==0)  
        throw new DivideByZeroException("Error!");  
  
    return ((double) num/(double)den);`  
}
```

NOTA: Las aplicaciones suelen definir sus propias subclases de la clase `Exception` para representar situaciones excepcionales específicas de cada aplicación.

El sistema de E/S en Java: Ficheros

Organización lógica de los datos

Ficheros y streams

La clase `File`

Streams de entrada: `InputStream`

Streams de salida: `OutputStream`

E/S de caracteres: `Reader & Writer`

Ficheros de texto

Creación de un fichero de texto

Acceso secuencial al contenido de un fichero de texto

Serialización de objetos

Serialización

Deserialización

Uso

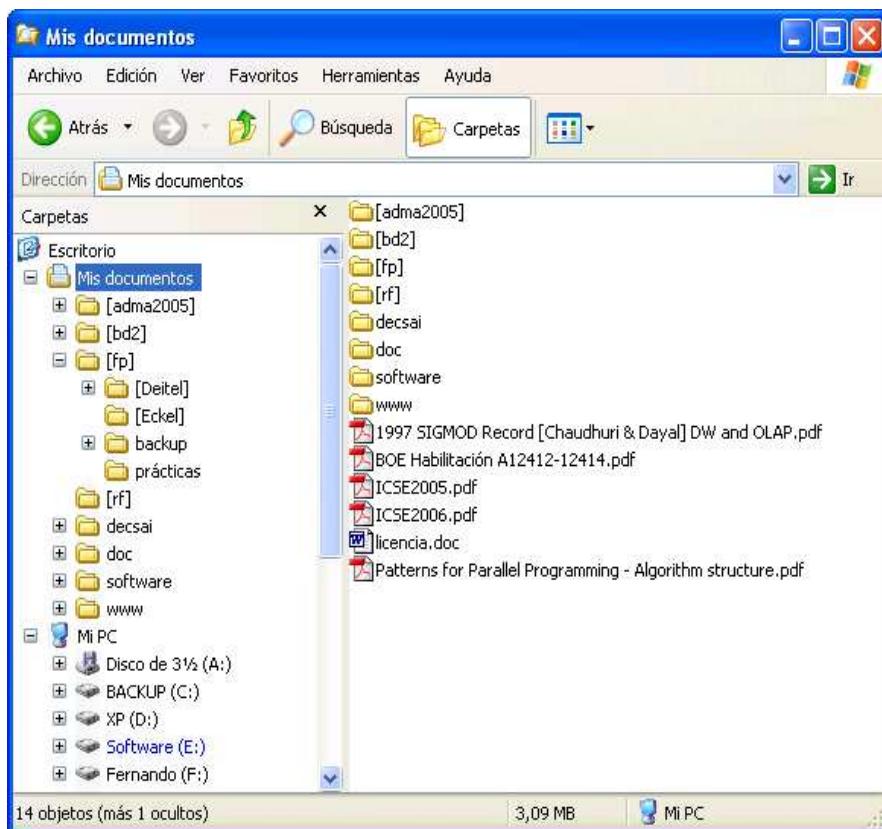
Ficheros de acceso aleatorio

Organización de los datos

Sistemas basados en archivos

| | |
|-----------------------------|--|
| Campo | Unidad mínima (conjunto de bytes que representa un dato, p.ej. cadena, fecha, número) |
| Registro | Conjunto de campos relacionados (p.ej. todos los relacionados con una persona concreta). |
| Fichero | Conjunto de registros relacionados |
| Carpeta / Directorio | Conjunto de ficheros relacionados |

Las carpetas o directorios se organizan jerárquicamente (en carpetas y subcarpetas, directorios y subdirectorios) para formar un árbol:



Ficheros y “streams”

Desde el punto de vista de Java,
cada fichero no es más que una secuencia o flujo de bytes [*stream*].

Los *streams* pueden ser

- de entrada (`InputStream`)
- de salida (`OutputStream`).

Los ficheros pueden almacenar los datos

- como secuencias de caracteres (ficheros de texto)
- como secuencias de bytes (ficheros binarios)

Ejemplo: El número 5 se puede almacenar como el carácter ‘5’ (código ASCII 53, representado en UNICODE mediante la secuencia de bits 0000 0000 0011 0101) o bien como una secuencia de 32 bits en binario (0000...0101)

Un programa en Java accede a un fichero creando un objeto asociado a un stream del tipo adecuado (de entrada o de salida, de caracteres para ficheros de texto o de bytes para ficheros binarios).

Los tipos más comunes de streams están definidos en el paquete `java.io` de la biblioteca de clases estándar de Java.

Ejemplos: En nuestros programas ya hemos utilizado distintos streams para leer datos desde el teclado y mostrar datos por pantalla:

- `System.out` es un `PrintStream`
- `System.in` es un `InputStream`

La clase File

Acceso a información acerca de ficheros y directorios:

```
import java.io.File;
import java.io.IOException;
import java.util.Date;

public class FileDemo
{
    public void mostrarInfoFichero( String path )
        throws IOException
    {
        File fichero = new File( path );
        if ( fichero.exists() ) {
            System.out.println("Nombre: "+fichero.getName());
            System.out.println("- Ruta completa: "
                + fichero.getAbsolutePath());
            System.out.println("- Tamaño: "
                + fichero.length() + " bytes");
            System.out.println("- Última modificación: "
                + new Date(fichero.lastModified()) );
            if (fichero.isFile()) {
                System.out.println("- Fichero normal");
            } else if (fichero.isDirectory()) {
                System.out.println("- Directorio");
                mostrarContenidoDirectorio(fichero);
            }
        } else {
            throw new IOException
                ( "El fichero '" +path+ "' no existe" );
        }
    }

    public void mostrarContenidoDirectorio
        (File directorio)
    {
        String ficheros[] = directorio.list();
        for (int i=0; i<ficheros.length; i++)
            System.out.println("\t"+ficheros[i]);
    }
}
```

```

public static void main (String args[])
    throws IOException
{
    FileDemo demo = new FileDemo();

    if (args.length>0) {
        demo.mostrarInfoFichero(args[0]);
    } else {
        System.out.println ("USO:");
        System.out.println ("java FileDemo <fichero>");
    }
}
}

```

Fichero convencional

```
java FileDemo FileDemo.java
```

| |
|--|
| Nombre: FileDemo.java |
| - Ruta completa: F:\[fp]\ejemplos\FileDemo.java |
| - Tamaño: 1496 bytes |
| - Última modificación: Thu Apr 28 16:31:23 CEST 2005 |
| - Fichero normal |

Directorio

```
java FileDemo f:\[fp]\ejemplos
```

| |
|--|
| Nombre: ejemplos |
| - Ruta completa: f:\[fp]\ejemplos |
| - Tamaño: 0 bytes |
| - Última modificación: Thu Apr 28 16:31:05 CEST 2005 |
| - Directorio |
| FileDemo.class |
| FileDemo.java |

Fichero inexistente

```
java FileDemo xxx.java
```

| |
|--|
| Exception in thread "main" java.io.IOException: |
| El fichero 'xxx.java' no existe |
| at FileDemo.mostrarInfoFichero(FileDemo.java:40) |
| at FileDemo.main(FileDemo.java:59) |

Streams de entrada

Las clases derivadas de `InputStream` representan flujos de datos de entrada que pueden provenir de distintas fuentes:

| Clase | Fuente de datos | Parámetros del constructor |
|--------------------------------------|---------------------------------|---|
| <code>ByteArrayInputStream</code> | Array de bytes | Buffer del que leer los bytes |
| <code>StringBufferInputStream</code> | String | Cadena de caracteres |
| <code>FileInputStream</code> | Fichero | Cadena con el nombre del fichero u objeto de tipo <code>File</code> |
| <code>PipedInputStream</code> | Pipeline* | <code>PipedOutputStream</code> |
| <code>SequenceInputStream</code> | Otros <code>InputStreams</code> | Secuencia de varios <code>InputStreams</code> |

- * Un “pipeline” se utiliza para enviar los datos de salida de un programa directamente a la entrada de otro programa (sin pasar por un fichero en disco)

Los tipos anteriores de `InputStreams` se utilizan como fuentes de datos y se conectan luego a un `FilterInputStream` que nos permite leer con mayor “comodidad” los datos de entrada.

`FilterInputStream` es una clase abstracta de la que derivan las siguientes clases:

| Clase | Función |
|------------------------------------|--|
| <code>DataInputStream</code> | Lectura de datos primitivos en binario (int, char...). Véase <code>DataOutputStream</code> |
| <code>BufferedInputStream</code> | Usa un buffer para evitar el acceso a disco cada vez que se lee un dato |
| <code>LineNumberInputStream</code> | Mantiene el número de línea actual, al que se accede con <code>getLineNumber()</code> |
| <code>PushbackInputStream</code> | Permite “devolver” el último byte leído |

Streams de salida

Las clases derivadas de `OutputStream` permiten decidir a dónde mandar los datos de salida:

| Clase | Destino de los datos | Parámetros del constructor |
|------------------------------------|----------------------|--|
| <code>ByteArrayOutputStream</code> | Array de bytes | Tamaño inicial del buffer |
| <code>FileOutputStream</code> | Fichero | Cadena con el nombre del fichero u objeto de tipo File |
| <code>PipedInputStream</code> | Pipeline* | <code>PipedInputStream</code> |

- * Un “pipeline” se utiliza para enviar los datos de salida de un programa directamente a la entrada de otro programa (sin pasar por un fichero en disco)

Los tipos anteriores de `OutputStreams` se conectan a un `FilterOutputStream` que facilita generar la salida en el formato adecuado.

`FilterOutputStream` es una clase abstracta de la que derivan las siguientes clases:

| Clase | Función |
|-----------------------------------|---|
| <code>DataOutputStream</code> | Salida de datos primitivos en binario. Véase <code>DataInputStream</code> |
| <code>PrintStream</code> | Salida con formato (p.ej. <code>System.out</code>) |
| <code>BufferedOutputStream</code> | Salida a través de un buffer. El método <code>flush()</code> permite vaciarlo. |

Lectores y escritores: Readers & Writers

Ficheros de texto en Java (E/S de caracteres)

Subclases de Reader

| Clase | Fuente | Uso |
|--------------------|-------------|--|
| BufferedReader | Reader | Añade un buffer del que leer los caracteres |
| CharArray Reader | char[] | Lectura de caracteres a partir de un array de caracteres |
| FileReader | Fichero | Lectura de caracteres de un fichero de texto |
| InputStream Reader | InputStream | Convierte un InputStream en un Reader |
| LineNumber Reader | Reader | Mantiene el número de línea |
| Piped Reader | Pipeline* | PipedWriter |
| String Reader | String | Lectura de caracteres de una cadena |

Subclases de Writer

| Clase | Destino | Uso |
|---------------------|--------------|---|
| Buffered Writer | Reader | Añade un buffer en el que escribir caracteres |
| CharArray Writer | char[] | Escritura de caracteres en un array de caracteres |
| FileWriter | Fichero | Escritura de caracteres en un fichero de texto |
| OutputStream Writer | OutputStream | Conecta un Writer con un OutputStream |
| Piped Writer | Pipeline* | PipedReader |
| PrintWriter | OutputStream | Salida con formato en un OutputStream |
| String Writer | String | Escritura de caracteres en una cadena |

Configuraciones típicas

Leer datos desde el teclado con `System.in`

```
import java.io.*;  
  
public class TestSystemIn  
{  
    public static void main (String[] args)  
        throws IOException  
    {  
        InputStreamReader reader;  
        BufferedReader     input;  
  
        // Secuencia de bytes -> Secuencia de caracteres  
  
        reader = new InputStreamReader(System.in);  
  
        // Secuencia de caracteres -> Secuencia de líneas  
  
        input = new BufferedReader(reader);  
  
        // Lectura de una línea de texto  
  
        System.out.println("Escriba algo...");  
        String cadena = input.readLine();  
        System.out.println("Ha escrito: "+cadena);  
    }  
}
```

Configuraciones típicas

Mostrar el contenido de un fichero

Solución 1: Byte a byte

```
import java.io.*;  
  
public class TestReadByte  
{  
    public static void main(String[] args)  
        throws IOException  
    {  
        FileInputStream      file;  
        BufferedInputStream buffered;  
        DataInputStream       in;  
  
        // Apertura del fichero (modo de lectura)  
  
        file = new FileInputStream(args[0]);  
  
        // ... con buffer  
  
        buffered = new BufferedInputStream(file);  
  
        // ... para leer datos en binario  
  
        in = new DataInputStream(buffered);  
  
  
        // Recorrido secuencial del fichero  
  
        while (in.available() != 0)  
            System.out.print((char)in.readByte());  
  
  
        // Cierre del fichero  
  
        in.close();  
    }  
}
```

Configuraciones típicas

Mostrar el contenido de un fichero

Solución 2: Línea a línea

```
import java.io.*;  
  
public class TestReadLine  
{  
    public static void main(String[] args)  
        throws IOException  
    {  
        FileReader      file;  
        BufferedReader in;  
        String         cadena;  
  
        // Apertura del fichero de texto (modo de lectura)  
  
        file = new FileReader(args[0]);  
  
        // ... con buffer para leer línea a línea  
  
        in = new BufferedReader(file);  
  
        // Lectura del fichero línea a línea  
  
        cadena = in.readLine();  
  
        while (cadena!=null) {  
            System.out.println(cadena);  
            cadena = in.readLine();  
        }  
  
        // Cierre del fichero  
  
        in.close();  
    }  
}
```

Caso práctico: Agenda

Vamos a trabajar con los datos de una agenda de contactos:

```
import java.util.Date;

public class Contacto
{
    private String nombre;
    private String teléfono;
    private String email;
    private String dirección;
    private Date nacimiento;
    private int grupo;
    private double deuda;

    // Constantes simbólicas

    public static final int TRABAJO = 1;
    public static final int FAMILIA = 2;
    public static final int AMIGOS = 3;

    // Constructor

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

    // Métodos set & get
    ...

