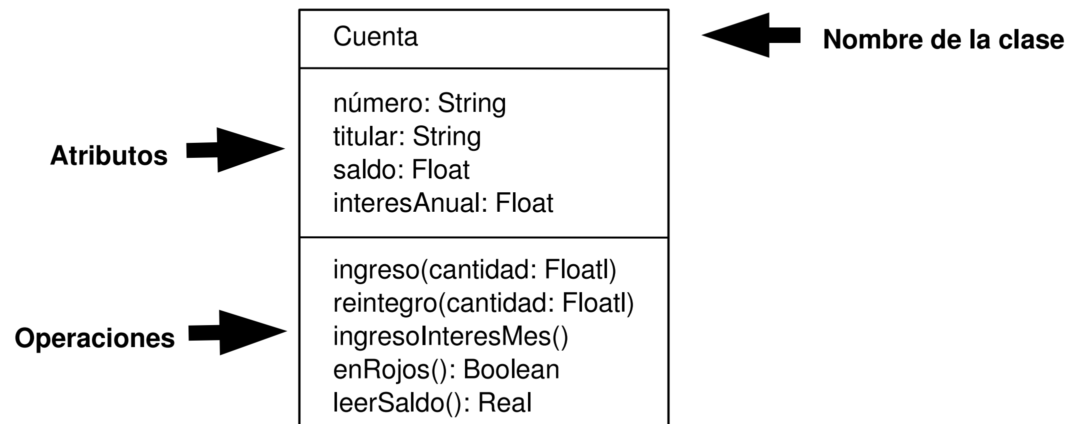
Tema 2: Programación orientada a objetos en Java

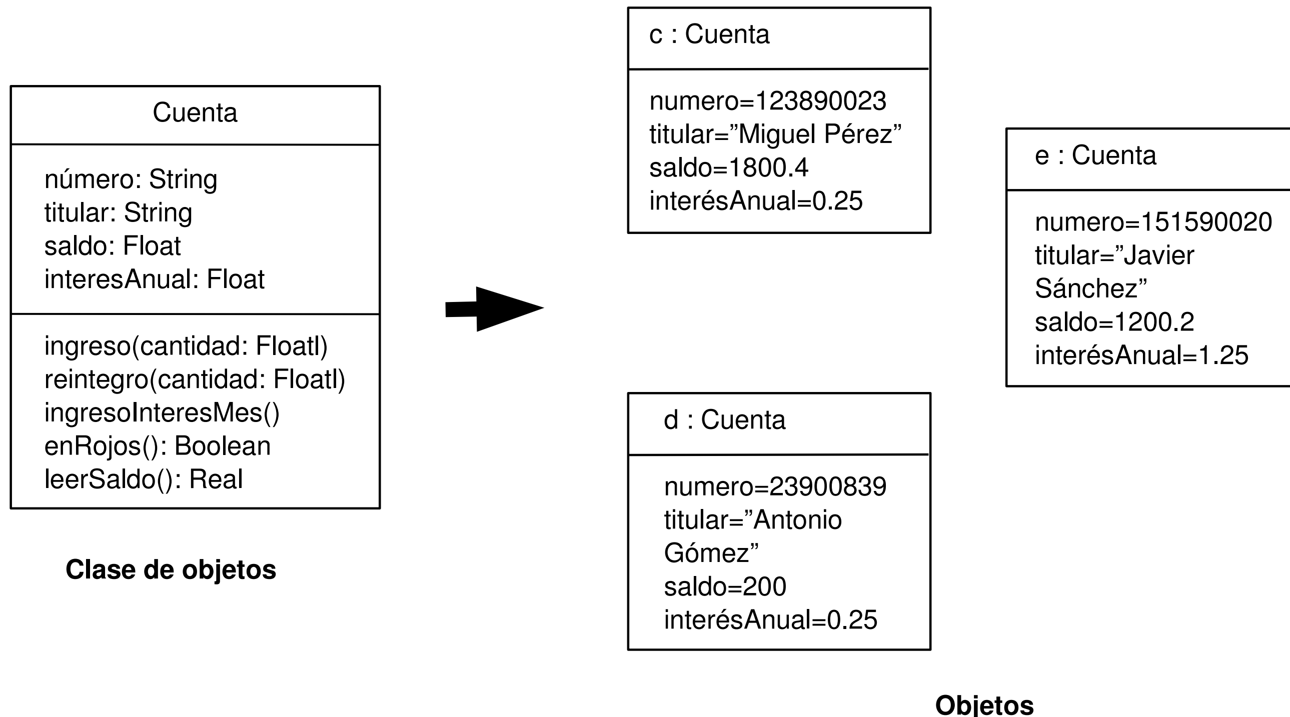
1. Clases de objetos
2. Protección de miembros
3. Protección de clases
4. Inicialización y finalización
5. Creación de objetos
6. Trabajando con objetos
7. Relaciones entre objetos
8. Clases anidadas e interiores
9. Autoreferencias
10. Aplicaciones orientadas a objetos
11. Herencia
12. Adición, redefinición y anulación
13. Protección y herencia
14. Herencia múltiple
15. Polimorfismo
16. Ligadura dinámica
17. Información de clases en tiempo de ejecución
18. Otros temas de interés en Java

Clases de objetos

- Las clases de objetos representan conceptos o entidades significativos en un problema determinado.
- Una clase describe las características comunes de un conjunto de objetos, mediante dos elementos:
  - Atributos (o variables miembro, variables de clase). Describen el estado interno de cada objeto
  - Operaciones (o métodos, funciones miembro). Describen lo que se puede hacer con el objeto, los servicios que proporciona



- Durante la ejecución de la aplicación se producirá la instanciación de la clase, es decir, la creación de los objetos que representan cada uno de los individuos con sus características propias, es decir, valores específicos para sus atributos



- La implementación de esta clase en Java se realizaría en un fichero con nombre Cuenta.java, y su contenido sería el siguiente:

Atributos →

Operaciones →

```
class Cuenta {
    long numero;
    String titular;
    float saldo;
    float interesAnual;

    void ingreso(float cantidad) { }
    void reintegro(float cantidad) { }
    void ingresoInteresMes() { }
    boolean enRojos() { }
    float leerSaldo() { }
}
```

•Los atributos pueden ser de cualquiera de los tipos básicos de Java: boolean, char, byte, short, int, long, float y double, referencias a otros objetos o arrays de elementos de alguno de los tipos citados

- Al contrario que C++, Java realiza la definición e implementación de la clase en el mismo lugar, en un único fichero .java

```
class Cuenta {
    long numero;
    String titular;
    float saldo;
    float interesAnual;

    void ingreso(float cantidad) {
        saldo += cantidad;
    }

    void reintegro(float cantidad) {
        saldo -= cantidad;
    }

    void ingresoInteresMes() {
        saldo += interesAnual * saldo / 1200;
    }

    boolean enRojos() { return saldo < 0; }
    float leerSaldo() { return saldo; }
}
```

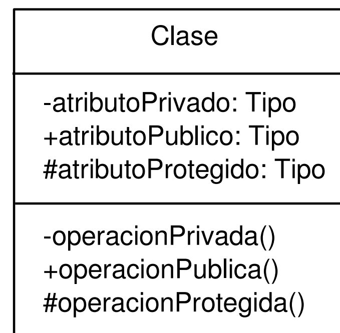
- El acceso a los atributos de la clase desde la implementación de las operaciones se realiza de forma directa
- Los atributos u operaciones estáticas (static) no son afectados por el proceso de instanciación de objetos a partir de la clase
- De un atributo estático no se genera una copia por cada objeto que se crea. Existe una única copia compartida y accesible desde todos los objetos de la clase
- Una operación estática únicamente puede acceder a miembros estáticos

- El atributo *nOp* mantiene una cuenta global del número de operaciones realizadas en las cuentas del banco, para la realización de estadísticas. La operación *leerNOperaciones()* permite leer este contador
- La operación *eurosAPesetas()* es una operación auxiliar de la clase *Cuenta* para ser usada cuando sea necesaria una conversión de moneda

```
class Cuenta {
    long numero;
    String titular;
    float saldo;
    float interesAnual;

    // Contador de operaciones
    static int nOp = 0;

    static int leerNOperaciones() { return nOp; }

    // Operación estática auxiliar de conversión
    static long eurosAPesetas(float euros) { return euros * 166.386f; }

    void ingreso(float cantidad) { saldo += cantidad; ++nOp; }

    void reintegro(float cantidad) { saldo -= cantidad; ++nOp; }
}
```

Protección de miembros de la clase

- El principio de ocultación de información se plasma en los lenguajes OO en diversos mecanismos de protección de los miembros de la clase
- UML permite asociar tres niveles de protección diferentes a cada miembro de la clase:
  - Miembros públicos (+). Sin ningún tipo de protección especial
  - Miembros privados (-). Inaccesibles desde el exterior de la clase
  - Miembros protegidos (#). Similares a los privados aunque se permite su acceso desde las clases descendientes*



*** Las veremos más adelante, al estudiar el mecanismo de la herencia**

- En Java un miembro se etiqueta como público colocando el identificador public delante de su declaración
- Para los miembros privados utilizaremos el identificador private

Cuenta
-numero: Long -titular: String -saldo: Float -interésAnual: Real
+ingreso(cantidad: Integer) +reintegro(cantidad: Integer) +ingresoInteresMes() +enRojos(): Boolean +leerSaldo(): Integer

```
class Cuenta {
    private long numero;
    private String titular;
    private float saldo;
    private float interesAnual;

    public void ingreso(float cantidad) {
        saldo += cantidad;
    }

    public void reintegro(float cantidad) {
        saldo -= cantidad;
    }

    public void ingresoInteresMes() {
        saldo += interesAnual * saldo / 1200;
    }

    public boolean enRojos() {  return saldo < 0; }
    public float leerSaldo() { return saldo; }
}
```

- Los miembros no etiquetados son accesibles por parte de clases amigas. En C++ y otros lenguajes OO las clases amigas a una dada pueden indicarse explícitamente
- En Java se consideran amigas todas aquellas que forman parte del mismo paquete
  - Un fichero fuente java forma en sí un paquete y por tanto todas las clases incluidas en él son amigas
  - Las clases incluidas en varios ficheros fuente pueden agruparse en un único paquete indicando el nombre de paquete al principio de cada fichero mediante el indicador package

```
package prueba;

class A {
  ...
}

class B {
  ...
}
```

```
package prueba;

class C {
  ...
}
```

```
class D {
  ...
}

class E {
  ...
}
```

Las clases A, B y C son amigas al pertenecer al mismo paquete "prueba"

Las clases D y E son amigas al pertenecer al mismo fichero fuente

- En este ejemplo, las clases *Cuenta* y *Banco* son amigas al pertenecer al mismo fichero fuente. El acceso a los atributos de los objetos de la clase *Cuenta* almacenados en el vector interno de *Banco* queda así garantizado. El atributo *saldo* puede mantenerse como privado puesto que existe una operación que permite obtener su valor:

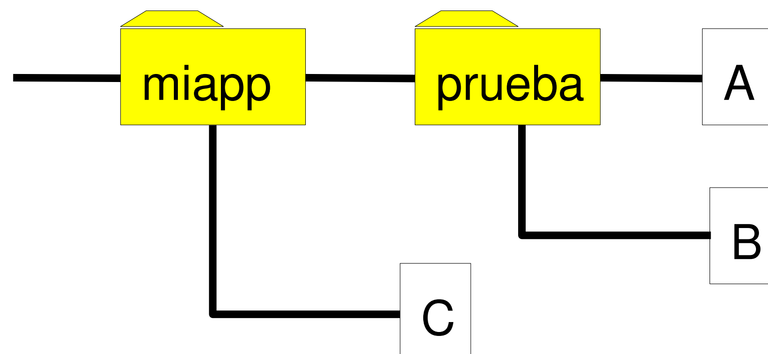
```
class Cuenta {
    long numero;
    String titular;
    private float saldo;
    float interesAnual;

    public void ingreso(float cantidad) {
        saldo += cantidad;
    }

    public void reintegro(float cantidad) {
        saldo -= cantidad;
    }

    public void ingresoInteresMes() {
        saldo += interesAnual * saldo / 1200;
    }

    public boolean enRojos() { return saldo < 0; }
    public float leerSaldo() { return saldo; }
}

class Banco {
    Cuenta[] c; // vector de cuentas
     ...
}
```