    // Salida estándar

    public String toString ()
    {
        return nombre + "(" + email + ")\n"
            + " Teléfono: " + teléfono + "\n"
            + " Cumpleaños: " + nacimiento + "\n"
            + " Deuda: " + deuda + "€";
    }
}
```

Ficheros de texto

Ejemplos de uso de ficheros de acceso secuencial

```
Fernando
958 24 05 99
fberzal@decsai.ugr.es
CCIA - Despacho 17
3 de diciembre de 1977
1
6.6
Juan Carlos
958 24 05 97
JC.Cubero@decsai.ugr.es
CCIA - Despacho 37
29 de enero de 1962
1
-6.6
```

Fichero de texto clientes.txt
con los datos de dos contactos de nuestra agenda

Creación de un fichero de texto

```
ContactoWriter salida;
...
salida = new ContactoWriter();
salida.abrir();
salida.escribir( contacto1 );
salida.escribir( contacto2 );
salida.cerrar();
```

```

import java.io.*;
import java.util.Date;
import java.text.DateFormat;

public class ContactoWriter
{
    private FileWriter      writer;
    private BufferedWriter  buffer;
    private PrintWriter    output;

    // Apertura del fichero de texto

    public void abrir()
        throws IOException
    {
        try {

            writer = new FileWriter( "clientes.txt" );
            buffer = new BufferedWriter(writer);
            output = new PrintWriter(writer);

        } catch (SecurityException securityException) {

            System.err.println
                ( "No tiene permiso para escribir en el fichero" );
            throw securityException;
        }

        } catch (FileNotFoundException fileException) {

            System.err.println
                ( "Error al crear el fichero" );
            throw fileException;
        }
    }

    // Cierre del fichero de texto

    public void cerrar()
        throws IOException
    {
        if ( output != null )
            output.close();
    }
}

```

```

// Escribir los datos de un contacto en el fichero

public void escribir (Contacto contacto)
{
    DateFormat df;
    String      str;

    df = DateFormat.getDateInstance(DateFormat.LONG);

    if (contacto!=null) {
        output.println(contacto.getNombre());
        output.println(contacto.getTelefono());
        output.println(contacto.getEmail());
        output.println(contacto.getDireccion());

        str = df.format(contacto.getNacimiento());
        output.println(str);

        output.println(contacto.getGrupo());
        output.println(contacto.getDeuda());
    }
}
}

```

- El fichero ha de estar abierto para escribir datos en él.
- Si queremos modificar un fichero de texto, tendremos que reescribirlo por completo (en un fichero auxiliar para no perder los datos si la operación falla).
- Cada llamada al método `escribir(contacto)` le añade al fichero de texto 7 líneas con los datos de contacto de una persona:

| |
|------------------------|
| Fernando |
| 958 24 05 99 |
| fberzal@decsai.ugr.es |
| CCIA - Despacho 17 |
| 3 de diciembre de 1977 |
| 1 |
| 6.6 |

Acceso al contenido de un fichero de texto

El siguiente programa lee secuencialmente los datos almacenados en el fichero de texto creado con la clase ContactoWriter:

```
import java.io.IOException;
import java.util.Date;

public class LeerContactos
{
    public static void main( String args[ ] )
        throws IOException, java.text.ParseException
    {
        Contacto         contacto;
        ContactoReader  entrada;

        // Lectura de datos
        entrada = new ContactoReader();
        entrada.abrir();

        do {
            contacto = entrada.leer();
            System.out.println(contacto);
        } while (contacto!=null);

        entrada.cerrar();
    }
}
```

- La clase auxiliar ContactoReader nos permite ir leyendo los datos de los contactos en el mismo orden en el que están en el fichero (acceso secuencial).
- Los ficheros de texto tienen la ventaja de que podemos leerlos fácilmente con cualquier otro programa y modificarlos a mano.
- Los ficheros de texto tienen el inconveniente de que su manipulación selectiva es difícil (su actualización, por ejemplo, suele requerir volver a escribir el fichero completo).

```

import java.io.*;
import java.util.Date;
import java.text.DateFormat;

public class ContactoReader
{
    private FileReader      reader;
    private BufferedReader input;

    // Abertura del fichero de texto

    public void abrir()
        throws IOException
    {
        try {

            reader = new FileReader( "clientes.txt" );
            input  = new BufferedReader(reader);

        } catch (SecurityException securityException) {

            System.err.println
                ( "No tiene permiso para leer el fichero" );
            throw securityException;

        } catch (FileNotFoundException fileException) {

            System.err.println
                ( "Error al acceder al fichero" );
            throw fileNotFoundException;
        }
    }

    // Cierre del fichero de texto

    public void cerrar()
        throws IOException
    {
        if ( input != null )
            input.close();
    }
}

```

```

// Lectura de datos del fichero de texto

public Contacto leer ()
    throws IOException,
           NumberFormatException,
           java.text.ParseException
{
    Contacto contacto = null;
    Date fecha;
    DateFormat df;
    int grupo;
    double deuda;
    String str;

    df = DateFormat.getDateInstance(DateFormat.LONG);

    str = input.readLine();

    if (str!=null) {

        contacto = new Contacto (str);

        contacto.setTelefono (input.readLine());
        contacto.setEmail (input.readLine());
        contacto.setDireccion (input.readLine());

        fecha = df.parse(input.readLine());
        contacto.setNacimiento (fecha);

        grupo = Integer.parseInt(input.readLine());
        contacto.setGrupo(grupo);

        deuda = Double.parseDouble(input.readLine());
        contacto.setDeuda(deuda);
    }

    return contacto;
}

```

Serialización de objetos

Java facilita el almacenamiento y transmisión del estado de un objeto mediante un mecanismo conocido con el nombre de serialización.

La **serialización** de un objeto consiste en generar una secuencia de bytes lista para su almacenamiento o transmisión. Después, mediante la **deserialización**, el estado original del objeto se puede reconstruir.

Para que un objeto sea serializable, ha de implementar la interfaz `java.io.Serializable` (que lo único que hace es marcar el objeto como serializable, sin que tengamos que implementar ningún método).

```
import java.io.Serializable;
import java.util.Date;

public class Contacto implements Serializable
{
    private String nombre;
    private String telefono;
    private String email;
    private String direccion;
    private Date nacimiento;
    private int grupo;
    private double deuda;

    ...
}
```

- Para que un objeto sea serializable, todas sus variables de instancia han de ser serializables.
- Todos los tipos primitivos en Java son serializables por defecto (igual que los arrays y otros muchos tipos estándar).

Serialización

```
import java.io.*;  
  
public class ContactoOutput  
{  
    private FileOutputStream    file;  
    private ObjectOutputStream output;  
  
    // Abrir el fichero  
  
    public void abrir()  
        throws IOException  
{  
    file = new FileOutputStream( "clientes.ser" );  
    output = new ObjectOutputStream(file);  
}  
  
    // Cerrar el fichero  
  
    public void cerrar()  
        throws IOException  
{  
    if (output!=null)  
        output.close();  
}  
  
    // Escribir en el fichero  
  
    public void escribir (Contacto contacto)  
        throws IOException  
{  
    if (output!=null)  
        output.writeObject(contacto);  
}  
}
```

Deserialización

```
import java.io.*;  
  
public class ContactoInput  
{  
    private FileInputStream    file;  
    private ObjectInputStream input;  
  
    public void abrir()  
        throws IOException  
{  
    file = new FileInputStream( "clientes.ser" );  
    input = new ObjectInputStream (file);  
}  
  
public void cerrar()  
    throws IOException  
{  
    if (input!=null )  
        input.close();  
}  
  
public Contacto leer ()  
    throws IOException, ClassNotFoundException  
{  
    Contacto    contacto = null;  
  
    if (input!=null) {  
  
        try {  
            contacto = (Contacto) input.readObject();  
        } catch (EOFException eof) {  
            // Fin del fichero  
        }  
    }  
  
    return contacto;  
}  
}
```

Uso

Escritura de datos

```
ContactoOutput salida;  
  
salida = new ContactoOutput();  
  
salida.abrir();  
salida.escribir( contacto1 );  
salida.escribir( contacto2 );  
salida.cerrar();
```

Lectura de datos

```
Contacto contacto;  
ContactoInput entrada;  
  
entrada = new ContactoInput();  
  
entrada.abrir();  
  
do {  
    contacto = entrada.leer();  
    System.out.println(contacto);  
} while (contacto!=null);  
  
entrada.cerrar();
```

NOTA: El fichero con los objetos serializados `contactos.ser` almacena los datos en un formato propio de Java, por lo que no se puede leer fácilmente con un simple editor de texto (ni editar).

Ficheros de acceso aleatorio

Los ficheros con los que hemos trabajado hasta ahora (ya sean ficheros de texto o ficheros binarios con objetos serializados) no resultan adecuados para muchas aplicaciones en las que hay que trabajar eficientemente con un subconjunto de los datos almacenados en disco.

En este tipo de aplicaciones, se ha de acceder a un registro concreto dentro de un fichero, por lo que los registros deben ser de tamaño fijo:

| Campo | Contenido | Tamaño |
|---------------------|---------------|------------------|
| Nombre | 32 caracteres | 64 bytes |
| Teléfono | 16 caracteres | 32 bytes |
| E-Mail | 32 caracteres | 64 bytes |
| Dirección | 64 caracteres | 128 bytes |
| Fecha de nacimiento | 32 caracteres | 64 bytes |
| Grupo | 1 int | 4 bytes |
| Deuda | 1 double | 8 bytes |
| TOTAL | | 364 bytes |

- Como Java utiliza caracteres UNICODE, cada carácter de una cadena de caracteres ocupa 16 bits (2 bytes).
- La clase `RandomAccessFile` nos permitirá representar un fichero de acceso aleatorio para el que nosotros definiremos el formato de sus registros.

Registro

```
import java.io.*;
import java.text.DateFormat;

public class Registro extends Contacto
{
    public final static int DIM = 364;

    // Lectura

    public void read (RandomAccessFile file)
                      throws IOException, java.text.ParseException
    {
        DateFormat df;

        setNombre ( readString (file, 32) );
        setTelefono ( readString (file, 16) );
        setEmail ( readString (file, 32) );
        setDireccion ( readString (file, 64) );

        df = DateFormat.getDateInstance(DateFormat.LONG);
        setNacimiento ( df.parse(readString(file,32)) );

        setGrupo(file.readInt());
        setDeuda(file.readDouble());
    }

    private String readString
                  (RandomAccessFile file, int dim)
                  throws IOException
    {
        char campo[ ] = new char[dim];

        for (int i=0; i<dim; i++)
            campo[i] = file.readChar();

        return new String(campo).replace('\0',' ');
    }
}
```

```

// Escritura

public void write (RandomAccessFile file)
    throws IOException
{
    DateFormat df;

    writeString (file, getNombre(), 32);
    writeString (file, getTelefono(), 16);
    writeString (file, getEmail(), 32);
    writeString (file, getDireccion(), 64);

    df = DateFormat.getDateInstance(DateFormat.LONG);
    writeString (file, df.format(getNacimiento()), 32);

    file.writeInt (getGrupo());
    file.writeDouble (getDeuda());
}

private void writeString
    (RandomAccessFile file, String str, int dim)
    throws IOException
{
    StringBuffer buffer = new StringBuffer();

    if (str!=null)
        buffer.append(str);

    buffer.setLength(dim);
    file.writeChars(buffer.toString());
}

```

Fichero de contactos

```
import java.io.*;  
  
public class Contactos  
{  
  
    // Fichero de acceso aleatorio  
  
    private RandomAccessFile file;  
  
    // Apertura del fichero  
  
    public void abrir()  
        throws IOException  
    {  
        file = new RandomAccessFile("clientes.dat", "rw");  
    }  
  
    // Cierre del fichero  
  
    public void cerrar()  
        throws IOException  
    {  
        if (file!=null)  
            file.close();  
    }  
  
    // Escribir un registro  
    // en la posición actual del cursor  
  
    public void escribir (Registro registro)  
        throws IOException  
    {  
        if (file!=null)  
            registro.write(file);  
    }  
}
```

```

// Escribir un registro en una posición cualquiera

public void escribir (Registro registro, int pos)
    throws IOException
{
    if (file!=null) {
        file.seek ( (pos-1)*Registro.DIM );
        escribir(registro);
    }
}

// Leer del fichero el registro
// que se encuentra en la posición actual del cursor

public Registro leer ()
{
    Registro registro = null;

    if (file!=null) {

        try {
            registro = new Registro();
            registro.read(file);
        } catch (Exception error) {
            registro = null;
        }
    }

    return registro;
}

// Leer del fichero un registro cualquiera
// (el parámetro indica la posición del registro)

public Registro leer (int pos)
    throws IOException
{
    if (file!=null) {
        file.seek ( (pos-1)*Registro.DIM );
    }

    return leer();
}
}

```

Ejemplos de uso

```
Registro contacto;
Contactos agenda;

agenda = new Contactos();
```

Escritura secuencial de datos

```
agenda.abrir();
agenda.escribir( contacto1 );
agenda.escribir( contacto2 );
...
agenda.cerrar();
```

Lectura secuencial del fichero

```
agenda.abrir();

do {
    contacto = agenda.leer();
    ...
} while (contacto!=null);

agenda.cerrar();
```

Acceso aleatorio a los datos (lectura y actualización)

```
agenda.abrir();

contacto = agenda.leer(2);

contacto.setNombre( "JC" );

agenda.escribir(contacto,2);

agenda.cerrar();
```

Concurrencia

Procesos y hebras

- Concurrencia
- Programación concurrente
- ¿Por qué usar hebras y procesos?

Ejecución de procesos

Ejecución de hebras

- Hebras vs. Procesos
- Creación y ejecución de hebras
- La prioridad de las hebras
- Finalización de la ejecución de una hebra

Uso de recursos compartidos

- Mecanismos de exclusión mutua
- `synchronized`

Hebras e interfaces de usuario

- `SwingUtilities.invokeLater()`
- `javax.swing.Timer`

Más información...

Procesos y hebras

Hoy en día, cualquier usuario espera poder hacer varias cosas a la vez y no verse forzado a ejecutar los programas secuencialmente.

Los sistemas operativos multitarea, como Windows o UNIX, se encargan de que varios programas se puedan ejecutar a la vez (*concurrentemente*) incluso cuando sólo se dispone de una única CPU.

Concurrencia

Dos **tareas** se dice que son **concurrentes** si transcurren durante el mismo intervalo de tiempo.

Se entiende por **programación concurrente** el conjunto de técnicas y notaciones que sirven para expresar el paralelismo potencial en los programas, así como resolver problemas de comunicación y sincronización.

Cuando se trabaja en entornos distribuidos o con máquinas con múltiples procesadores, se suele hablar de **programación distribuida o paralela**, respectivamente.

NOTA: En un PC, el sistema operativo pasa el control de la CPU de una tarea a otra cada pocos milisegundos, algo conocido como **cambio de contexto**.

Al realizar los cambios de contexto en intervalos de tiempo muy cortos, el usuario tiene la percepción de que las distintas tareas se ejecutan en paralelo (algo que, obviamente, sólo sucede si disponemos de un multiprocesador o de un multicomputador).

Cuando decidimos descomponemos un programa en varias tareas potencialmente paralelas, estas tareas las podemos implementar en el ordenador como procesos o como hebras:

■ Un **PROCESO** es un programa en ejecución con un estado asociado.

- Las distintas aplicaciones que se pueden ejecutar en un sistema operativo multitarea son procesos independientes.
- Cada proceso ocupa un espacio de memoria independiente (para no interferir con la ejecución de otros procesos).
- Una aplicación puede implementarse como un conjunto de procesos que colaboren entre sí para lograr sus objetivos, para lo que se pueden emplear distintos *mecanismos de comunicación entre procesos*.

■ Los sistemas operativos actuales permiten un nivel adicional de paralelismo dentro de un proceso: En un proceso pueden existir varias **HEBRAS** de control independientes [*threads*].

- Cada hebra es una vía simultánea de ejecución dentro del espacio de memoria del proceso.
- La comunicación entre las distintas hebras se puede realizar a través del espacio de memoria que comparten, aunque habrá que utilizar *mecanismos de sincronización* para controlar el acceso a este recurso compartido por todas las hebras de un proceso.

Se denomina **APLICACIÓN CONCURRENTE** a una aplicación que se descompone en un conjunto de procesos y/o hebras. Del mismo modo, una **APLICACIÓN MULTIHEBRA** está constituida por distintas hebras que comparten el espacio de memoria de un proceso.

Programación concurrente

Independientemente de si utilizamos procesos o hebras, el desarrollo de aplicaciones concurrentes involucra el uso de técnicas específicas y la superación de dificultades que no se presentan en la implementación de programas secuenciales.

A la hora de crear aplicaciones concurrentes, distribuidas o paralelas, deberemos tener en mente ciertas consideraciones:

- El **diseño** de aplicaciones concurrentes es más complejo que el de aplicaciones secuenciales, ya que hemos de descomponer el programa en un conjunto de tareas con el fin de aprovechar el paralelismo que pueda existir. Si no existe ese paralelismo potencial, no tiene sentido que intentemos descomponer nuestra aplicación en tareas independientes.
- La **implementación** de aplicaciones concurrentes es también más compleja que la de aplicaciones secuenciales convencionales porque hemos de garantizar la coordinación de las distintas hebras o procesos.
- La **depuración** de las aplicaciones concurrentes es extremadamente difícil, dado que la ejecución de los distintos procesos/hebras se entrelaza conforme el sistema operativo les va asignando la CPU (algo que no podemos prever por completo).
- En tiempo de ejecución, además, cada hebra o proceso supone una carga adicional para el sistema, por lo hay que tener en cuenta la **eficiencia** de la implementación resultante. Deberemos ser cuidadosos para asegurar que se aprovecha el paralelismo para mejorar el rendimiento de la aplicación. Este rendimiento puede medirse en función del tiempo de respuesta del sistema o de la cantidad de trabajo que realiza por unidad de tiempo [*throughput*].

¿Por qué usar hebras y procesos?

El no determinismo introducido por el entrelazado de las operaciones de las distintas hebras o procesos de una aplicación concurrente provoca la aparición de errores difíciles de detectar y más aún de corregir: la vida del programador resultaría mucho más sencilla si no hiciese falta la concurrencia.

Sin embargo, existen razones por las cuales es aconsejable utilizar procesos y hebras:

1. De cara al usuario

Hebras y procesos permiten la creación de interfaces que respondan mejor a las órdenes del usuario.

Cuando una aplicación tiene que realizar alguna tarea larga, su interfaz debería seguir respondiendo...

- La ventana de la aplicación debería refrescarse y no quedarse en blanco (como pasa demasiado a menudo).
- Los botones existentes para cancelar una operación deberían cancelar la operación de un modo inmediato (y no al cabo de un rato, cuando la operación ya ha terminado de todos modos).

Si nuestra aplicación ha de atender las peticiones de distintos usuarios (como sucede, por ejemplo, en un servidor web), el uso de hebras o procesos permite que varios usuarios accedan simultáneamente a la aplicación sin tener que esperar turno.

2. Aprovechamiento de los recursos del sistema

Cualquier operación que pueda bloquear nuestra aplicación durante un período de tiempo apreciable es recomendable que se realice de forma independiente.

- La CPU es el dispositivo más rápido del ordenador: Desde su punto de vista, todos los demás dispositivos del ordenador son lentos, desde una impresora hasta un disco duro con Ultra-DMA.
- En un entorno distribuido, las operaciones de E/S son aún más lentas, ya que el tiempo necesario para acceder a un recurso disponible en otra máquina a través de una red depende, entre otros factores, de la carga de la máquina a la que accedemos y del ancho de banda del que dispongamos.

Paralelismo real

Su aprovechamiento requiere el uso de hebras y procesos...

- Un sistema multiprocesador dispone de varias CPUs.
- Un cluster de ordenadores está compuesto por un conjunto de ordenadores conectados entre sí, a los que se accede como si se tratase de un único ordenador.
- Algunas versiones del Pentium 4 de Intel incorporan *Simultaneous MultiThreading* (SMT), que Intel denomina comercialmente *Hyper-Threading*, con lo que un único microprocesador puede funcionar como si tuviésemos un multiprocesador.

3. Modularización: Paralelismo implícito

A veces, los motivos que nos llevan a utilizar hebras o procesos no tienen una justificación física, sino que un programa puede diseñarse con más comodidad si lo descomponemos en un conjunto de tareas independientes.

Un ejemplo mundano...

El problema de perder peso puede verse como una combinación de dos actividades: hacer ejercicio y mantener una dieta equilibrada. Ambas deben realizarse durante el mismo período de tiempo, aunque no necesariamente a la vez.

En situaciones de este tipo, la concurrencia de varias actividades es la solución adecuada para un problema: la solución natural al problema implica concurrencia, mientras que una solución secuencial sería extremadamente difícil.

Ejemplos: Simulaciones, videojuegos...

IMPORTANTE

El objetivo principal del uso de paralelismo es mejorar el rendimiento del sistema.

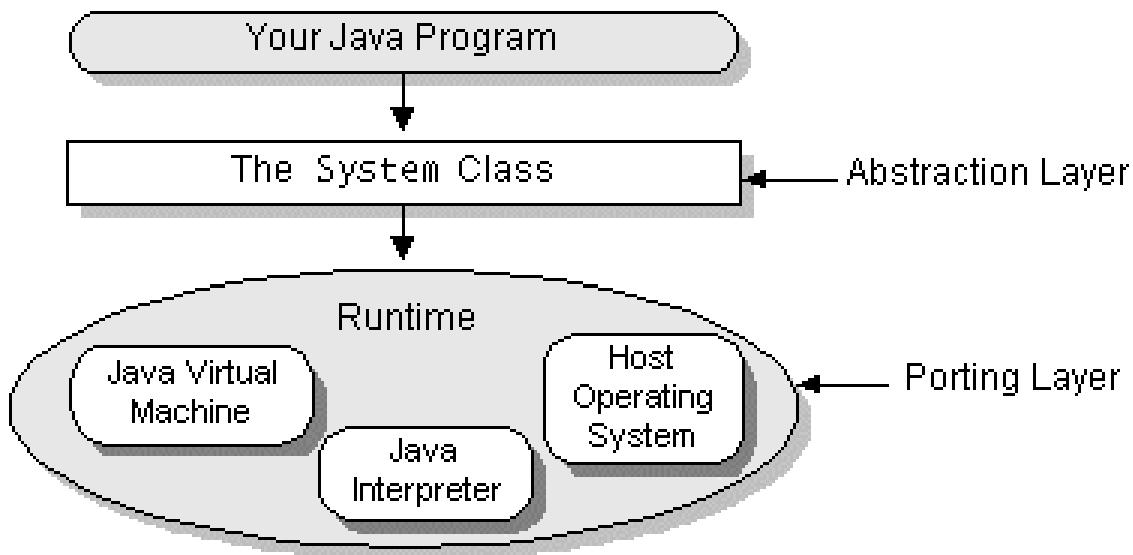
El uso de paralelismo (hebras o procesos) permite mejorar el tiempo de respuesta de una aplicación o la carga de trabajo que puede soportar el sistema.

No obstante, un número excesivo de hebras o procesos puede llegar a degradar el rendimiento del sistema (debido al tiempo de CPU requerido por los cambios de contexto): hay que analizar hasta qué punto compensa utilizar hebras y procesos.

Ejecución de procesos

La máquina virtual Java nos aísla de los detalles particulares de la máquina en la que se ejecutan nuestras aplicaciones. No obstante,

- La clase `System` nos permite acceder a recursos del sistema de forma portable (aunque de una forma muy rudimentaria):
 - Usar los dispositivos de E/S estándar
`System.in & System.out`
 - Acceder a algunas variables de entorno
`System.getProperties()`
- La clase `Runtime` nos permite ejecutar comandos del sistema operativo y “controlar” la máquina virtual Java (ver cuánta memoria queda libre o forzar la ejecución del recolector de basura, por ejemplo).



¡OJO! Para mantener la portabilidad de nuestras aplicaciones, es aconsejable evitar, en la medida que sea posible, el uso de características específicas de un sistema operativo concreto.

La ejecución de procesos es responsabilidad del sistema operativo, del cual nos aísla la máquina virtual Java, si bien podemos usar la clase Runtime:

1. Sólo existe una instancia de la clase Runtime, que se obtiene con:

```
Runtime rt = Runtime.getRuntime();
```

2. El objeto obtenido con la llamada a Runtime.getRuntime() nos permite ejecutar procesos con

```
rt.exec("miprograma");
```

Ejemplo (NO PORTABLE)

```
public class Procesos
{
    public static void main(String[] args)
        throws java.io.IOException
    {
        Runtime rt = Runtime.getRuntime();
        rt.exec("cmd /c start iexplore http://elvex.ugr.es/");
    }
}
```

Podemos evitar el uso del shell (cmd.exe) si especificamos la ruta completa del programa que deseamos ejecutar, p.ej.