•Atención: todas las clases que no se declaren como pertenecientes a ningún paquete de forma explícita, pertenecen a un paquete "por defecto" y por tanto son amigas

•Un paquete crea un espacio de nombres propios. Esto significa que la clase pasa a tener como prefijo el propio nombre del paquete. A la hora de utilizarla tenemos tres opciones:

- Utilizar su nombre completo: prueba.A
- Importar esa clase, para poder utilizarla sin el prefijo. Esto se indica al principio del código fuente mediante import prueba.A
- Importar directamente todas las clases del paquete, que se usarían sin prefijo: import prueba.*

- Un paquete puede estar situado dentro de otro paquete formando estructuras jerárquicas. Ejemplo: *miapp.prueba.A*
- Java obliga a que exista una correspondencia entre la estructura de paquetes de una clase y la estructura de directorios donde está situada
- La raiz de la estructura de directorios debe estar incluida en el classpath de Java (parámetro -cp <dir>)
  - Las clases *miapp.prueba.A*, *miapp.prueba.B* y *miapp.C* deben estar en la siguiente estructura de directorios:



Protección de clases

- Por protección de clases entendemos un nivel superior de la ocultación de información, a nivel de clases. Es decir, se trata de especificar que clases pueden ser utilizadas y cuales no, y por quién
- Dentro de un paquete, las clases son amigas y por tanto no existen restricciones respecto a la utilización de una clase por las otras
- Sin embargo, desde el punto de vista del exterior, únicamente podrán ser utilizadas las clases públicas del paquete, es decir, aquellas con el identificador public situado delante de su declaración

- Atención: Java sólo permite una clase pública por fichero fuente, y el nombre de la clase y el fichero deben coincidir obligatoriamente
  - En nuestro ejemplo, si queremos que la clase *Cuenta* pueda ser utilizada desde el exterior del fichero *Cuenta.java* deberemos declararla como pública

```
public class Cuenta {
    private long numero;
    private String titular;
    private float saldo;
    private float interesAnual;

    public void ingreso(float cantidad) {
        saldo += cantidad;
    }

    public void reintegro(float cantidad) {
        saldo -= cantidad;
    }

    public void ingresoInteresMes() {
        saldo += interesAnual * saldo / 1200;
    }

    public boolean enRojos() { return saldo < 0; }
    public float leerSaldo() { return saldo; }
}
```

Inicialización y finalización

•La iniciación de los atributos de la clase se realiza en Java, al igual que en C++, mediante el uso de constructores cuyo nombre coincide con el de la clase

```
public class Cuenta {
    private long numero;
    private String titular;
    private float saldo;
    private float interesAnual;

    Cuenta(long aNumero, String aTitular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = aInteresAnual;
     }

     public void ingreso(float cantidad) {
         saldo += cantidad;
     }

     // Resto de operaciones de la clase Cuenta a partir de aquí
```

- Java permite la sobrecarga de operaciones, por tanto se pueden definir varios constructores posible para una clase siempre que se diferencien en la lista de argumentos

```
// Importar todas las clases del paquete java.io
import java.io.*;

public class Cuenta {
    private long numero;
    private String titular;
    private float saldo;
    private float interesAnual;

  // Constructor general
    Cuenta(long aNumero, String aTitular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = aInteresAnual;
    }
```

```
  // Constructor para obtener los datos de la cuenta de un fichero
     Cuenta(long aNumero) throws FileNotFoundException, IOException,
         ClassNotFoundException {

         FileInputStream fis = new FileInputStream(aNumero + ".cnt");
         ObjectInputStream ois = new ObjectInputStream(fis);
         numero = aNumero;
         titular = (String) ois.readObject();
         saldo = ois.readFloat();
         interesAnual = ois.readFloat();
         ois.close();
     }

     public void ingreso(float cantidad) {
         saldo += cantidad;
     }

     public void reintegro(float cantidad) {
         saldo -= cantidad;
     }

     public void ingresoInteresMes() {
         saldo += interesAnual * saldo / 1200;
     }

     public boolean enRojos() { return saldo < 0; }
     public float leerSaldo() { return saldo; }
}
```

Nota: véase el apartado "I/O: Reading and Writing" del tutorial Java de Sun como apoyo para entender el código del nuevo constructor

- Si no se proporciona ningún constructor, Java proporciona automáticamente un constructor por defecto, que no recibe argumentos y realiza una inicialización por defecto de los atributos
- Una vez implementado un constructor propio por parte del programador, Java elimina dicho constructor, aunque puede ser definido nuevamente de manera explícita

```
Cuenta() {
    numero = "00000000";
    titular = "ninguno";
    saldo = 0;
    interesAnual = 0;
}
```

•Naturalmente los constructores pueden ser marcados como públicos, privados, protegidos o con acceso a nivel de paquete, lo que especificará quien puede crear objetos de esta clase y de que manera

```
// Constructor general
  public Cuenta(long aNumero, String aTitular, float aInteresAnual) {
      numero = aNumero;
      titular = aTitular;
      saldo = 0;
      interesAnual = aInteresAnual;
  }

  // Constructor para obtener los datos de la cuenta de un fichero
  public Cuenta(long aNumero) throws FileNotFoundException,
        IOException, ClassNotFoundException {
      FileInputStream fis = new FileInputStream(aNumero + ".cnt");
      ObjectInputStream ois = new ObjectInputStream(fis);
      numero = aNumero;
      titular = (String)ois.readObject();
      saldo = ois.readFloat();
      interesAnual = ois.readFloat();
      ois.close();
  }
```

- Cuando finaliza el uso de un objeto, es frecuente la realización de ciertas tareas antes de su destrucción, principalmente la liberación de la memoria solicitada durante su ejecución. Esto se realiza en C++ y otros lenguajes OO en los denominados destructores
- Estos destructores son operaciones invocadas automáticamente justo antes de la destrucción del objeto
- Sin embargo, en Java la liberación de memoria se realiza de manera **automática** por parte del recolector de basura, por tanto la necesidad de este tipo de operaciones no existe en la mayor parte de los casos

- Sin embargo sí puede ser necesario realizar alguna tarea no relacionada con la liberación de memoria antes de la destrucción del objeto, como por ejemplo salvar el estado de la clase en un fichero o base de datos
- Java permite introducir código para este fin implementando una operación pública especial denominada finalize. Esta operación es invocada automáticamente antes de la destrucción del objeto por parte del recolector de basura

► Siguiendo nuestro ejemplo, vamos asegurarnos de que el estado de una cuenta queda salvado en disco antes de su destrucción, para poder ser recuperada posteriormente. Para ello introducimos el código de escritura en fichero en la operación *finalize* de la clase *Cuenta*

```
public void finalize() : throws FileNotFoundException, IOException {

    FileOutputStream fos = new FileOutputStream(numero + ".cnt");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(titular);
    oos.writeFloat(saldo);
    oos.writeFloat(interesAnual);
    oos.close();
}
```

- Problema: no es posible saber con seguridad en que momento será invocada *finalize*, puesto que el recolector de basura puede decidir su eliminación en un momento indeterminado, e incluso no ser eliminado hasta el final de la ejecución de la aplicación

- Una posible solución es ordenar al recolector de basura que realice una limpieza de memoria inmediata, para asegurar la finalización de los objetos. Esto se realiza mediante *Runtime.getRuntime().gc()*
- Por motivos de eficiencia, lo anterior es poco recomendable, sobre todo si se hace con frecuencia
- Una mejor opción es definir una operación ordinaria con este mismo propósito, a llamar de manera explícita cuando haya finalizado el uso del objeto
  - Introduciremos en la clase *Cuenta* una operación pública *salvar* en lugar de *finalize*, con la misma implementación. Tras finalizar las operaciones sobre la cuenta, invocaremos a *salvar* para guardar los cambios realizados

Creación de objetos

- En Java los objetos se crean únicamente de forma dinámica. Para ello se utiliza el operador new, de manera similar a C++
- Los objetos en Java se utilizan siempre a través de referencias. Las referencias son similares a los punteros de C/C++, aunque su uso es mucho más sencillo
- Por tanto los pasos a seguir en la creación de un objeto son:
  - Declarar una referencia a la clase
  - Crear un objeto mediante el operador new invocando al constructor adecuado
  - Conectar el objeto con la referencia

- La creación de un objeto de la clase Cuenta se realizaría de la siguiente forma:

```
Cuenta c; // Una referencia a un objeto de la clase Cuenta
c = new Cuenta(18400200, "Pedro Jiménez", 0.1f);
```

- En cambio, los tipos básicos (int, long, float, etc.) sí pueden ser creados directamente en la pila. Esto es posible porque Java no los implementa realmente como clases de objetos, por motivos de eficiencia y comodidad, ya que su uso es muy frecuente

```
Cuenta c;
float in;
long num;

in = 0.1f;
num = 18400200;

c = new Cuenta(num, "Pedro Jiménez", in);
```

- Las cadenas de caracteres se implementan con una clase (String). Sin embargo no suele ser necesaria su creación de manera explícita, ya que Java lo hace de manera automática al asignar una cadena constante*

```
String s; // Una referencia a un objeto de la clase String

// Conexión de la referencia s con un objeto String
// creado dinámicamente e inicializado con la constante "Pedro"
s = "Pedro";

// Sería equivalente a:
// char[] cc = {'P', 'e', 'd', 'r', 'o'}
// s = new String(cc);
```

- Los arrays también deben ser creados dinámicamente con *new* como si fueran objetos

```
int[] v; // Una referencia a un vector de enteros
v = new int[10] // Creación de un vector de 10 enteros
```

*** Véase el apartado *Characters and Strings* del tutorial de Java de Sun para más información**

- Si el array es de referencias a objetos, habrá que crear además cada uno de los objetos referenciados por separado

```
Cuenta[] v; // Un vector de referencias a objetos de la clase Cuenta
int c;

v = new Cuenta[10] // Crear espacio para 10 referencias a cuentas
for (c = 0; c < 10; c++)
    v[c] = new Cuenta(18400200 + c, "Cliente n. " + c, 0.1f);
```

- La destrucción de los objetos se realiza de manera automática cuando el recolector de basura detecta que el objeto no está siendo usado, es decir, no está conectado a ninguna referencia

```
Cuenta c1 = new Cuenta(18400200, "Cliente 1", 0.1f);
Cuenta c2 = new Cuenta(18400201, "Cliente 2", 0.1f);

c1 = c2
// El objeto asociado a la cuenta 18400200 ha
// quedado desconectado y será eliminado por el
// recolector de basura
```

Trabajando con objetos

- Trabajar con un objeto Java es similar a C++, aunque las referencias permiten un uso mucho más sencillo

```
Cuenta c1 = new Cuenta(18400200, “Pedro Jiménez”, 0.1f);
Cuenta c2 = new Cuenta(18400201);

c2.reintegro(1000);
c1.ingreso(500);

if (c2.enRojos())
    System.out.println(“Atención: cuenta 18400201 en números rojos”);

System.out.println(“Saldo actual de la cuenta 18400201: ” + c1.leerSaldo());

c1 = new Cuenta(18400202);
// El objeto asociado a la Cuenta 18400200 queda desconectado
c1.ingreso(500);

System.out.println(“Saldo actual de la cuenta 18400202: ” + c1.leerSaldo());
```