```
c:/Archivos de Programa/Internet Explorer/iexplore.exe
```

Si estuviésemos usando UNIX, deberíamos haber escrito algo como:

```
rt.exec("/usr/mozilla/mozilla-bin http://elvex.ugr.es/");
```

En Windows, también podemos omitir el nombre del programa y poner directamente la ruta del recurso al que queremos acceder (el shell del S.O. se encarga de buscar cuál es el programa adecuado):

```
rt.exec("cmd /c start http://elvex.ugr.es/");
rt.exec("cmd /c start mailto:fberzal@decsai.ugr.es");
rt.exec("cmd /c apuntes.doc");
```

Ejecución de hebras

En realidad, todas las aplicaciones escritas en Java son aplicaciones multihebra (recuerde el recolector de basura).

Hebras vs. Procesos

- Los cambios de contexto son más costosos en el caso de los procesos al tener que cambiar el espacio de direcciones en el que se trabaja.
- La comunicación entre procesos que se ejecutan en distintos espacios de memoria es más costosa que cuando se realiza a través de la memoria que comparten las distintas hebras de un proceso.
- Sin embargo, un fallo en una hebra puede ocasionar la caída completa de la aplicación mientras que, si se crean procesos independientes, los fallos se pueden contener en el interior de un proceso, con lo que la tolerancia a fallos de la aplicación será mayor.

Creación y ejecución de hebras en Java

La forma más sencilla de crear una hebra en Java es diseñar una subclase de `java.lang.Thread`.

En la subclase redefiniremos el método `run()`, que viene a ser algo así como el `main()` de una hebra.

Para comenzar la ejecución paralela de la nueva hebra, usaremos su método `start()`.

```

public class HebraContador extends Thread
{
    private int contador = 10;

    private static int hebras = 0;

    // Constructor

    public HebraContador()
    {
        super("Hebra " + ++hebras);
    }

    // Salida estándar

    public String toString()
    {
        return getName() + ":" + contador;
    }

    // Ejecución de la hebra

    public void run()
    {
        while (contador>0) {
            System.out.println(this);
            contador--;
        }
    }

    // Programa principal

    public static void main(String[] args)
    {
        int i;
        Thread hebra;

        for (i = 0; i < 5; i++)
            hebra = new HebraContador();
            hebra.start();
    }
}

```

El orden en que se realizan las operaciones de las distintas hebras puede cambiar de una ejecución a otra
(en función de cómo se asigne la CPU a la distintas hebras de nuestro programa).

NOTA: En el ejemplo anterior, lo más probable es que las cinco hebras se ejecuten secuencialmente salvo que incrementemos el valor inicial del contador.

Si queremos ceder explícitamente la CPU para que se le asigne a otra hebra distinta a la actual, podemos utilizar `yield()`:

```
public void run()
{
    while (contador>0) {
        System.out.println(this);
        contador--;
        yield();
    }
}
```

El método `yield()` no se suele utilizar en la práctica, aunque sí suele ser útil detener la ejecución de una hebra durante un período de tiempo determinado (expresado en milisegundos) con una llamada al método `sleep()`:

```
public void run()
{
    while (contador>0) {
        System.out.println(this);
        contador--;

        try {
            sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

Como Java sólo soporta herencia simple y puede que el objeto que representa nuestra hebra sea mejor implementarlo como subclase de otra clase de nuestro sistema, Java nos permite crear una hebra a partir de cualquier objeto que implemente la interfaz Runnable, que define únicamente un método run():

```
public class Contador implements Runnable
{
    private String id;
    private int     contador = 10;
    private static int hebras = 0;

    public Contador()
    {
        hebras++;
        id = "Hebra " + hebras;
    }

    public void run()
    {
        while (contador>0) {
            System.out.println(this);
            contador--;
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String[] args)
    {
        int     i;
        Thread hebra;

        for (i = 0; i < 5; i++) {
            hebra = new Thread ( new Contador() );
            hebra.start();
        }
    }
}
```

La prioridad de las hebras

La prioridad de una hebra le indica al planificador de CPU la importancia de una hebra, de tal modo que le asignará más tiempo de CPU a las hebras de mayor prioridad.

La prioridad de una hebra se cambia con una llamada al método `setPriority()`, usualmente en el constructor de la hebra:

```
public class HebraContador extends Thread  
...  
    public HebraContador (int prioridad)  
    {  
        super("Hebra " + ++hebras);  
        setPriority (prioridad);  
    }
```

Por defecto, la prioridad de las hebras es
Thread.NORM_PRIORITY

En el siguiente fragmento de código, la última hebra que lanzamos se ejecutará “antes” que las demás:

```
int      i;  
Thread  hebra;  
  
for (i = 0; i < 5; i++) {  
    hebra = new HebraContador (Thread.MIN_PRIORITY);  
    hebra.start();  
}  
  
hebra = new HebraContador (Thread.MAX_PRIORITY);  
hebra.start();
```

Cuando en nuestra aplicación tenemos varias hebras ejecutándose, les daremos mayor prioridad aquellas hebras cuyo tiempo de respuesta deba ser menor.

Finalización de la ejecución de una hebra

Se puede utilizar el método `join()` de la clase `Thread` para esperar la finalización de la ejecución de una hebra:

```
private static final int N = 5;

public static void main(String[] args)
{
    int i;
    Thread hebra[] = new Thread[N];

    for (i=0; i<N; i++) {
        hebra[i] = new HebraContador(Thread.NORM_PRIORITY);
        hebra[i].start();
    }

    for (i=0; i<5; i++) {

        try {
            hebra[i].join();
        } catch (InterruptedException e) {
        }

        System.out.println (hebra[i]+" ha terminado");
    }

    System.out.println
        ("El programa principal ha terminado");
}
```

Uso de recursos compartidos

Cada proceso o hebra se ejecuta de forma independiente. Sin embargo, cuando varias hebras (o procesos) han de acceder a un mismo recurso, se ha de coordinar el acceso a ese recurso.

Ejemplos

Una impresora compartida en red debe imprimir los documentos enviados por distintos usuarios uno detrás de otro, sin mezclar partes de uno con partes de otro.

Cuando varias hebras (o procesos) acceden a los mismos datos (ya sea en memoria o en un fichero), hemos de tener mucho cuidado a la hora de actualizarlos.

Proceso 1

Ingreso en una cuenta

P1.A Obtener saldo actual

... ++

P1.B Guardar saldo modificado

Proceso 2

Transferencia a otra cuenta

P2.A Obtener saldo actual

... --

P2.B Guardar saldo modificado

Si el saldo inicial de la cuenta es de 1000€, realizamos un ingreso de 500€ y una transferencia de 750€, esperamos que el saldo final de la cuenta sea de 750€.

Ahora bien, en función de cómo se ordenen las operaciones de las dos tareas independientes (el ingreso y la transferencia), el resultado final puede ser muy distinto:

- ↳ P1.A (lee 1000€)... P2.A (lee 1000€)... P1.B (escribe 1500€)... P2.B (escribe 250€), por lo que nos quedan **250 €**.
- ↳ P2.A (lee 1000€)... P1.A (lee 1000€)... P2.B (almacena 250€)... P1.B (almacena 1500€), por lo que tenemos **1500 €** al final ;-)

El problema proviene de que no sabemos en qué momento se le asignará la CPU a cada hebra/proceso.

Si lo único que nos interesa es leer datos desde distintas tareas independientes, podemos hacerlo en paralelo sin problemas.

Sin embargo, en el momento en el que queramos realizar alguna modificación, hemos de evitar que otras tareas accedan al recurso compartido mientras nosotros estamos usándolo (para que no lean valores incorrectos ni interfieran con lo que estemos haciendo).

Mecanismos de exclusión mutua

Una solución para este problema consiste en que, para acceder a un recurso compartido, haya antes que “adquirirlo”.

La hebra que adquiere el recurso bloquea el acceso a él.

Las demás hebras, para acceder al recurso, intentarán adquirirlo y se quedarán bloqueadas hasta que lo “libere” la hebra que lo posee.

De esta forma, el acceso al recurso compartido se realiza secuencialmente (por lo que deberemos tener cuidado para no eliminar completamente el paralelismo de nuestra aplicación).

IMPORTANTE

Un programa debe avanzar hasta proporcionar una respuesta.

En el caso de los programas secuenciales, la existencia de un bucle infinito es la única causa posible de que el programa no proporcione respuesta alguna.

Cuando trabajamos con aplicaciones multihebra, pueden aparecer bloqueos [*deadlocks*] que se ocasionan cuando cada hebra se queda esperando a que otra de las hebras bloqueadas libere un recurso compartido al que quiere acceder.

synchronized

Java permite definir secciones críticas (fragmentos de código que sólo puede estar ejecutando una hebra) mediante la palabra reservada **synchronized**:

```
synchronized (objeto) {  
  
    // A este bloque de código  
    // sólo puede acceder una hebra  
    // en cada momento  
  
}
```

Si tenemos un objeto cuyo estado queremos actualizar desde distintas hebras de forma coordinada, podemos etiquetar con **synchronized** todos aquellos métodos cuya ejecución concurrente podría causar algún tipo de error:

```
public class Cuenta  
{  
    private int saldo;  
  
    public synchronized void ingreso (int cantidad)  
    {  
        saldo += cantidad;  
    }  
  
    public synchronized void retirada (int cantidad)  
    {  
        saldo -= cantidad;  
    }  
}
```

Cuando se está ejecutando un método **synchronized** asociado a un objeto, el objeto se bloquea y no se puede ejecutar ningún otro método **synchronized** del objeto hasta que termine la ejecución del método que bloqueó el acceso al objeto.

Ejemplo: Realización de reservas

```
if (asiento.disponible())
    asiento.reservar();
```

Si justo después de comprobar que el asiento está disponible y antes de hacer la reserva, se asigna la CPU a otra hebra, ésta podría comprobar que el asiento está libre y reservarlo.

Cuando el control de la CPU vuelva a la primera hebra (que ya había comprobado la disponibilidad de asientos), ésta también hará también la reserva (con lo que tendremos *overbooking*).

Para eliminar el problema, podemos:

1. Incluir el bloque de código en una sección crítica:

```
synchronized (asiento) {

    if (asiento.disponible())
        asiento.reservar();
}
```

2. Marcar con synchronized los métodos `disponible()` y `reservar()` de la clase `Asiento`, además de cualquier otro método que pueda modificar el estado del asiento y de asegurarnos de que las operaciones se completan correctamente:

```
public class Asiento
{
    ...

    public synchronized boolean disponible ()
    {...}

    public synchronized void reservar (...)

        throws AsientoYaReservadoException
    {...}
}
```

Sobre el uso de mecanismos de exclusión mutua

No hay que abusar del uso de mecanismos secciones críticas. Cada cerrojo que se usa impide potencialmente el progreso de la ejecución de otras hebras de la aplicación. Las secciones críticas eliminan el paralelismo y, con él, muchas de las ventajas que nos llevaron a utilizar hebras en primer lugar. En un caso extremo, la aplicación puede carecer por completo de paralelismo y convertirse en una aplicación secuencial pese a que utilicemos hebras en nuestra implementación.

La ausencia de paralelismo no es, ni mucho menos, el peor problema que se nos puede presentar durante el desarrollo de una aplicación multihebra. El uso de mecanismos de sincronización como cerros o monitores puede conducir a otras situaciones poco deseables que también hemos de tener en cuenta:

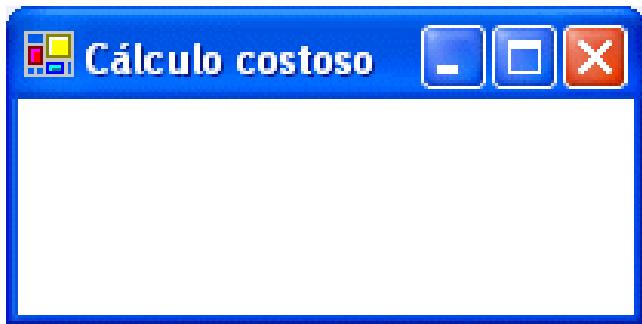
- ✚ **Interbloqueos [deadlocks]:** Una hebra se bloquea cuando intenta bloquear el acceso a un objeto que ya está bloqueado. Si dicho objeto está bloqueado por una hebra que, a su vez, está bloqueada esperando que se libere un objeto bloqueado por la primera hebra, ninguna de las dos hebras avanzará. Ambas se quedarán esperando indefinidamente. El interbloqueo es consecuencia de que el orden de adquisición de los cerros no sea siempre el mismo y la forma más evidente de evitarlo es asegurar que los cerros se adquieran siempre en el mismo orden.
- ✚ **Inanición [starvation],** cuando una hebra nunca avanza porque siempre hay otras a las que se les da preferencia. Por ejemplo, si le damos siempre prioridad a un “lector” frente a un “escritor”, el escritor nunca obtendrá el cerrojo para modificar el recurso compartido mientras haya lectores. Los escritores “se morirán de inanición”.
- ✚ **Inversión de prioridad:** El planificador de CPU decide en cada momento qué hebra, de entre todas las no bloqueadas, ha de disponer de tiempo de CPU. Usualmente, esta decisión viene influida por la prioridad de las hebras. Sin embargo, al usar cerros se puede dar el caso de que una hebra de prioridad alta no avance nunca, justo lo contrario de lo que su prioridad nos podría hacer pensar. Imaginemos que tenemos tres hebras H1, H2 y H3, de mayor a menor prioridad. H3 está ejecutándose y bloquea el acceso al objeto O. H2 adquiere la CPU al tener más prioridad que H3 y comienza un cálculo muy largo. Ahora, H1 le quita la CPU a H2 y, al intentar adquirir el cerrojo de O, queda bloqueada. Entonces, H2 pasa a disponer de la CPU para proseguir su largo cálculo. Mientras, H1 queda bloqueada porque H3 no llega a liberar el cerrojo de O. Mientras que H3 no disponga de algo de tiempo de CPU, H1 no avanzará y H2, pese a tener menor prioridad que H1, sí que lo hará. El planificador de CPU puede solucionar el problema si todas las hebras disponibles avanzan en su ejecución, aunque sea más lentamente cuando tienen menor prioridad. De ahí la importancia de darle una prioridad alta a las hebras cuyo tiempo de respuesta haya de ser bajo, y una prioridad baja a las hebras que realicen tareas muy largas.

Todos estos problemas ponen de manifiesto la dificultad que supone trabajar con aplicaciones multihebra. Hay que asegurarse de asignar los recursos cuidadosamente para minimizar las restricciones de acceso en exclusiva a recursos compartidos. En muchas ocasiones, por ejemplo, se puede simplificar notablemente la implementación de las aplicaciones multihebra si hacemos que cada una de las hebras trabaje de forma independiente, incluso con copias distintas de los mismos datos si hace falta. Sin datos comunes, los mecanismos de sincronización se hacen innecesarios.

Hebras e interfaces de usuario

Cuando ejecutamos una aplicación con una interfaz gráfica de usuario, automáticamente se crea una hebra que se encarga de procesar todos los eventos asociados a la interfaz de usuario.

Cuando nuestra aplicación ha de realizar alguna operación costosa en tiempo de ejecución, esta operación hemos de realizarla en una hebra aparte. Si no lo hacemos así, podemos encontrarnos con algo como lo siguiente:



Si mantenemos ocupada la hebra que se encarga de gestionar los eventos de la interfaz de usuario, ésta dejará de responder al usuario (la ventana deja de refrescarse y no podemos controlar la ejecución de la aplicación).

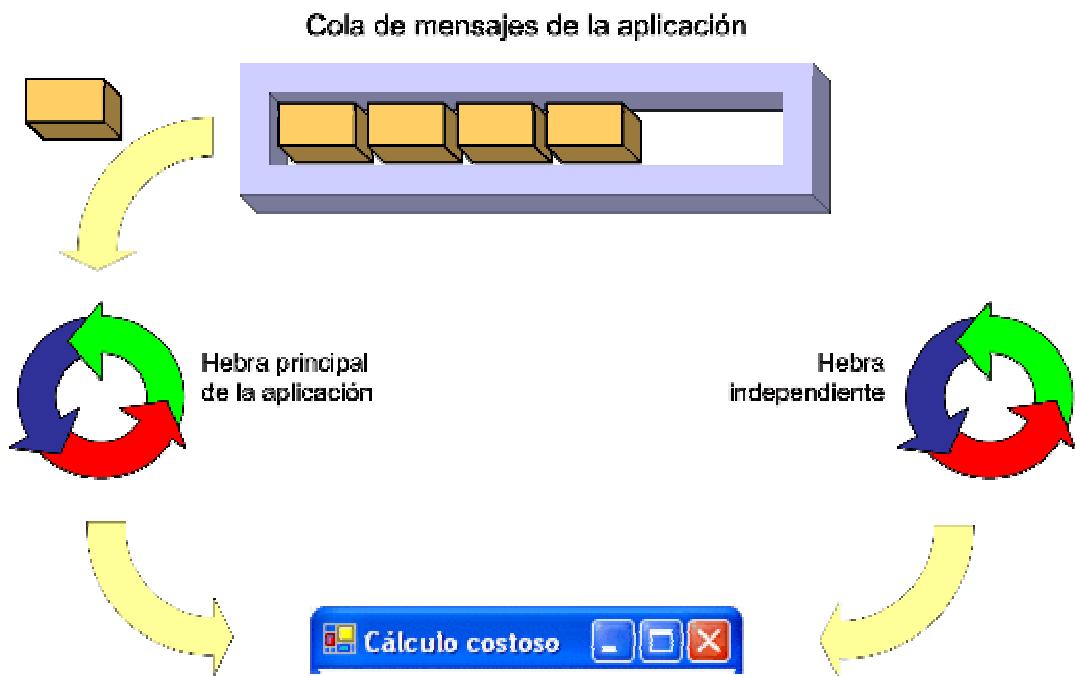
Por tanto, crearemos una hebra independiente cada vez que necesitemos realizar tareas largas (de más de un par de segundos):



Al usar una hebra auxiliar [*worker thread*], nuestra aplicación se comportará como cabría esperar...

... siempre que tengamos cuidado a la hora de utilizar recursos compartidos entre las distintas hebras.

Por ejemplo, si nuestras hebras auxiliares también acceden a los componentes Swing, estamos usando un recurso compartido:



Cuando esto sucede, pueden darse situaciones desagradables si no controlamos el acceso a los recursos compartidos.

La solución más sencilla consiste en eliminar el recurso compartido, de tal forma que sólo una de las hebras pueda acceder a lo que antes era un recurso compartido: se elimina el acceso concurrente a un recurso desde distintas hebras.

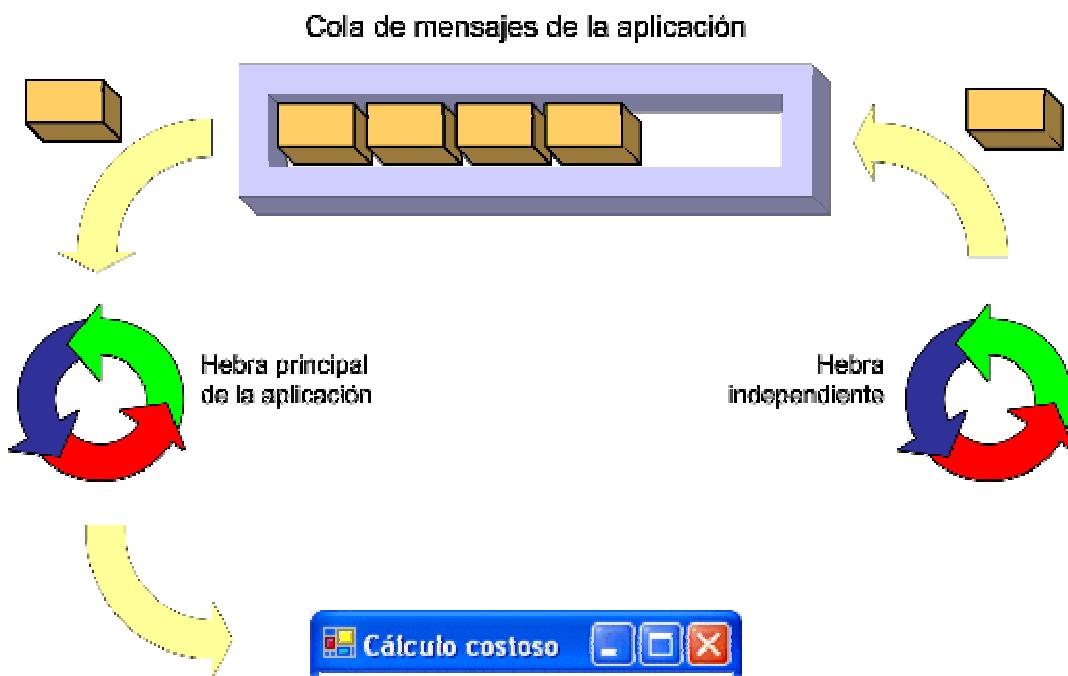
Por convención, la hebra responsable de gestionar los eventos de la interfaz de usuario será **siempre** responsable de manipular todos los componentes y controles de nuestra interfaz.

SwingUtilities.invokeLater()

`== EventQueue.invokeLater()`

SWING

AWT



Para hacer que la hebra encargada de los eventos se encargue también de ejecutar un fragmento de código, se utiliza el método `SwingUtilities.invokeLater()`.

El método `invokeLater()` recibe como parámetro un objeto que implemente la interfaz `Runnable`.

Por tanto, deberemos encapsular en el método `run()` de un objeto `Runnable` lo que queramos hacer sobre los componentes de la interfaz desde una hebra independiente.

Una variante de `invokeLater()` es el método `SwingUtilities.invokeAndWait()`, que detiene la ejecución de la hebra auxiliar hasta que la hebra principal de la interfaz haya ejecutado el código correspondiente al objeto `Runnable` que se le pasa como parámetro a `invokeAndWait()`.

javax.swing.Timer

Cuando lo que queremos es realizar una operación de forma periódica, no es necesario que creemos explícitamente una hebra.

Podemos utilizar la clase `javax.swing.Timer`, que asociaremos a un `ActionListener` encargado de realizar la operación:

```
// Animación a X fps (frames por segundo)

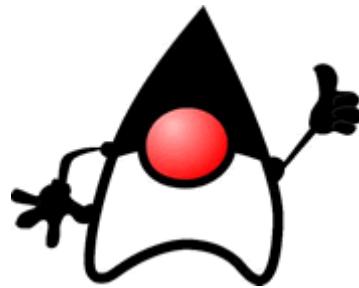
public class Animator
    extends JFrame
    implements ActionListener
{
    ...
    Timer timer;

    public Animator (int fps)
    {
        timer = new Timer(1000/fps, this);
        ...
    }

    public void startAnimation()
    {
        timer.start();
    }

    public void stopAnimation()
    {
        timer.stop();
    }

    public void actionPerformed (ActionEvent e)
    {
        // Mostrar el siguiente frame
        ...
        repaint();
    }
    ...
}
```



Más información...

The Java™ Tutorial

“Threads: Doing Two or More Tasks At Once”

<http://java.sun.com/docs/books/tutorial/essential/threads/>

Bruce Eckel:

“Thinking in Java” [3rd edition]. Chapter 13: Concurrency

<http://www.mindview.net/Books/TIJ/>

Concurrent Programming Using Java

<http://elvis.rowan.edu/~hartley/ConcProgJava/index.html>

Threads and Swing

<http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html>

Using a worker thread

<http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html>

Distribución

Mecanismos de comunicación entre procesos

Redes de ordenadores: Internet

Sockets

Sockets TCP

Creación de un cliente TCP

Creación de un servidor TCP

Sockets UDP

La clase URL

RMI (Remote Method Invocation)

Extensiones: Jini

Alternativas: CORBA y .NET Remoting

“the network is the computer”

Lema de Sun Microsystems

Comunicación entre procesos

El desarrollo de aplicaciones concurrentes involucra el uso de mecanismos de comunicación. Estos mecanismos, conocidos genéricamente como **mecanismos de comunicación entre procesos** (IPC) permiten que los distintos procesos que conforman una aplicación "hablen entre sí".

Los procesos en los que se descompone una aplicación pueden ejecutarse en un mismo ordenador (siempre que tengamos un sistema operativo multitarea) o en máquinas diferentes. En este último caso, la comunicación entre procesos involucra el uso de **redes de ordenadores**.

Mecanismos de comunicación entre procesos [IPC: InterProcess Communication]

Cuando en un sistema tenemos distintos procesos, necesitamos disponer de mecanismos que hagan posible la comunicación entre ellos. Se pueden utilizar distintos mecanismos de comunicación entre procesos:

- **Pipes anónimos [Anonymous pipes]:** Permiten redireccionar la entrada o salida estándar de un proceso (utilizando | en la línea de comandos, por ejemplo).
- **Sockets:** Usan la familia de protocolos TCP/IP (la que se utiliza en Internet). Su diseño original proviene del BSD UNIX [Berkeley Software Distribution].
- **Estándares de paso de mensajes** como MPI (Message Passing Interface, muy utilizado en clusters y supercomputadores) o PVM (Parallel Virtual Machine, otro estándar utilizado en multiprocesadores y multicamputadores).