- En este ejemplo se pide un número de cuenta al usuario y una cantidad a retirar. A continuación se carga la cuenta solicitada y se realiza el reintegro

```
BufferedReader br = new BufferedReader(InputStreamReader(System.in));

long nc;
float mi;
try {
    System.out.println("Introduzca núm. de de cuenta: ");
    nc = Long.parseLong(br.readLine());

    System.out.println("Introduzca importe a retirar: ");
    mi = Float.parseFloat(br.readLine());
}
catch(Exception e) {
    System.out.println("Error al leer datos");
    return;
}

Cuenta c;
try {
    c = new Cuenta(nc);
}
catch(Exception e) {
    System.out.println("Imposible recuperar cuenta");
    return;
}

if (c.leerSaldo() < mi)
    System.out.println("Saldo insuficiente");
else
    c.reintegro(mi);
c.salvar();
```

- Naturalmente el compilador producirá un error ante cualquier acceso ilegal a un miembro de la clase

```
Cuenta c = new Cuenta(18400200, "Pedro Jiménez", 0.1f);

c.saldo = 1000;
```

```
Cuenta.java:XX: saldo has private access

c.saldo = 1000;
 ^
1 error
```

- El acceso a un miembro estático se realiza utilizando el nombre de la clase en lugar de un objeto

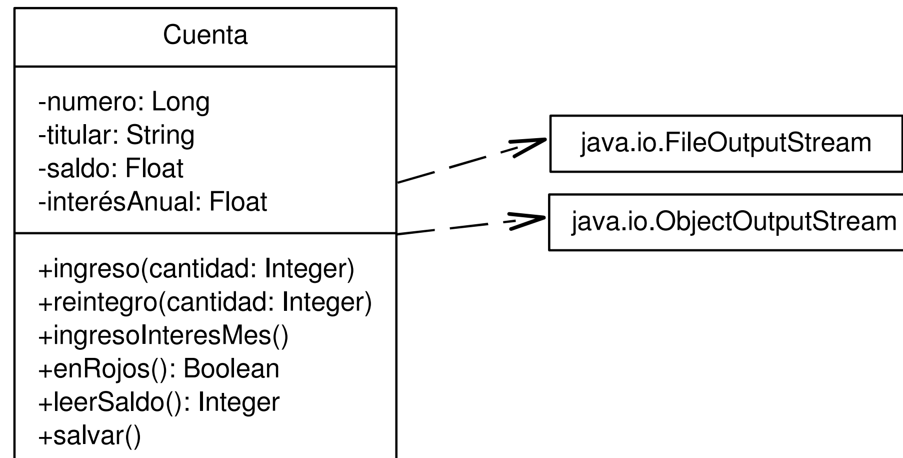
```
Cuenta c = new Cuenta(18400200, "Cliente 1", 0.1f);

c.ingreso(1000);
int pts = Cuenta.eurosAPesetas(c.leerSaldo());

System.out.println("Saldo: " + c.leerSaldo() + "(" + pts + " pesetas");
```

Relaciones entre objetos

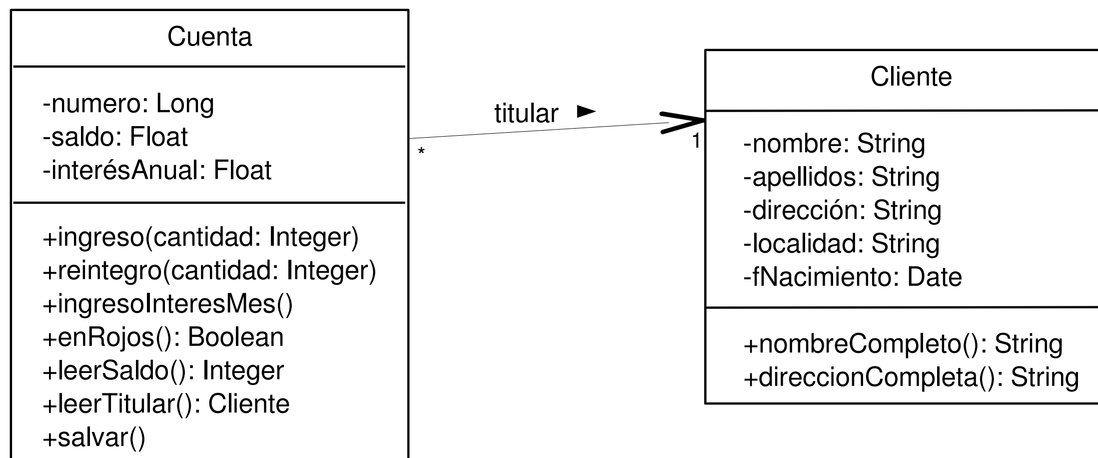
- Un conjunto de objetos aislados tiene escasa capacidad para resolver un problema. En una aplicación real los objetos colaboran e intercambian información, existiendo distintos tipos de relaciones entre ellos
- A nivel de diseño, podemos distinguir entre 5 tipos de relaciones básicas entre clases de objetos: dependencia, asociación, agregación, composición y herencia*

* La veremos más adelante, en un apartado específico

- La dependencia es la relación menos importante. Simplemente refleja que la implementación de una clase depende de otra
- Una dependencia puede indicar la utilización de un objeto de una clase como argumento de una operación de otra o en su implementación
  - Como vimos anteriormente, la clase *Cuenta* requiere las clases *FileOutputStream* y *ObjectOutputStream* de la librería de clases de Java para la implementación de la operación *salvar*



- En cambio, la asociación es la relación más importante y común. Refleja una relación entre dos clases independientes que se mantiene durante la vida de los objetos de dichas clases o al menos durante un tiempo prolongado
- En UML suele indicarse el nombre de la relación, el sentido de dicha relación y las cardinalidades en los dos extremos
  - Vamos a sustituir el atributo *titular* por una asociación con una nueva clase Cliente completa

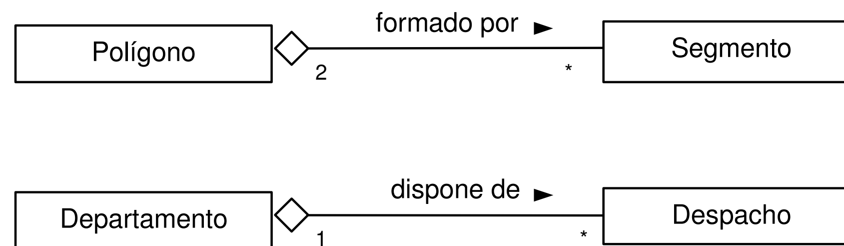


- Una asociación se implementa en Java introduciendo referencias a objetos una clase como atributos en la otra
- Si la relación tiene una cardinalidad superior a uno entonces será necesario utilizar un array de referencias. También es posible utilizar una estructura de datos dinámica del paquete *java.util* como *Vector* o *LinkedList* para almacenar las referencias
- Normalmente la conexión entre los objetos se realiza recibiendo la referencia de uno de ellos en el constructor o una operación ordinaria del otro

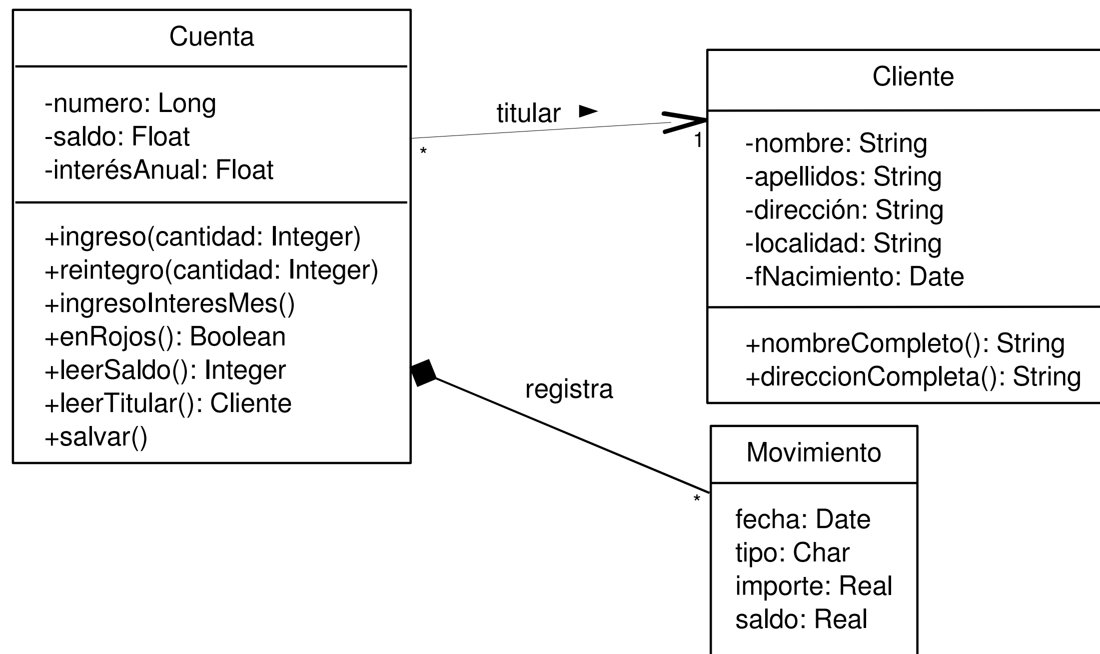
```
public class Cliente {
    private String nombre, apellidos;
    private String direccion, localidad;
    private Date fNacimiento;

    Cliente(String aNombre, String aApellidos, String aDireccion,
        String aLocalidad, Date aFNacimiento) {
        nombre = aNombre;
        apellidos = aApellidos;
        direccion = aDireccion;
        localidad = aLocalidad;
        fNacimiento = aFNacimiento;
    }

    String nombreCompleto() { return nombre + " " + apellidos; }
    String direccionCompleta() { return direccion + ", " + localidad; }
}
```

```
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo;
    private float interesAnual;

  // Constructor general
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = aInteresAnual;
    }

    Cliente leerTitular() { return titular; }

  // Resto de operaciones de la clase Cuenta a partir de aquí
```

- La agregación es un tipo especial de asociación donde se añade el matiz semántico de que la clase de donde parte la relación representa el “todo” y las clases relacionadas “las partes”
- Realmente Java y la mayoría de lenguajes orientados a objetos no disponen de una implementación especial para este tipo de relaciones. Básicamente se tratan como las asociaciones ordinarias



- La composición es un tipo de agregación que añade el matiz de que la clase “todo” controla la existencia de las clases “parte”. Es decir, normalmente la clase “todo” creará al principio las clases “parte” y al final se encargará de su destrucción
  - Supongamos que añadimos un registro de movimientos a la clase Cuenta, de forma que quede constancia tras cada ingreso o reintegro



- Las composiciones tienen una implementación similar a las asociaciones, con la diferencia de que el objeto principal realizará en algún momento la construcción de los objetos compuestos

```
import java.util.Date

class Movimiento {
    Date fecha;
    char tipo;
    float importe;
    float saldo;

    public Movimiento(Date aFecha, char aTipo, float aImporte, float aSaldo) {
        fecha = aFecha;
        tipo = aTipo;
        importe = aImporte;
        saldo = aSaldo;
    }
}
```

```
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo;
    private float interesAnual;
    private LinkedList movimientos; // Lista de movimientos

    // Constructor general
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero; titular = aTitular; saldo = 0; interesAnual = aInteresAnual;
        movimientos = new LinkedList();
    }

    // Nueva implementación de ingreso y reintegro
    public void ingreso(float cantidad) {
        movimientos.add(new Movimiento(new Date(), 'I', cantidad, saldo += cantidad));
    }

    public void reintegro(float cantidad) {
        movimientos.add(new Movimiento(new Date(), 'R', cantidad, saldo -= cantidad));
    }

    public void ingresoInteresMes() { ingreso(interesAnual * saldo / 1200); }

    // Resto de operaciones de la clase Cuenta a partir de aquí
```