 *Llamadas a procedimientos remotos* (RPC: Remote Procedure Call): Permiten realizar la comunicación entre procesos como si se tratase de simples llamadas a funciones.

- En Java, se utiliza un mecanismo conocido como *RMI* [*Remote Method Invocation*].
- En la plataforma .NET, se utiliza *.NET Remoting* (un mecanismo similar a RMI)
- El RPC de Windows cumple con el estándar OSF DCE [Open Software Foundation Distributed Computing Environment], lo que permite la comunicación entre procesos que se ejecuten en sistemas operativos diferentes a Windows.

 *Middleware*: Software que se utiliza para conectar los componentes de un sistema distribuido.

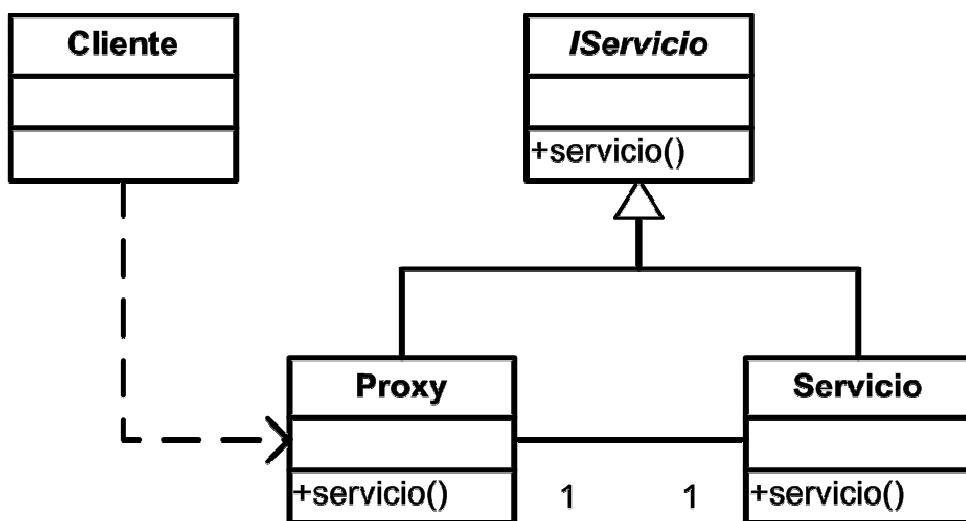
- **CORBA** [Common Object Request Broker Architecture], estándar del OMG [Object Management Group].
- **Servicios web** [Web services], promovidos por el W3C (el consorcio que propone los estándares usados en la web).
- **J2EE** [Java 2 Enterprise Edition] estandariza los servicios que ha de ofrecer un servidor de aplicaciones Java.
© Sun Microsystems
- **COM / DCOM** [Component Object Model / Distributed COM] establece un estándar binario mediante el cual se puede acceder a los servicios ofrecidos por un componente.
© Microsoft Corporation

Como se puede ver, disponemos de una amplia variedad de mecanismos de comunicación entre procesos (y eso sin contar los múltiples mecanismos específicos que ofrece cada sistema operativo).

En el caso particular de **Windows**: el *portapapeles*, *DDE* [*Dynamic Data Exchange*], *OLE* [*Object Linking and Embedding*], *ActiveX*, *Mailslots* (en Win32 y OS/2), el mensaje *WM_COPYDATA*, *ficheros mapeados en memoria*, *pipes con nombre* [*Named pipes*], semáforos, eventos, “*mutex*” y otras primitivas de sincronización...

Independientemente del mecanismo de comunicación entre procesos que decidamos emplear, nuestra aplicación debería acceder a los recursos externos de la misma forma que accede a recursos locales.

Para encapsular el acceso a recursos externos se suelen emplear proxies o gateways (para que en nuestra aplicación se pueda cambiar el mecanismo de comunicación entre procesos con el menor esfuerzo posible):



Siempre debemos tener en cuenta que...

- Los mecanismos de comunicación no siempre son fiables (algunos paquetes se pierden)
- La comunicación entre procesos consume tiempo (la latencia no es cero)
- La capacidad del canal de comunicación no es infinita (el ancho de banda es un recurso muy valioso)
- Las comunicaciones no siempre se realizan a través de medios seguros.

TERMINOLOGÍA: Los procesos de una aplicación distribuida suelen clasificarse como clientes o servidores, si bien pueden desempeñar ambos roles en distintos momentos. El **cliente** es el que solicita algún servicio proporcionado por otro proceso. El **servidor** es el que atiende las peticiones de los clientes.

Redes de ordenadores: Internet

TERMINOLOGÍA

Red de ordenadores

Conjunto de sistemas autónomos interconectados.

Protocolo

Conjunto de reglas para que emisor y receptor interpreten de forma adecuada los datos que se transmiten.

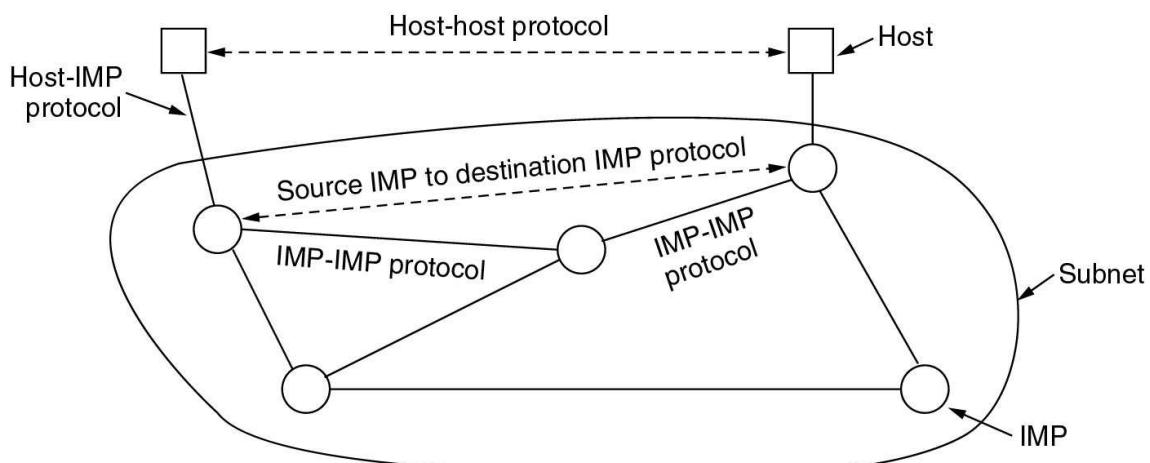
Origen de Internet

Proyecto del US Defense Advanced Research Project Agency (DARPA) para el desarrollo de su red ARPANET.

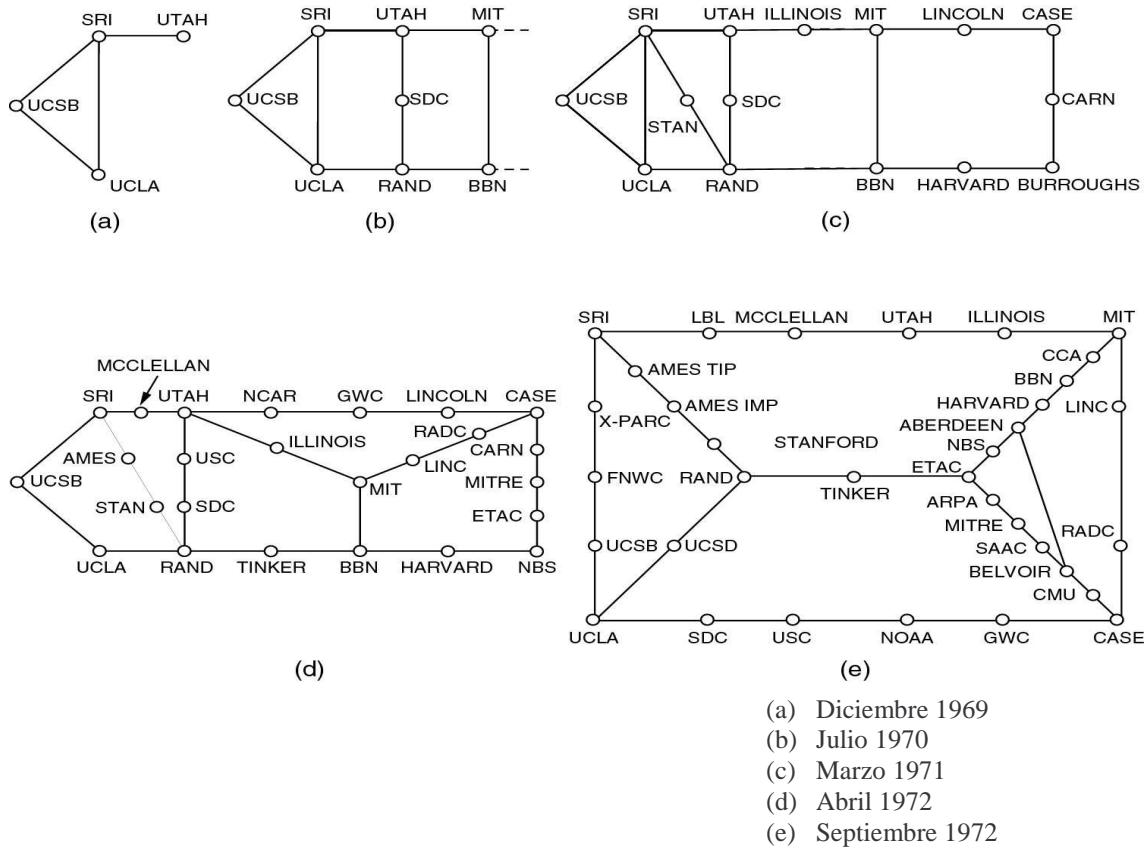
Objetivo

Tolerancia a errores en los elementos de la subred.