Nota: también sería necesario modificar el otro constructor y la operación salvar para tener en cuenta la lista de movimientos a la hora de leer/escribir la información de la Cuenta en disco

Clases anidadas e interiores

- Java y algunos otros lenguajes OOP permiten la definición de una clase de objetos dentro de otra, con una doble utilidad:
  - Organizar mejor el código. Empaquetar en una clase principal otras que no tienen utilidad o sentido fuera del contexto de ésta
  - Evitar colisiones de nombres. La clase principal define un espacio de nombres al que pertenecen las anidadas
- Al igual que cualquier otro miembro de una clase, una clase anidada puede ser estática o no estática y utilizar los niveles de protección public, private y protected
- El tipo de clase anidamiento más sencillo es aquel en que la clase contenida se declara como estática

- Desde el punto de vista de la organización del código, tendría mucho más sentido introducir la clase *Movimiento* en el interior de *Cuenta*. Al ser declarada como privada, se impediría su utilización desde el exterior

```
import java.util.Date

public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo;
    private float interesAnual;
    private LinkedList movimientos; // Lista de movimientos

    static private class Movimiento {
        Date fecha;
        char tipo;
        float importe;
        float saldo;

        public Movimiento(Date aFecha, char aTipo, float aImporte, float aSaldo) {
            fecha = aFecha; tipo = aTipo; importe = aImporte; saldo = aSaldo;
        }
    }

  // Constructor general
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero; titular = aTitular; saldo = 0; interesAnual = aInteresAnual;
        movimientos = new LinkedList();
    }

    // Resto de operaciones de la clase Cuenta a partir de aquí
```

- Cuando la clase anidada no es estática, se denomina clase interior y tiene características especiales:
  - Pueden ser creadas únicamente dentro de la clase continente
  - Tiene acceso completo y directo a todos los atributos y operaciones del objeto que realiza su creación
- Los objetos de la clase interior quedan ligados permanentemente al objeto concreto de la clase continente que realizó su creación
- No debe confundirse este elemento con la relación de composición, aunque en muchos casos es posible utilizar clases interiores para la implementación de este tipo de relaciones

- Implementando la clase Movimiento como una clase interior es posible copiar el valor del saldo actual de la cuenta que realiza el movimiento de manera directa

```
import java.util.Date

public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo, interesAnual;
    private LinkedList movimientos; // Lista de movimientos

    private class Movimiento {
        Date fecha;
        char tipo;
        float importe, saldoMov;

        public Movimiento(Date aFecha, char aTipo, float aImporte) {
            fecha = aFecha;
            tipo = aTipo;
            importe = aImporte;
            saldoMov = saldo; // Copìamos el saldo actual
        }
    }

    // Sigue la implementación de la clase Cuenta
```

```
  // Constructor general
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero; titular = aTitular; saldo = 0;
        interesAnual = aInteresAnual;
        movimientos = new LinkedList();
    }

  // Nueva implementación de ingreso y reintegro
    public void ingreso(float cantidad) {
        saldo += cantidad;
        movimientos.add(new Movimiento(new Date(), 'I', cantidad));
    }

    public void reintegro(float cantidad) {
        saldo -= cantidad;
        movimientos.add(new Movimiento(new Date(), 'R', cantidad));
    }

    public void ingresoInteresMes() { ingreso(interesAnual * saldo / 1200); }
    public boolean enRojos() { return saldo < 0; }
    public float leerSaldo() { return saldo; }
}
```

Autoreferencias

- En ocasiones es necesario obtener una referencia en la implementación de una operación al propio objeto sobre el que ha sido invocada la operación
- Esta referencia se obtiene en C++ y Java mediante el operador this
- Cuando encontremos this en una expresión, podremos sustituirlo mentalmente por “este objeto”
- Aunque no es necesario, podemos utilizar this para llamar desde la implementación de una operación a otra operación del mismo objeto

- La llamada a la operación ingreso desde *ingresoInteresMes()* puede realizarse utilizando this como referencia del objeto sobre el que se invoca la operación

```
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo, interesAnual;

    public void ingresoInteresMes() { this.ingreso(interesAnual * saldo / 1200); }

    // Resto de las operaciones de la clase Cuenta
```

- En este ejemplo, el uso de this es realmente útil. Nos permite implementar la operación *transferirDesde()* llamando a una operación *transferirHasta()*, previamente implementada

```
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo, interesAnual;

    public void transferirHasta(Cuenta c, float cant) {
        reintegro(cant); c.ingreso(cant);
    }
    public void transferirDesde(Cuenta c, float cant) {
        c.transferirHasta(this, cant);
    }
    // Resto de las operaciones de la clase Cuenta
```

- Otra utilidad de *this* en Java es realizar una llamada a un constructor desde otro constructor

```
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo, interesAnual;

    // Constructor general
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero; titular = aTitular; saldo = 0; interesAnual = aInteresAnual;
        movimientos = new LinkedList();
    }

    // Constructor específico para cuentas de ahorro (interesAnual = 0.1%)
    public Cuenta(long aNumero, Cliente aTitular) {
          this(aNumero, aTitular, 0.1);
    }

    // Resto de la clase Cuenta
```

- Pero atención: un constructor no es una operación ordinaria. Únicamente puede llamarse a un constructor desde otro constructor y debe ser la primera instrucción dentro de la implementación

Aplicaciones orientadas a objetos

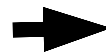
- En una aplicación orientada a objetos debe existir una clase que represente la propia aplicación. Este sería el punto donde comenzaría la ejecución de la misma
- En lenguajes no totalmente orientados como C++ en la función *main* se crea una instancia de esta clase y se llama a alguna operación como ejecutar para arrancar la aplicación. Sin embargo esto no es obligatorio, y un mal programador puede realizar una aplicación híbrida, con código no orientado a objetos

- En un lenguaje orientado a objetos puro como Java esta clase de aplicación es obligatoria.
- La máquina virtual Java se encarga de instanciar esta clase y llamar a una operación especial con nombre main. La existencia de esta operación especial es lo que caracteriza a la clase de aplicación
  - La clase de aplicación debe ser pública y no tener ningún constructor o un constructor por defecto
  - Al menos debe implementar la operación *main*, con la siguiente declaración: *public static main(String[] args)*

```
public class BancoApp {
    public static void main(String[] args) {
        Cuenta c1 = new Cuenta(18400200, "Pedro Jiménez", 0.1f);

        c1.ingreso(1000);

        System.out.println("Ingreso realizado");
    }
}
```

- A la hora de ejecutar la aplicación, deberá indicarse esta clase a la máquina virtual Java
  - Tras compilar los ficheros de la última versión de nuestro ejemplo: *Cliente.java*, *Cuenta.java* y *BancoApp.java* obtendremos los ficheros en byte code: *Cliente.class*, *Cuenta.class*, *Movimiento.class* y *BancoApp.class*
  - Finalmente, pasando la clase *BancoApp.class* a la máquina virtual java pondremos en funcionamiento la aplicación

La máquina virtual java producirá un error si se le pasa una clase sin la operación *main*



```
$ls
BancoApp.java  Cliente.java  Cuenta.java
$javac *.java
$ls
BancoApp.class  Cliente.class  Cuenta.class  Movimiento.class
BancoApp.java   Cliente.java   Cuenta.java
$java Cuenta
Exception in thread “main” java.lang.NoSuchMethodError: main
$java BancoApp
Transferencia realizada
$
```

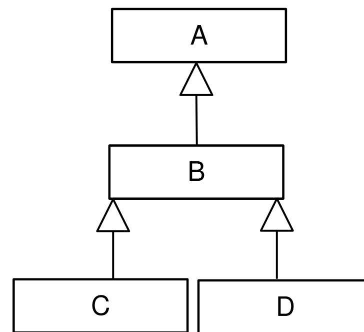
Nota: Las clases que constituyen una aplicación Java también pueden distribuirse de manera mucho más compacta en un único fichero JAR. Consúltese la bibliografía para ver como crean y utilizan estos ficheros

Herencia

•La herencia es un mecanismo de la OOP que permite construir una clase incorporando de manera implícita todas las características de una clase previamente existente. Las razones que justifican su necesidad son variadas:

- Modelado de la realidad. Son frecuentes las relaciones de especialización/generalización entre las entidades del mundo real, por tanto es lógico que dispongamos de un mecanismo similar entre las clases de objetos
- Evitar redundancias. Toda la funcionalidad que aporta una clase de objetos es adoptada de manera inmediata por la clase que hereda, por tanto evitamos la repetición de código entre clases semejantes
- Facilitar la reutilización. Una clase no tiene por qué limitarse a recibir una serie de características de otra clase por herencia de forma pasiva. También disponen de cierto margen de adaptación de estas características
- Soporte al polimorfismo

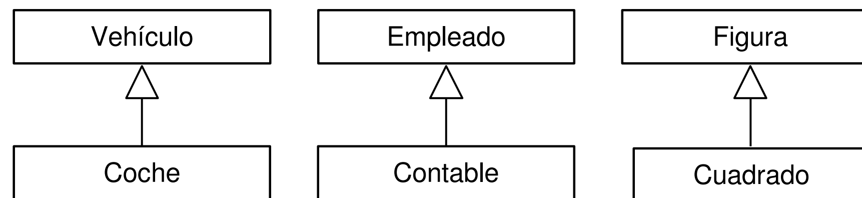
- Sea una clase *A*. Si una segunda clase *B* hereda de *A* entonces decimos:
  - *A* es un ascendiente o superclase de *B*. Si la herencia entre *A* y *B* es directa decimos además que *A* es la clase padre de *B*
  - *B* es un descendiente o subclase de *A*. Si la herencia entre *A* y *B* es directa decimos además que *B* es una clase hija de *A*



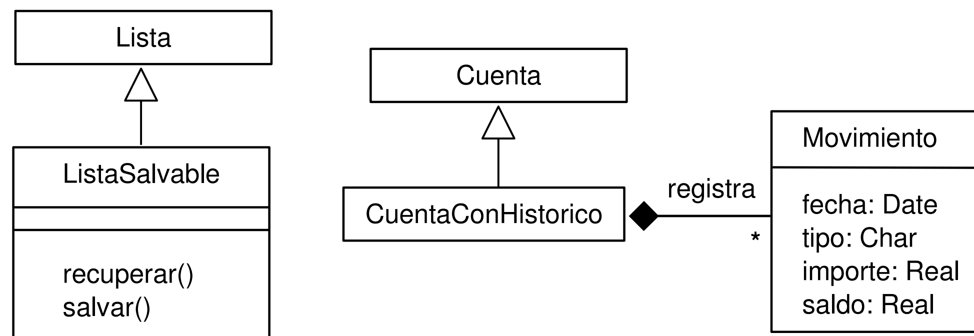
- En Java, Eiffel, Smalltalk y otros lenguajes orientados a objetos puros, todas las clases heredan automáticamente de una superclase universal. En Java esta superclase se denomina Object

•Existen diferentes situaciones en las que puede aplicarse herencia:

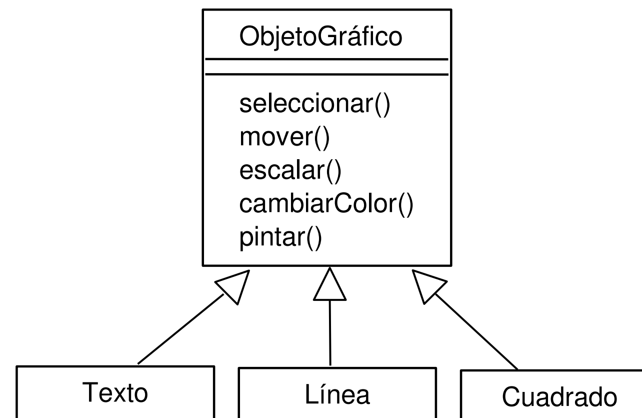
- Especialización. Dado un concepto B y otro concepto A que representa una especialización de A, entonces puede establecerse una relación de herencia entre las clases de objetos que representan a A y B. En estas situaciones, el enunciado “A es un B” suele ser aplicable



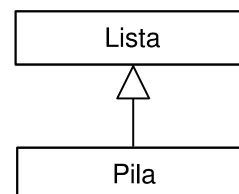
- Extensión. Una clase puede servir para extender la funcionalidad de una superclase sin que represente necesariamente un concepto más específico



- Especificación. Una superclase puede servir para especificar la funcionalidad mínima común de un conjunto de descendientes. Existen mecanismos para obligar a la implementación de una serie de operaciones en estos descendientes

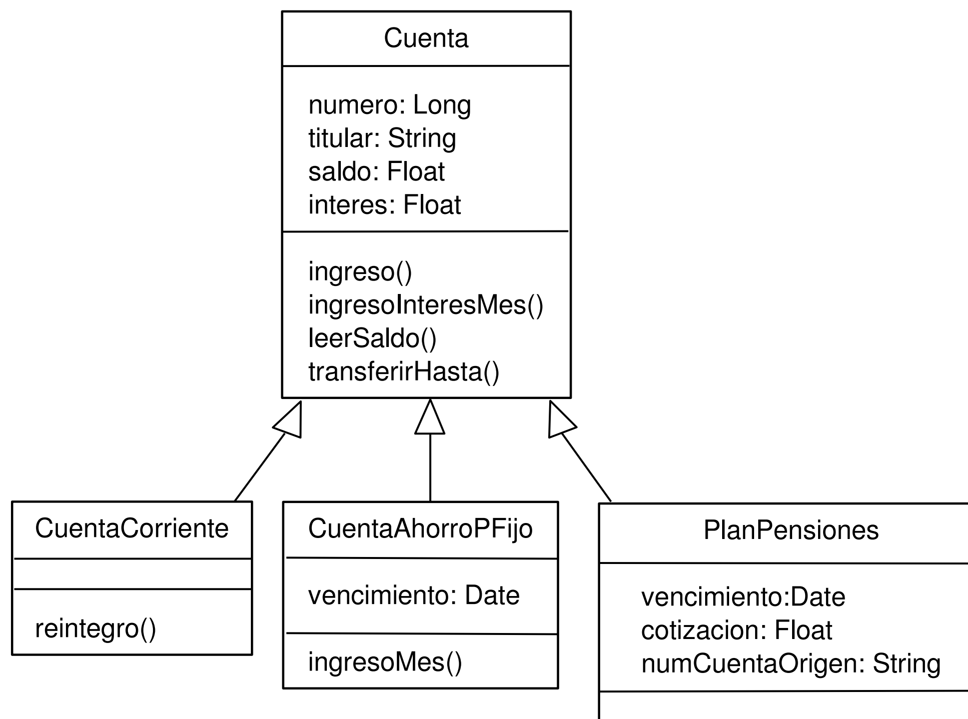


- Construcción. Una clase puede construirse a partir de otra, simplemente porque la hija puede aprovechar internamente parte o toda la funcionalidad del padre, aunque representen entidades sin conexión alguna

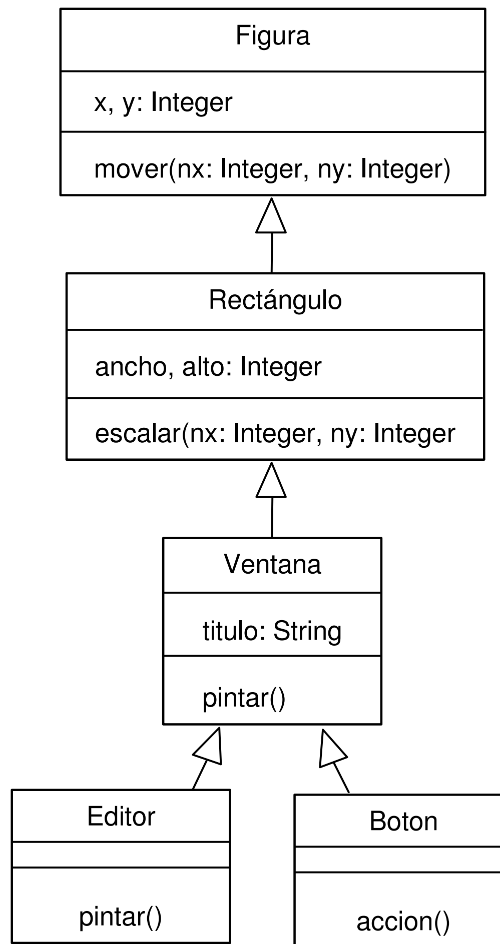


•Ejemplos de herencia:

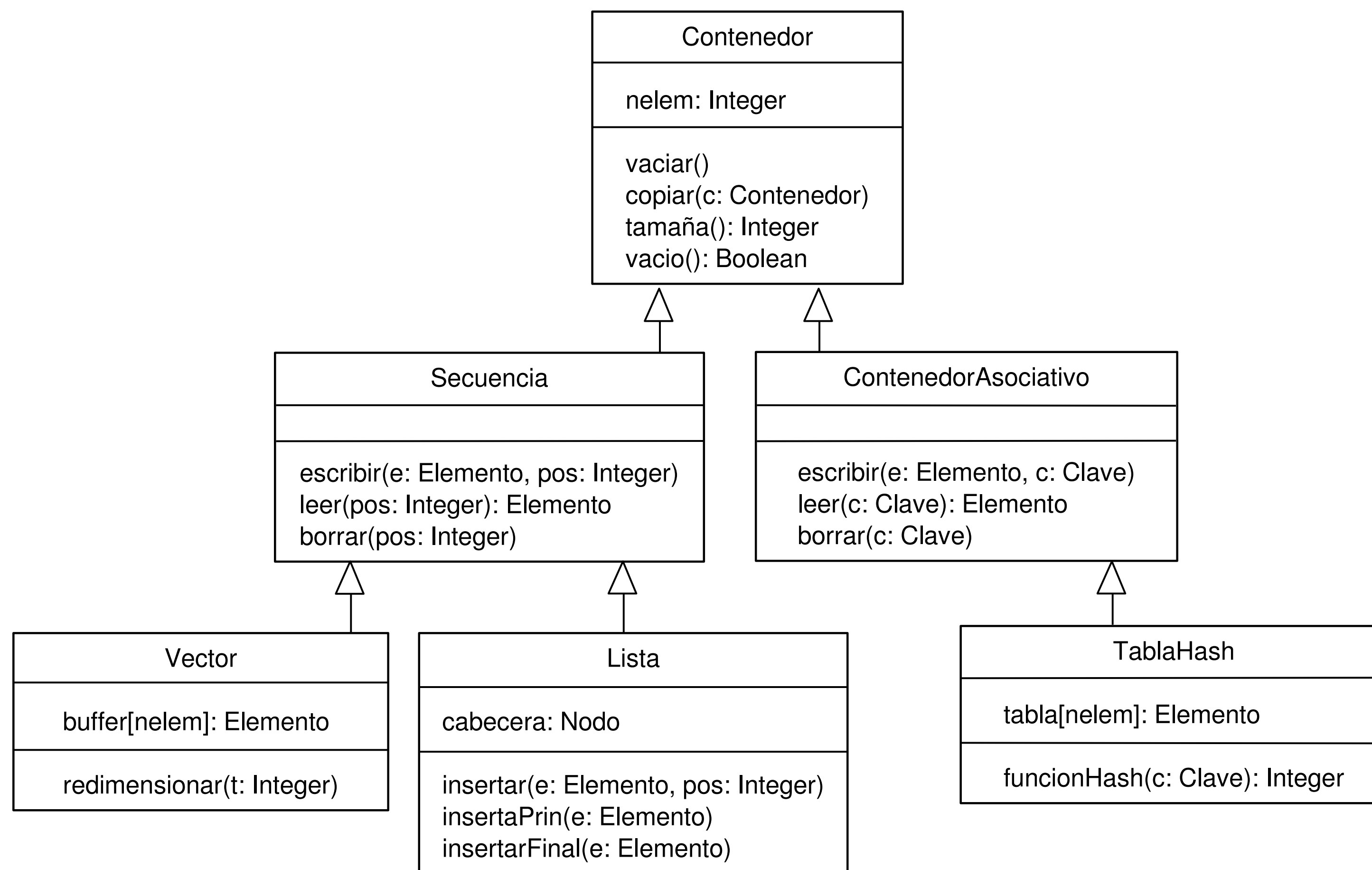
- Distintos tipos de cuentas bancarias



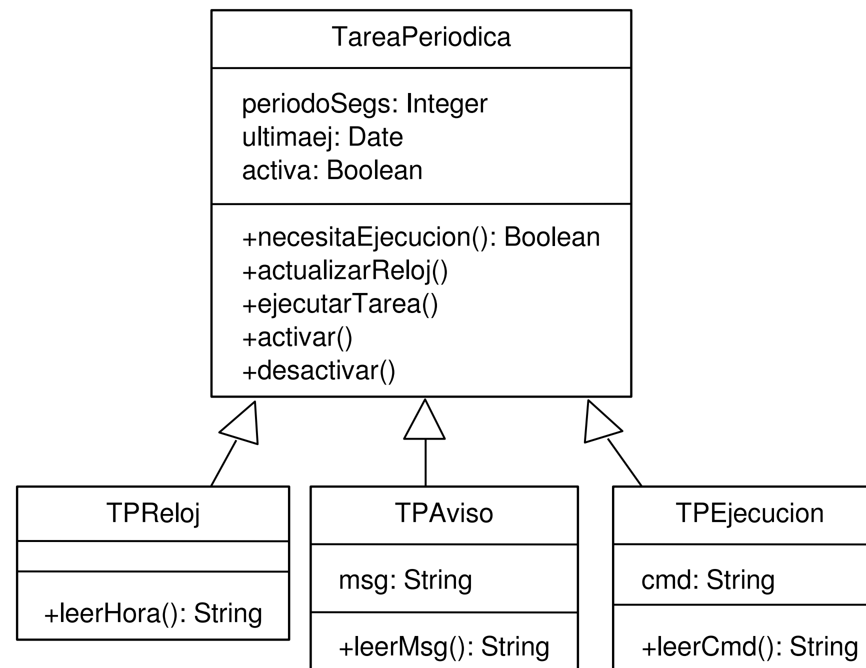
•Elementos de una interfaz de usuario



- Estructuras de datos



- Vamos a estudiar la implementación de la herencia en Java mediante el ejemplo de un conjunto de tareas programables: *TPReloj* (actualizar un reloj digital cada segundo), *TPAviso* (mostrar un aviso periódicamente) y *TPEjecucion* (ejecución de un comando cada cierto tiempo)



- La clase *TareaPeriodica* tiene las características comunes a los tres tipos de tarea: periodo de ejecución en segundos, hora de la última ejecución y bandera de estado activo/inactivo

```
import java.util.*;

public class TareaPeriodica {
    int periodoSegs; // Periodo de ejecución
    Date ultimaEj;   // Hora de última ejecución
    boolean activa;

    public TareaPeriodica(int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
    }

    // Constructor para ejecuciones cada segundo
    public TareaPeriodica() {
        this(1);
    }

    // Establecer la última ejecución a la hora actual
    public void actualizarReloj() {
        ultimaEj = new Date(); // Hora actual
    }
```

- La operación *ejecutarTarea* realmente no tiene una implementación concreta a este nivel

```
    public boolean necesitaEjecucion() {
        if (!activa)
            return false;

    // Calcular la hora de la próxima ejecución
        Calendar calProximaEj = new GregorianCalendar();
        calProximaEj.setTime(ultimaEj);
        calProximaEj.add(Calendar.SECOND, periodoSegs);

        Calendar calAhora = new GregorianCalendar();

    // Comprobar si ha pasado a la hora actual
        return (calProximaEj.before(calAhora));
    }

    public void ejecutarTarea() {
        System.out.println("Ejecucion de tarea");
    }

    public void activar() { activa = true; }
    public void desactivar() { activa = false; }
}
```

- Para que una clase herede de otra, utilizaremos el indicador extends en la declaración de la clase

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TPReloj extends TareaPeriodica {

    public TPReloj() {
        periodoSegs = 60; // Comprobar cada minuto
        actualizarReloj();
        activa = true;
    }

    public String leerHora() {
        Calendar cal = new GregorianCalendar();
        return cal.get(Calendar.HOUR_OF_DAY) + ":" + cal.get(Calendar.MINUTE);
    }
}
```

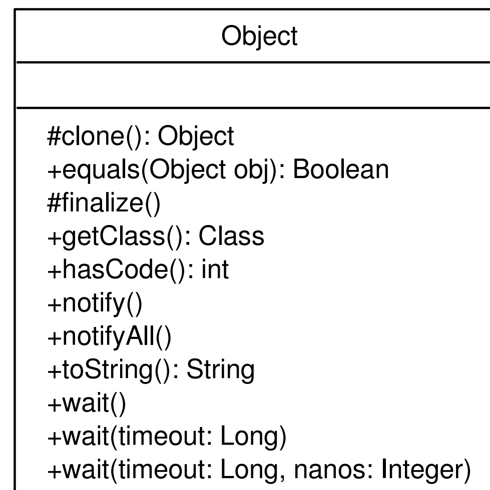
Atención!: Aunque el código de estas clases compila perfectamente, la implementación de los constructores es formalmente incorrecta. Más adelante veremos por qué.

```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
        msg = aMsg;
    }

    public String leerMsg() { return msg; }
}
```

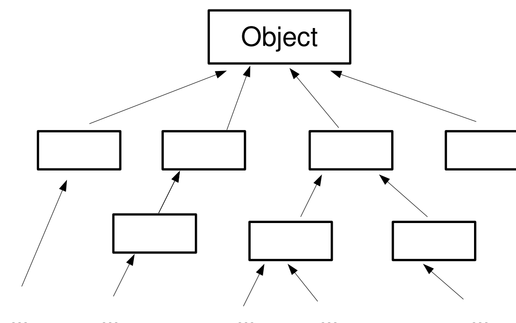
```
import java.lang.Runtime;
import java.io.IOException;

public class TPEjecucion extends TareaPeriodica {
    String cmd;

    public TPEjecucion(String aCmd, int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
        cmd = aCmd;
    }

    String leerCmd() { return cmd; }
}
```

Atención!: Aunque el código de estas clases compila perfectamente, la implementación de los constructores es formalmente incorrecta. Más adelante veremos por qué.

- Todos las clases en Java heredan en última instancia de Object. Incluso si creamos una clase independiente, Java la hace heredar implícitamente de *Object*



- Esto hace que las clases formen una jerarquía con *Object* como raíz



•En la implementación de una operación de la subclase no existe diferencia aparente entre un atributo u operación propia de la clase y un atributo u operación heredados

- Ahora podemos crear y usar objetos de cualquiera de las clases anteriores. Desde el exterior tampoco existen diferencias aparentes entre la llamada a una operación heredada o propia de la clase:

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp = new TareaPeriodica(5);
        TPAviso tpa = new TPAviso("Estudiar Programación Avanzada !", 60);

        while (!tp.necesitaEjecucion())
            System.println("Esperando ejecución de tarea periódica...");
        tp.ejecutarTarea();

        while (!tpa.necesitaEjecucion())
            System.println("Esperando ejecución de aviso...");
        System.println("Aviso: " + tpa.leerMsg());
    }
}
```

- La inicialización de los atributos de una superclase en el constructor de una subclase presenta varios inconvenientes serios:
  - Resulta redundante. La superclase tiene ya un constructor que hace ese trabajo. ¿Por qué repetir código entonces?
  - Si la clase tiene una larga lista de ascendientes, entonces el constructor sería muy largo
  - La superclase puede tener una inicialización compleja, y la inclusión del código de inicialización en la subclase puede requerir un conocimiento excesivo de la superclase por parte del implementador

```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
        msg = aMsg;
    }

    public String leerMsg() { return msg; }
}
```

- El procedimiento correcto consiste en realizar una llamada al constructor de la superclase para que realice la inicialización de los atributos heredados
- En Java esta llamada al constructor de la superclase se realiza con la operacion super() seguida de los parámetros de inicialización de alguno de los constructores del padre de la clase
  - La implementación correcta del constructor de la clase *TPAviso* sería por tanto la siguiente:

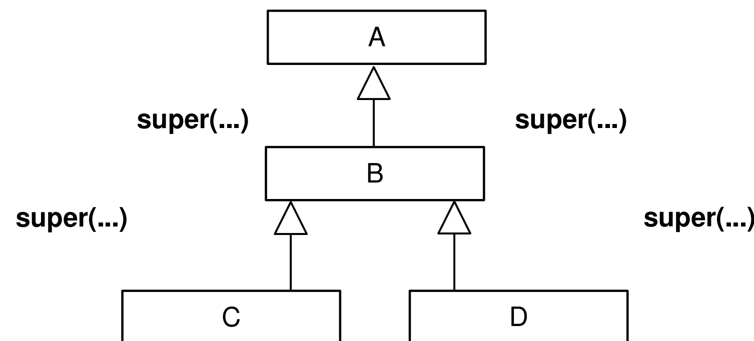
```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        super(aPeriodoSegs);
        msg = aMsg;
    }

    public String leerMsg() { return msg; }
}
```

- Y de las otras dos subclases:

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TPReloj extends TareaPeriodica {

    public TPReloj() {
        super(60);
    }

    public String leerHora() {
        Calendar cal = new GregorianCalendar();
    return cal.get(Calendar.HOUR_OF_DAY) + ":"
         + cal.get(Calendar.MINUTE);
    }
}
```

```
import java.lang.Runtime;
import java.io.IOException;

public class TPEjecucion extends TareaPeriodica {
    String cmd;

    public TPEjecucion(String aCmd, int aPeriodoSegs) {
        super(aPeriodoSegs);
        cmd = aCmd;
    }

    String leerCmd() { return cmd; }
}
```

- Únicamente debe llamarse explícitamente a un constructor del ascendiente inmediato
- El constructor de este último realizará a su vez una llamada a un constructor de su ascendiente inmediato y así sucesivamente hasta inicializar todos los atributos heredados



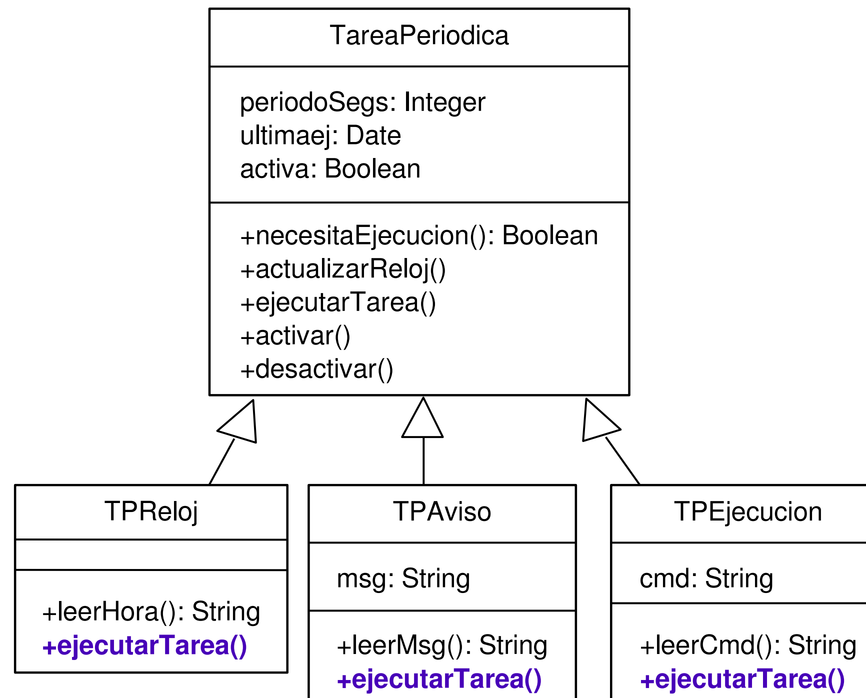
- Para terminar, es posible impedir la herencia a partir de una clase declarándola como final
- Sin embargo, esta es una característica que debe ser utilizada con prudencia, ya que puede restringir en exceso la extensión y reutilización de las clases del sistema en el futuro

```
import java.lang.Runtime;
import java.io.IOException;

final public class TPEjecucion extends TareaPeriodica {
    String cmd;

    public TPEjecucion(String aCmd, int aPeriodoSegs) {
        super(aPeriodoSegs);
        cmd = aCmd;
    }

    String leerCmd() { return cmd; }
}
```

Adición, redefinición y anulación

- La herencia en sí no sería tan interesante si no fuera por la posibilidad de adaptar en el descendiente los miembros heredados
  - Adición. Trivialmente el descendiente puede añadir nuevos atributos y operaciones que se suman a los recibidos a través de la herencia
  - Redefinición. Es posible redefinir la implementación de una operación heredada para adaptarla a las características de la clase descendiente. También es posible cambiar el tipo de un atributo heredado
  - Anulación. Cuando un atributo u operación heredados no tienen utilidad en el descendientes, pueden ser anulados para impedir su utilización
- No todos los lenguajes orientados a objetos soportan estas características, en especial la anulación

- La redefinición se realiza en Java y la mayoría de los lenguajes OO definiendo nuevamente la operación (con los mismos argumentos) en el descendiente
  - Las clases descendientes *TPReloj*, *TPEjecucion* y *TPAviso* no están operativas todavía porque la implementación de *ejecutarTarea()* que contienen es la heredada de *TareaPeriodica*, que no hace nada en particular
  - Es preciso redefinir esta operación en cada una de las subclases para que realicen las tareas correspondientes



```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TPReloj extends TareaPeriodica {

    public TPReloj() {
        super(60);
    }

    public String leerHora() {
        Calendar cal = new GregorianCalendar();
    return cal.get(Calendar.HOUR_OF_DAY) + ":" + cal.get(Calendar.MINUTE);
    }

    public void ejecutarTarea() {
        Calendar cal = new GregorianCalendar();
        int min = cal.get(Calendar.MINUTE);

        if (min == 0 || min == 30)
            System.out.println("Hora: " + cal.get(Calendar.HOUR_OF_DAY)
                + " " + min);
    }
}
```

```
import java.lang.Runtime;
import java.io.IOException;

public class TPEjecucion extends TareaPeriodica {
    String cmd;

    public TPEjecucion(String aCmd, int aPeriodoSegs) {
        super(aPeriodoSegs);
        cmd = aCmd;
    }