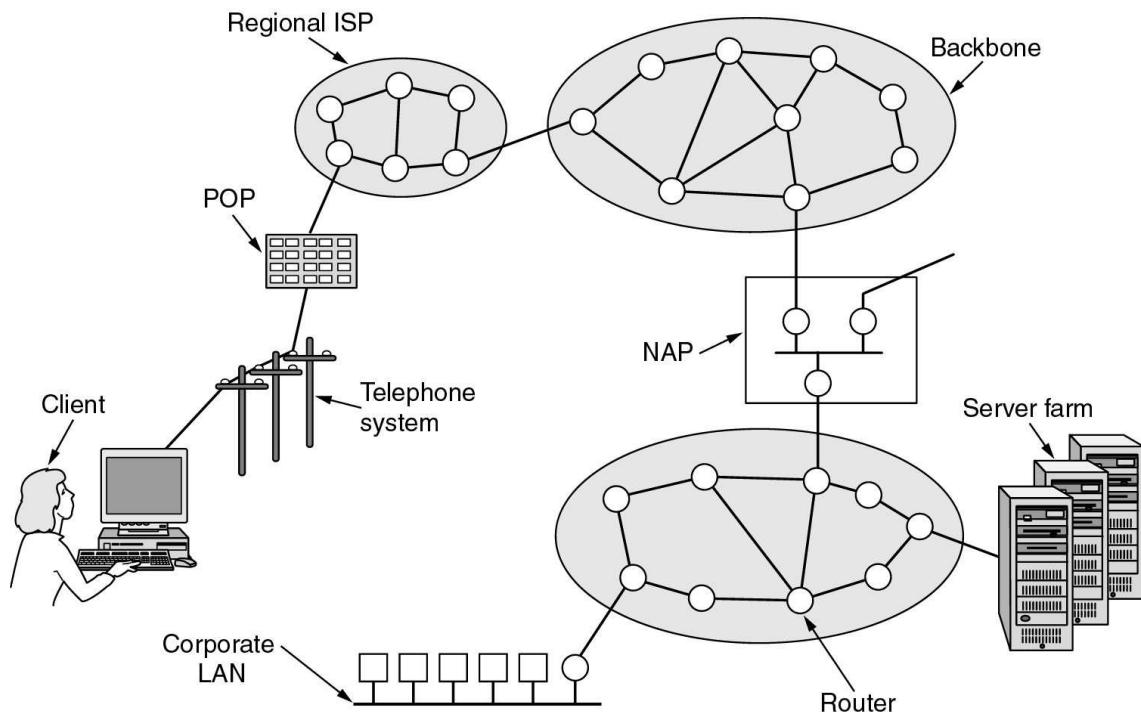
Diseño original de ARPANET



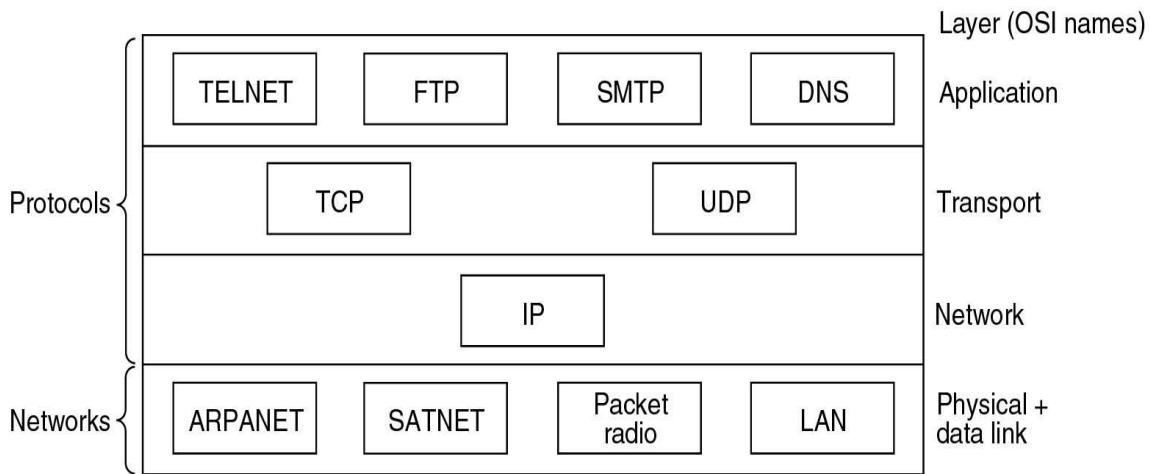
Evolución de ARPANET



Arquitectura actual de Internet

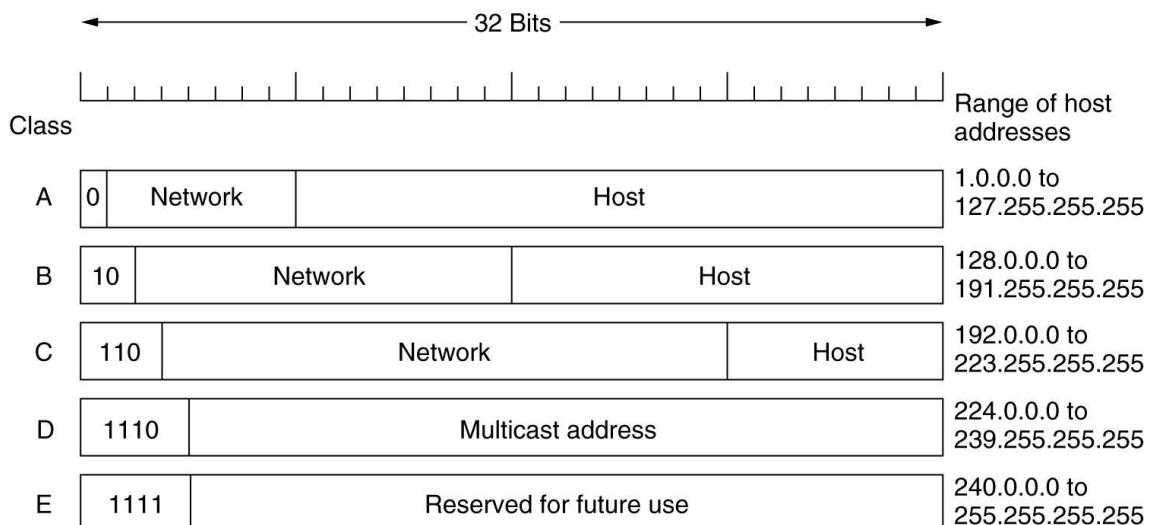


La familia de protocolos TCP/IP



El protocolo **IP [Internet Protocol]** se utiliza para transmitir paquetes (fragmentos de datos) de un ordenador a otro en Internet.

Para saber a dónde van dirigidos los datos, a cada ordenador se le asigna una dirección IP:



Los protocolos TCP y UDP permiten la existencia de varias conexiones con una misma dirección IP (*multiplexación de conexiones*). Además de la dirección IP, cuando queremos conectarnos con una aplicación hemos de especificar un número de puerto TCP o UDP.

El protocolo **TCP [Transmission Control Protocol]** proporciona un conjunto de primitivas de servicio (operaciones básicas) con las que se pueden construir aplicaciones que requieran servicios orientados a conexión (aquéllas en las que primero se establece una conexión y luego se transmiten los datos).

TCP también se encarga de reordenar los datos si éstos se reciben desordenados y de pedir automáticamente que se retransmitan los datos si se produce un error en la transmisión.

Las distintas aplicaciones que se usan en Internet se suelen construir sobre TCP y suelen estar asignadas a puertos estándar:

| Puerto | Protocolo | Uso |
|--------|-----------|--------------------------------|
| 21 | FTP | Transferencia de ficheros |
| 23 | Telnet | Acceso remoto |
| 25 | SMTP | Envío de correo electrónico |
| 79 | Finger | Información acerca de usuarios |
| 80 | HTTP | World Wide Web |
| 110 | POP3 | Lectura de correo electrónico |
| 119 | NNTP | Grupos de noticias USENET |
| ... | ... | ... |

El protocolo **UDP [User Datagram Protocol]** proporciona servicios no orientados a conexión, no garantiza la entrega de los paquetes, ni su llegada en orden, ni la no existencia de duplicados.

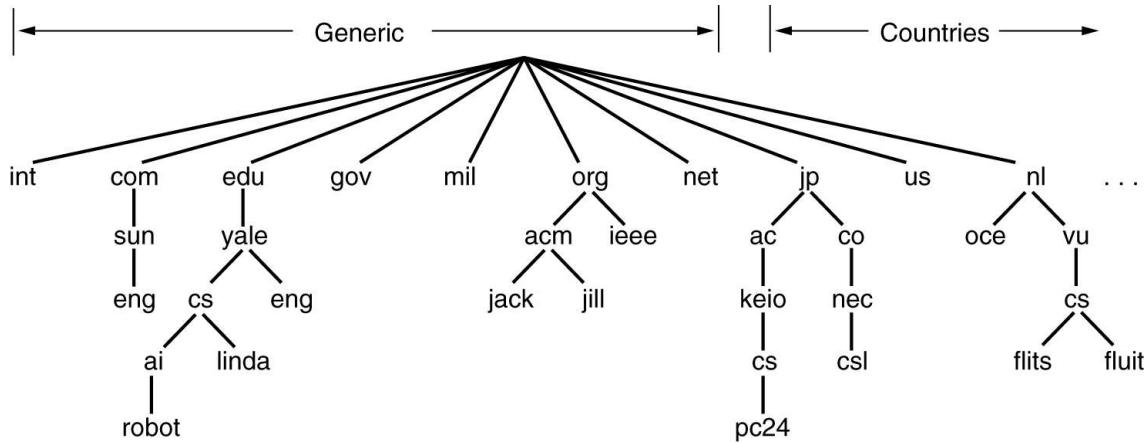
UDP se utiliza en algunas aplicaciones,
como **SNMP [Simple Network Management Protocol]**
o **RTP [Real-time Transport Protocol]**

ACLARACIÓN:

El teléfono proporciona un servicio orientado a conexión, mientras que el correo ofrece servicios no orientados a conexión.

El servicio de nombres **DNS [Domain Name Service]** es una aplicación que se utiliza en Internet para convertir un nombre (más fácil de recordar) en una dirección IP.

En Internet,
los nombres se agrupan en dominios de forma jerárquica:



Para identificar un recurso concreto en Internet, hay que especificar la dirección IP en la que se encuentra el recurso, el protocolo que se utiliza para acceder a él y el puerto a través del cual se establece la conexión. Toda esta información se recoge en una **URL [Uniform Resource Locator]**:

| Name | Used for | Example |
|--------|------------------|---|
| http | Hypertext (HTML) | http://www.cs.vu.nl/~ast/ |
| ftp | FTP | ftp://ftp.cs.vu.nl/pub/minix/README |
| file | Local file | file:///usr/suzanne/prog.c |
| news | Newsgroup | news:comp.os.minix |
| news | News article | news:AA0134223112@cs.utah.edu |
| gopher | Gopher | gopher://gopher.tc.umn.edu/11/Libraries |
| mailto | Sending e-mail | mailto:JohnUser@acm.org |
| telnet | Remote login | telnet://www.w3.org:80 |

Sockets

La biblioteca estándar de clases de Java nos proporciona todo lo que necesitamos para utilizar sockets en nuestras aplicaciones en el paquete `java.net`, por lo que tendremos que añadir la siguiente línea al comienzo de nuestros ficheros de código:

```
import java.net.*;
```

Los **sockets** son un mecanismo de comunicación entre procesos que se utiliza en Internet.

Un socket (literalmente, “enchufe”) es un extremo de una comunicación en Internet, por lo que se identifica con una dirección IP (un número entero de 32 bits) y un puerto (un número entero de 16 bits).

NOTA: Java encapsula el concepto de dirección IP con la clase `java.net.InetAddress`

Existen dos tipos de sockets, en función de si queremos utilizar TCP (orientado a conexión) o UDP (no orientado a conexión)

- `Socket` y `ServerSocket` se utilizan para establecer conexiones y enviar datos utilizando el protocolo TCP.
- `DatagramSocket` se utiliza para transmitir datos usando UDP.

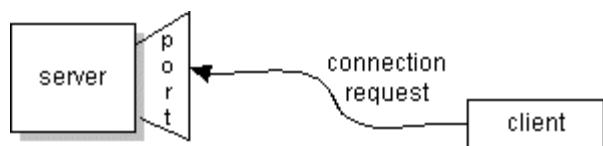
Sockets TCP

Las clases `Socket` y `ServerSocket` permiten utilizar el protocolo TCP en Java:

- Un `Socket` se utiliza para transmitir y recibir datos.
- Un `ServerSocket` nunca se utiliza para transmitir datos. Su único cometido es, en el servidor, esperar a que un cliente quiera establecer una conexión con el servidor.

Funcionamiento

El cliente crea un `Socket` para solicitar una conexión con el servidor al que desea conectarse.



Cuando el `ServerSocket` recibe la solicitud, crea un `Socket` en un puerto que no se esté usando y la conexión entre cliente y servidor queda establecida.



Entonces, el `SocketServer` vuelve a quedarse escuchando para recibir nuevas peticiones de clientes.

Transmisión de datos

Cuando la conexión queda establecida, los dos sockets conectados pueden comunicarse entre sí mediante operaciones de lectura y escritura idénticas a las cualquier otro “stream” en Java (`read` y `write`).

La entrada y la salida de un socket se representan en Java mediante las clases `InputStream` y `OutputStream`, respectivamente (las mismas que se utilizan para trabajar con ficheros).

Creación de un cliente TCP: Conexión a un servidor web

Un caso particular de socket TCP es el socket que se utiliza para conectarse a un servidor web. Los servidores web suelen escuchar peticiones en el puerto 80 TCP y emplean el protocolo HTTP.

Tras establecer la conexión TCP con el servidor web, se realiza una petición HTTP como la siguiente:

```
GET ejemplo.html^  
^
```

Si el fichero `ejemplo.html` existe, el servidor nos lo devuelve:

```
<HTML>  
  <HEAD>  
    <TITLE>Página web de ejemplo</TITLE>  
  </HEAD>  
  <BODY>  
    ...  
  </BODY>  
</HTML>
```

El siguiente fragmento de código muestra cómo nos podemos conectar a un servidor web para obtener un fichero de forma remota (/ en este caso, la página principal del servidor web):

```
Socket          socket;
PrintWriter     out;
InputStreamReader reader;
BufferedReader  in;
String          line;

// Conexión TCP

try {
    socket = new Socket("elvex.ugr.es", 80);
} catch (UnknownHostException e) {
    System.err.println("Host desconocido");
}

// Streams de E/S

out = new PrintWriter(socket.getOutputStream());
reader = new InputStreamReader(socket.getInputStream());
in = new BufferedReader(reader);

// Solicitud HTTP

out.println("GET /");
out.println("");
out.flush();

// Respuesta HTTP

line = in.readLine();
while (line!=null) {
    System.out.println(line);
    line = in.readLine();
}

// Cierre de la conexión

out.close();
in.close();
socket.close();
```

Creación de un servidor TCP: Servido de eco

```
import java.net.*;
import java.io.*;

class EchoServer
{
    public static void main( String args[ ] )
        throws IOException
    {
        ServerSocket    serverSocket;
        Socket         clientSocket;
        BufferedReader   in;
        PrintWriter     out;
        String          mensaje;
        Boolean         terminar = false;

serverSocket = new ServerSocket(4444);

while ( !terminar ) {

        clientSocket = serverSocket.accept();

        out = new PrintWriter(
                clientSocket.getOutputStream( ) );
        in = new BufferedReader(
                new InputStreamReader(
                    clientSocket.getInputStream( ) ) );

        mensaje = in.readLine();
        out.println(mensaje);
        out.flush();

        terminar = mensaje.equals("FIN");

        in.close();
        out.close();
        clientSocket.close();
    }

    serverSocket.close();
}
}
```

- El servidor anterior se instala en el puerto 4444 TCP y se queda esperando peticiones en la llamada al método `accept()`.
- Cuando se acepta la petición, a través del `InputStream` asociado al socket se lee una línea de texto enviada por el cliente.
- Esa misma línea de texto se le envía al cliente a través del `OutputStream` asociado al socket.
- Despues, se cierra la conexión con el cliente y el servidor se vuelve a quedar esperando la llegada de nuevas solicitudes de eco.
- El servidor seguirá ejecutándose hasta que reciba una petición de un cliente que solicite el eco de “FIN”.