    String leerCmd() { return cmd; }

    public void ejecutarTarea() {
        try {
            Runtime.getRuntime().exec(cmd);
        }
        catch(IOException e) {
            System.out.println("Imposible ejecutar comando: "
                 + cmd);
        }
    }
}
```

```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        super(aPeriodoSegs);
        msg = aMsg;
    }

    public String leerMsg() { return msg; }

    public void ejecutarTarea() {
        System.out.println("ATENCIÓN AVISO: " + msg);
        desactivar();
    }
}
```

▶ Cada tarea ejecuta ahora su función, aunque la llamada es aparentemente la misma

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp = new TareaPeriodica(5);
        TPAviso tpa = new TPAviso("Estudiar Programación Avanzada !", 60);
        TPEjecucion tpe = new TPEjecucion("rm ~/tmp/*", 3600);

     while (!tp.necesitaEjecucion())
         System.println("Esperando ejecución de tarea periódica...");
        tp.ejecutarTarea();

        while (!tpa.necesitaEjecucion())
            System.println("Esperando ejecución de aviso...");
        tpa.ejecutarTarea();

        while (!tpr.necesitaEjecucion())
            System.println("Esperando ejecución de comando...");
        tpe.ejecutarTarea();
    }
}
```

• Después de la redefinición, en el descendiente es posible llamar a la versión original de la operación en el ascendiente mediante: *super.operacionRedefinida()*

- Otro uso posible de la palabra clave final es impedir la redefinición de una operación en las subclases

```
import java.util.*;

public class TareaPeriodica {
    int periodoSegs;
    Date ultimaEj;
    boolean activa;

    public TareaPeriodica(int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
    }

    public TareaPeriodica() { this(1); }

    final public void actualizarReloj() {
        ultimaEj = new Date(); // Hora actual
    }

    final public boolean necesitaEjecucion() {
     // Implementación de la operación
    }

    public void ejecutarTarea() {
        System.out.println("Ejecucion de tarea");
    }

    final public void activar() { activa = true; }
    final public void desactivar() { activa = false; }
}
```

- La anulación es un mecanismo menos útil, y con menor soporte por parte de los lenguajes de programación
- En Java es posible impedir el acceso a un atributo redeclarándolo en una subclase como privado o protegido, según sea el nivel de protección que se desee

```
public class TPAviso extends TareaPeriodica {
    String msg;

    // Impedir el acceso desde el exterior y las subclases
    // al atributo activa
    private boolean activa;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        super(aPeriodoSegs);
        msg = aMsg;
    }

    // Resto de la implementación de la clase a partir de aquí
```

- Sin embargo, Java no permite redefinir una operación haciendo su nivel de acceso más restrictivo
- Una solución parcial consistiría en redefinirla como vacía o incluyendo un código que impida su utilización

```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        super(aPeriodoSegs);
        msg = aMsg;
    }

    public void activar() {}

    public void desactivar() {
        System.out.printl("Error: llamada a operación privada");
        System.getRuntime().exit(1);
    }

    public String leerMsg() { return msg; }

    public void ejecutarTarea() {
        System.out.println("ATENCIÓN AVISO: " + msg);
        desactivar();
    }
}
```

Protección y herencia

•Hemos visto anteriormente como los distintos niveles de protección limitan el acceso a los miembros de la clase desde el exterior. ¿Pero como afectan estos niveles de protección a los miembros heredados?

- Miembros públicos. Son accesibles desde los descendientes, y se heredan como públicos
- Miembros privados. No son accesibles desde los descendientes
- Miembros con acceso a nivel de paquete. Son accesibles desde los descendientes siempre y cuando pertenezcan al mismo paquete que el ascendiente. Se heredan con el mismo nivel de protección

•Un nuevo nivel de protección es el de miembros protegidos (protected). Un miembro protegido es accesible únicamente desde los descendientes

•Además, un miembro protegido mantiene en las subclases el nivel de acceso protegido

- En nuestro ejemplo, los atributos de la clase *TareaPeriodica* son accesibles desde *TPReloj*, *TPEjecucion* y *TPAviso* porque al pertenecer al mismo paquete son amigas
- Para permitir el acceso a los atributos de la clase *TareaPeriodica* únicamente desde los descendientes es conveniente marcarlos como protegidos

```
import java.util.*;

public class TareaPeriodica {
    protected int periodoSegs;
    protected Date ultimaEj;
    boolean activa;

    public TareaPeriodica(int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
    }

    // Resto de operaciones de la clase a partir de aquí
```

Clases abstractas

- Existen clases que representan conceptos tan genéricos que no tiene sentido su instanciación en objetos
- Además en este tipo de clases puede ser imposible o inútil la implementación de ciertas operaciones
- La utilidad de este tipo de clases está en la aplicación de herencia para obtener clases que representan conceptos concretos para los que sí que tiene sentido su instanciación
  - La clase *TareaPeriodica* es un claro ejemplo: por sí sola no tiene utilidad, pero simplifica mucho la construcción de las otras tres clases. De hecho, la operación *ejecutarTarea()* en *TareaPeriodica* no tiene una implementación útil

- Estas clases se denominan clases abstractas y este tipo de operaciones "sin implementación posible", operaciones abstractas
- Las operaciones abstractas deben ser implementadas obligatoriamente en alguna de las subclases para que la clase correspondiente sea instanciable
- Una clase abstracta puede no tener ninguna operación abstracta, pero una clase que contenga al menos una operación abstracta debe ser declarada como abstracta
- En Java, utilizando la declaración abstract podremos establecer una clase o una operación como abstracta

- Vamos a declarar la clase *TareaPeriodica* y su operación *ejecutarTarea()* como abstractas

```
import java.util.*;

abstract class TareaPeriodica {
    int periodoSegs;
    Date ultimaEj;
    boolean activa;

    public TareaPeriodica(int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
    }

    public TareaPeriodica() { this(1); }

    public void actualizarReloj() {
        ultimaEj = new Date(); // Hora actual
    }

    public boolean necesitaEjecucion() {
        if (!activa)
            return false;

     // Resto de la implementación de esta
     // operación aquí
    }

    abstract public void ejecutarTarea();

    public void activar() { activa = true; }
    public void desactivar() { activa = false; }
}
```

- Java devuelve ahora un error en tiempo de compilación si se intenta crear un objeto de la clase *TareaPeriodica*

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp = new TareaPeriodica(5);

    while (!tp.necesitaEjecucion())
        System.println("Esperando ejecución de tarea periódica...");
        tp.ejecutarTarea();
    }
}
```

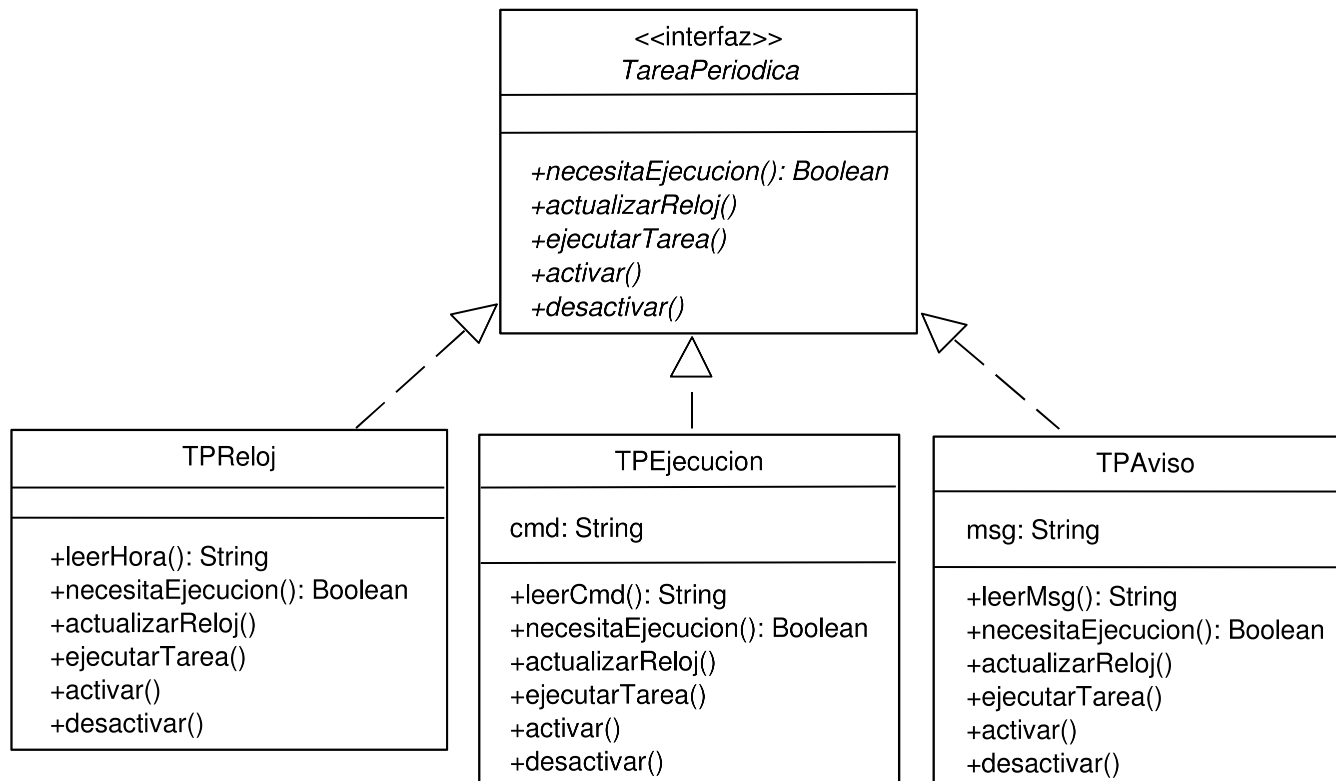
```
AppGestorTareas.java:XX: class TareaPeriodica is an abstract class; cannot be instantiated

TareaPeriodica tp = new TareaPeriodica();
                    ^
1 error
```

•La "abstracción" de una clase se propaga por la jerarquía de herencia hasta que todas las operaciones quedan implementadas

- La idea de clase abstracta, llevada al extremo, nos lleva en Java a las interfaces. Una interfaz es similar a una clase totalmente abstracta:
  - Todas las operaciones de la interfaz son implícitamente abstractas, es decir, carecen de implementación
  - Una interfaz no puede contener atributos
- Las interfaces sirven para especificar las operaciones que obligatoriamente deben implementar una serie de clases
- La implementación de una interfaz no se realiza mediante herencia (extends) sino mediante implements. No obstante, el comportamiento es similar al de la herencia, aunque más sencillo

- La idea de clase implementa una interfaz, esta implementación debe ser completa, es decir, de todas las operaciones de la interfaz

  - Podemos transformar *TareaPeriodica* en una interfaz, de forma que especifique lo que tiene que implementar cualquier clase que represente una tarea periódica. Este enfoque proporciona mayor libertad a la hora de diseñar las otras clases



- La interfaz *TareaPeriodica* y la clase *TPReloj* tendrían ahora el siguiente aspecto. Las otras clases tendrían implementaciones similares