Si queremos probar el servidor, podemos usar la utilidad `telnet` para conectarnos al puerto 4444 de la máquina local con:

```
telnet localhost 4444
```

Cualquier cliente Java puede acceder a nuestro servidor de eco utilizando el siguiente fragmento de código:

```
BufferedReader in;
PrintWriter      out;

Socket socket = new Socket("localhost",4444);

out = new PrintWriter(socket.getOutputStream());
in  = new BufferedReader(
          new InputStreamReader(
                  socket.getInputStream()));

out.println(args[0]);
out.flush();

String eco = in.readLine();
System.out.println(eco);

in.close();
out.close();
socket.close();
```

Creación de un servidor TCP: Servidor de hora

Wed May 18 18:18:18 CEST 2005

```
import java.net.*;
import java.io.*;

class DateServer
{
    public static void main( String args[] )
        throws IOException {

        ServerSocket serverSocket;
        Socket      clientSocket;
        PrintWriter out;

        serverSocket = new ServerSocket(666);

        try {
            while (true) { // ¡OJO!
                clientSocket = serverSocket.accept();
                out = new PrintWriter
                    (clientSocket.getOutputStream());
                out.println(new java.util.Date());
                out.flush();
                out.close();
                clientSocket.close();
            }
        } catch (IOException error) {
            serverSocket.close();
        }
    }
}
```

NOTA: El servidor es lo suficientemente simple como para atender todas las peticiones secuencialmente. Si el procesamiento de cada solicitud proveniente de un cliente fuese más costoso, deberíamos utilizar **hebras para atender en paralelo a distintos clientes** (y usaríamos protocolos algo más complejos para hacer algo más útil).

IMPORTANTE

Los servidores, usualmente, han de atender múltiples peticiones provenientes de distintos clientes, por lo que siempre debemos implementarlos como aplicaciones multihebra.

El pseudocódigo de cualquier servidor es siempre similar a:

```
while (!fin) {  
    // 1. Aceptar una solicitud  
    ...  
    // 2. Crear una hebra independiente  
    //      para procesar la solicitud  
    ...  
}
```

Sockets UDP

El tipo más sencillo de sockets, puesto que no es necesario establecer ninguna conexión para enviar y recibir datos.

En Java, un objeto de tipo `DatagramSocket` representa un socket UDP y puede enviar o recibir datos directamente de otro socket UDP.

Los datos se envían y reciben en paquetes autocontenidos denominados datagramas (igual que el correo convencional).

Los datagramas se representan en Java mediante la clase `DatagramPacket`, que consiste simplemente en un array de bytes dirigido a una dirección IP y a un puerto UDP concretos.

La clase `MulticastSocket` se puede emplear para enviar un mismo datagrama a muchos destinatarios simultáneamente.

Ejemplos disponibles en:

<http://java.sun.com/docs/books/tutorial/networking/datagrams/>

La clase URL

Cuando alguien accede a un servidor web, generalmente lo hace para obtener un documento, para lo que usa una URL (que en java se representa mediante la clase `java.net.URL`).

El siguiente fragmento de código muestra cómo podemos crear una URL y acceder a un recurso en Internet a partir de la URL:

```
URL url = new URL("http://elvex.ugr.es/");

// Datos de la URL

System.out.println("Protocolo = " + url.getProtocol());
System.out.println("Host = " + url.getHost());
System.out.println("Fichero = " + url.getFile());
System.out.println("Puerto = " + url.getPort());

// Acceso al contenido asociado a la URL

InputStreamReader reader = new InputStreamReader(
    url.openStream());

BufferedReader in = new BufferedReader(reader);

String inputLine;
while ((inputLine = in.readLine()) != null)
    System.out.println(inputLine);

in.close();
```

De esta forma no tenemos que conocer los detalles del protocolo HTTP que utilizan los servidores web (ni la dirección IP del servidor, ya que la clase URL se encarga de traducir el nombre del host a una dirección IP usando el servicio DNS).

NOTA:

Con `URL.openConnection()` podemos establecer una conexión TCP a través de la cual también se pueden transmitir datos.

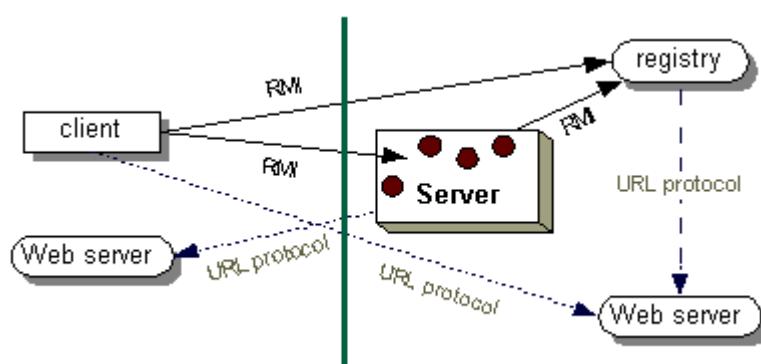
RMI

[*Remote Method Invocation*]

Cuando utilizamos sockets, hemos de preocuparnos de cómo se transmiten físicamente los datos entre los extremos de una conexión (a nivel de bytes, ya que usamos los “streams” estándar)

RMI permite olvidarnos de los detalles de la transmisión de datos y centrarnos en el diseño de la lógica de nuestra aplicación, puesto que nos permite acceder a un objeto remoto como si de un objeto local se tratase.

- Internamente, RMI utiliza **serialización** de objetos para encargarse de la transmisión de datos a través de la red (de cara al programador, el acceso al objeto remoto es como una llamada a un método local).
- Para localizar un objeto al que se desee acceder, RMI proporciona un **registro** que se usa a modo de páginas amarillas.
- Como respuesta de las llamadas a métodos de un objeto remoto, RMI devuelve objetos y se encarga de obtener los **bytecodes** que sean necesarios (cuando se obtiene una referencia a un objeto cuyos bytecodes no están disponibles en la máquina virtual del receptor).



En RMI:

- El servidor crea algunos objetos y los hace accesibles a través del registro. A continuación, se queda esperando a recibir peticiones.
- El cliente obtiene una referencia a un objeto remoto (que está alojado en el servidor) y la utiliza para invocar métodos del objeto de forma remota.

Objetos e interfaces remotos

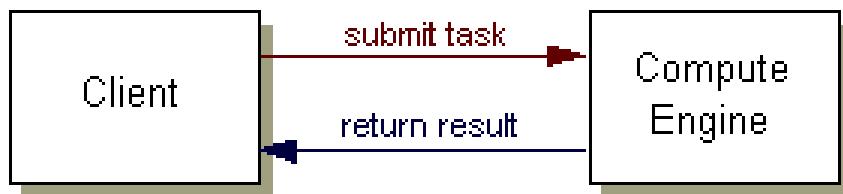
Para que un objeto sea accesible de forma remota, ha de implementar un interfaz remoto (derivado de `java.rmi.Remote`)

Cuando una llamada a un método realizada desde el cliente devuelve un objeto remoto, en vez de obtener una copia del objeto, se obtiene una referencia al objeto remoto (un *stub* que hace de *proxy* y se comporta igual que el objeto remoto).

Desarrollo de aplicaciones distribuidas con RMI

1. Crear los distintos componentes de la aplicación, teniendo en cuenta que, para que a un objeto se pueda acceder de forma remota con RMI, es necesario que implemente una interfaz derivada de `java.rmi.Remote`.
2. Compilar con `javac` y generar los stubs con `rmiic`.
3. Hacer accesibles los objetos remotos (usualmente, dejando los bytecodes correspondientes a los interfaces remotos y a los stubs en algún servidor web).
4. Arrancar la aplicación, que incluye el registro RMI, el servidor y el cliente.

Ejemplo: Plataforma distribuida de cómputo



El servidor RMI

1. Interfaz remota

Para los objetos a los que se accederá de forma remota:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote
{
    Object executeTask(Task t)
        throws RemoteException;
}
```

2. Objetos serializables

Aquellos que se transmitirán a través de la red:

```
import java.io.Serializable;

public interface Task extends Serializable {
    Object execute();
}
```

3. Implementación del servidor

```
import java.rmi.*;
import java.rmi.server.*;

public class ComputeEngine
    extends UnicastRemoteObject // Objeto remoto
    implements Compute // Interfaz remota
{
    public ComputeEngine()
        throws RemoteException
    {
        super();
    }

    public Object executeTask (Task t)
    {
        return t.execute();
    }

    // Programa principal

    public static void main(String[] args)
        throws Exception
    {
        System.setSecurityManager
            (new RMISecurityManager());
        String name = "//elvex.ugr.es/Compute";
        Compute engine = new ComputeEngine();
        Naming.rebind(name, engine);
    }
}
```

- El SecurityManager se encarga de controlar las acciones realizadas por el código que se descarga a través de la red (si no creamos uno, no se podrá descargar código remoto).
- Naming.rebind() registra un objeto para que se pueda acceder a él de forma remota a través de una URL.
- Mientras que exista alguna referencia a ese objeto (aunque sea la del registro), el servidor no finalizará su ejecución.

El cliente RMI

```
import java.rmi.*;
import java.math.*;

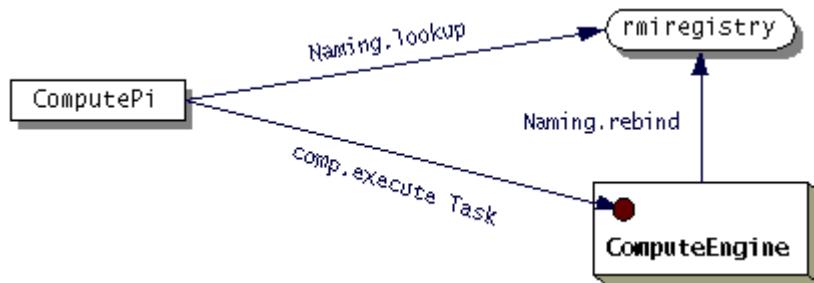
public class ComputePi {

    public static void main(String args[])
    {
        System.setSecurityManager
            (new RMISecurityManager());

        String name = "//" + args[0] + "/Compute";
        Compute comp = (Compute) Naming.lookup(name);

        ...
        tarea = new XTask();
        resultado = (XResult) comp.executeTask(task);

        System.out.println(resultado);
    }
}
```



NOTA

La característica más destacable de RMI es que el servidor no tiene por qué disponer de antemano de los bytecodes asociados a la tarea que ha de ejecutar.

Cuando recibe la petición del cliente, carga los bytecodes en su máquina virtual, ejecuta la tarea y devuelve el resultado.

Extensiones de RMI: Jini

Jini está montado sobre RMI y permite descubrir dinámicamente qué dispositivos y servicios están disponibles en la red, sin tener que saber de antemano su localización exacta (“plug & play”).



Alternativas a RMI: CORBA y .NET Remoting

CORBA

RMI se puede interpretar como una versión simplificada de CORBA (un estándar complejo que permite la implementación de sistemas distribuidos basados en objetos, independientemente del lenguaje de programación que se utilice para implementar las distintas partes del sistema).

NOTA: RMI sólo puede utilizarse en Java, por lo que no sirve para conectar una aplicación Java a un sistema no escrito en Java.

.NET Remoting

La plataforma .NET (la alternativa de Microsoft a Java), incluye un mecanismo similar a RMI que se denomina .NET Remoting.

Más información en

<http://elvex.ugr.es/decsai/csharp/distributed/remoting.xml>

Más información...

Essentials of the Java Programming Language, Part 2
“Sockets Communications”

<http://java.sun.com/developer/onlineTraining/Programming/BasicJava2/socket.html>

The JavaTM Tutorial
“Custom Networking: All about sockets”
<http://java.sun.com/docs/books/tutorial/networking/>

The JavaTM Tutorial
“RMI”
<http://java.sun.com/docs/books/tutorial/rmi/>