```
public interface TareaPeriodica {

    boolean necesitaEjecucion();
    void ejecutarTarea();

    void activar();
    void desactivar();
}
```

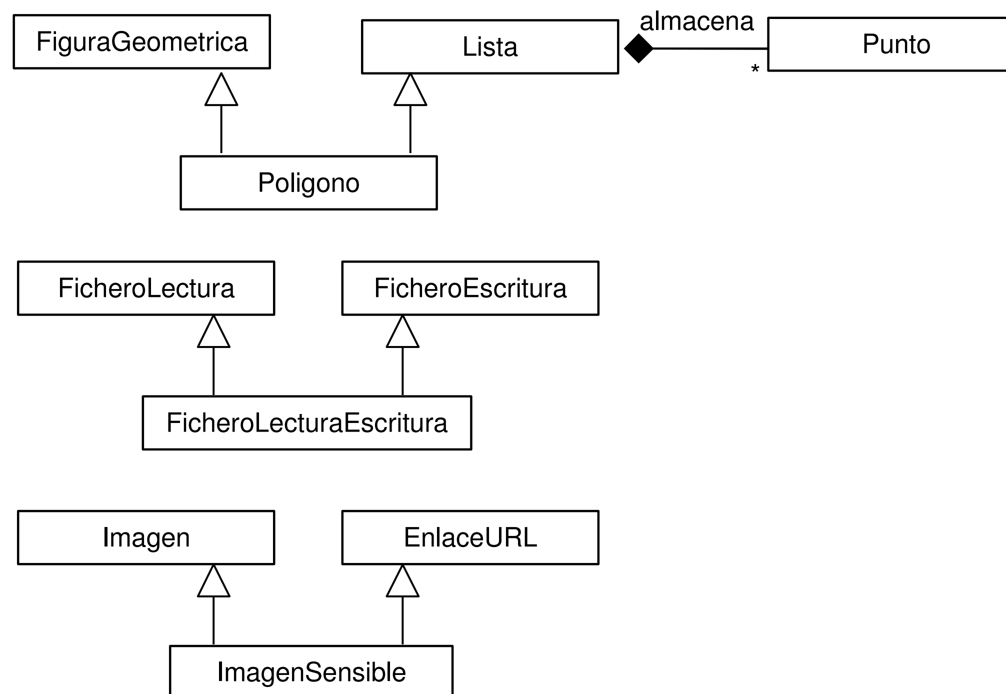
```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TPReloj implements TareaPeriodica {
    Date ultEjecucion;
    boolean activa;

    public TPReloj() { activa = true; ultEjecucion = new Date() }

    public void ejecutarTarea() {
        Calendar cal = new GregorianCalendar();
        int min = cal.get(Calendar.MINUTE);
        System.out.println("Hora: " + cal.get(Calendar.HOUR_OF_DAY)
            + " " + min);
        ultEjecucion = cal.getTime();
    }
```

```
    public boolean necesitaEjecucion() {
        if (!activa)
            return false;

        Calendar calProximaEj = new GregorianCalendar();
        Calendar calUltEjecucion = new GregorianCalendar();
        calUltEjecucion.setTime(ultEjecucion);

        Calendar calAhora = new GregorianCalendar();
        if (calAhora.equal(calUltEjecucion))
            return false;

        int min = calAhora.get(Calendar.MINUTE);

        if (min == 00 || min == 30)
             return true;
        return false;
    }

    public void activar() { activa = true; }
    public void desactivar() { activar = false; }

    public String leerHora() {
        Calendar cal = new GregorianCalendar();
    return cal.get(Calendar.HOUR_OF_DAY) + ":" +
          cal.get(Calendar.MINUTE);
    }

}
```

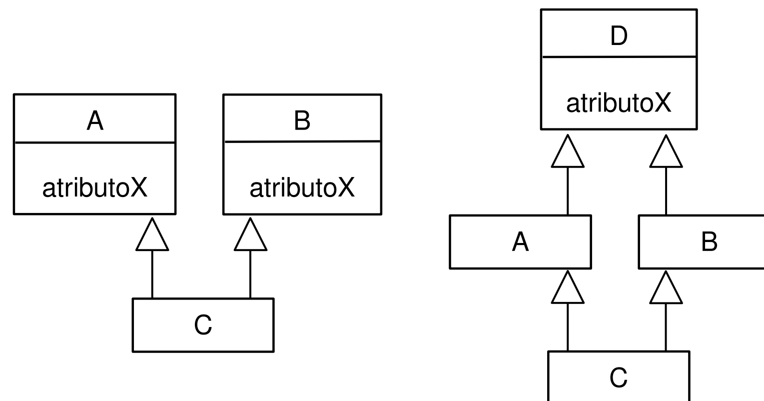
- Una clase puede implementar más de una interfaz
- Una interfaz puede heredar de otra interfaz
- ¿Cuando utilizar una interfaz en lugar de una clase abstracta?
  - Por su sencillez se recomienda utilizar interfaces siempre que sea posible
  - Si la clase debe incorporar atributos, o resulta interesante la implementación de alguna de sus operaciones, entonces declararla como abstracta
- En la biblioteca de clases de Java se hace un uso intensivo de las interfaces para caracterizar las clases. Algunos ejemplos:
  - Para que un objeto pueda ser guardado en un fichero la clase debe implementar la interfaz *Serializable*
  - Para que un objeto sea duplicable, su clase debe implementar *Cloneable*
  - Para que un objeto sea ordenable, su clase debe implementar *Comparable*

Herencia múltiple

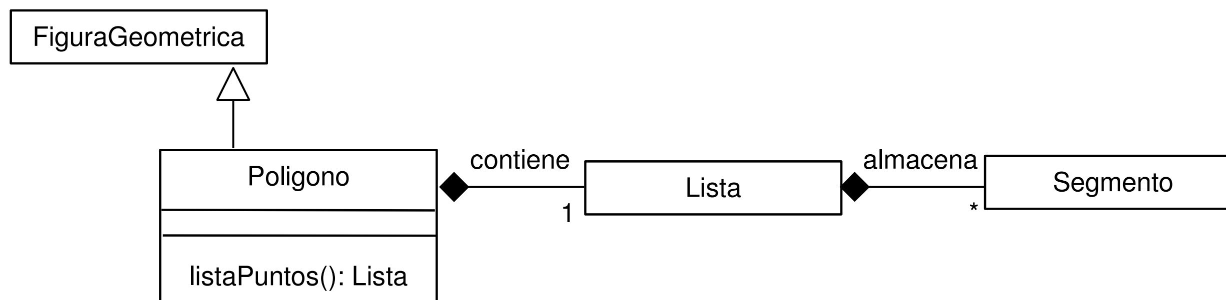
- Consiste en la posibilidad de que una clase tenga varios ascendientes directos
- Puede surgir de manera relativamente frecuente y natural durante el diseño

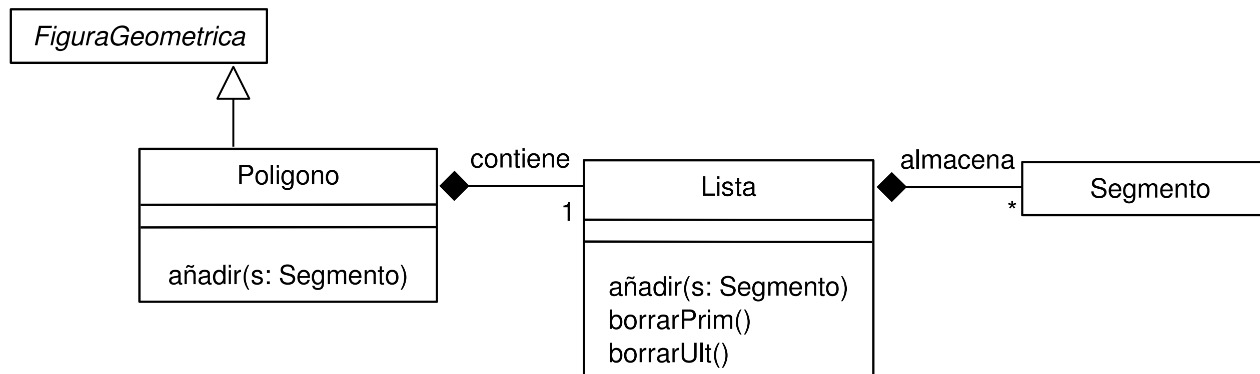


- Tiene claramente aspectos positivos
  - Surge de manera natural al describir la estructura de un sistema
  - Proporciona mucha flexibilidad a la hora de construir clases nuevas
- Pero también aspectos negativos
  - Complica el diseño. La jerarquía de clases deja de ser tal jerarquía para pasar a ser una red
  - Provoca problemas de eficiencia. La llamada a una operación heredada implica la búsqueda por múltiples caminos
  - Ambigüedad: dos atributos u operaciones con el mismo nombre pueden llegar a una clase por dos caminos distintos
  - Herencia repetida: en una estructura con forma de rombo, un atributo u operación puede llegar a una clase por dos caminos distintos

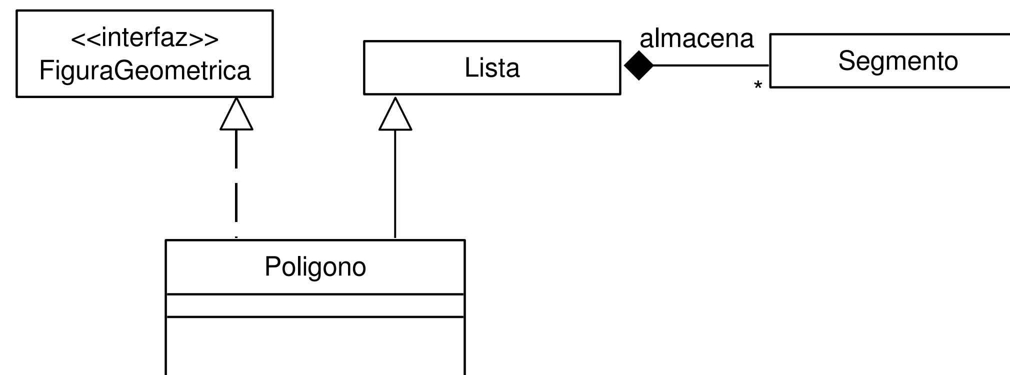


- La apuesta de los creadores de Java es clara: no permitir la herencia múltiple, por las razones expuestas anteriormente
- A veces es posible sustituir la herencia múltiple por una combinación herencia/composición, aunque el resultado no puede considerarse equivalente



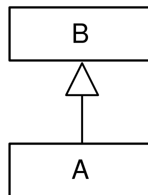


- Además, Java sí que permite la implementación de una o varias interfaces además de la herencia, lo que puede considerarse una forma restringida de herencia múltiple
- Una clase puede heredar de otra e implementar una o varias interfaces sin que aparezcan los problemas asociados con la herencia múltiple

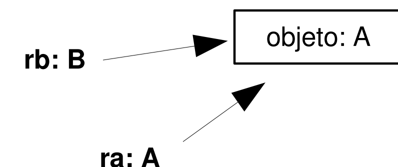


Polimorfismo

- Son dos mecanismos relacionados que otorgan a la OOP una gran potencia frente a otros paradigmas de programación
- Únicamente tienen sentido por la existencia de la herencia
- El polimorfismo (o upcasting) consiste en la posibilidad de que una referencia a objetos de una clase pueda conectarse también con objetos de descendientes de ésta




```
A ra = new A(); // Asignación ordinaria
B rb = ra; // Asignación polimorfa
B rb = new A(); // Asignacion polimorfa
```



```
public class AppGestorTareas {
    public static void main(String[] args) {
        TPReloj tpr = new TPReloj();
        TPAviso tpa = new TPAviso("Ha pasado un minuto", 60);
        TPEjecucion tpe = new TPEjecucion("/bin/sync", 120);
        TareaPeriodica tp;

        tp = tpr;
        tp.desactivar();

        tp = tpa;
        tp.desactivar();

        tp = tpe;
        tp.desactivar();
    }
}
```

- El sentido del polimorfismo es realizar una generalización, olvidar los detalles concretos de uno o varios objetos de distintas clases y buscar un punto común a todos ellos en un ancestro

- La mayoría de las veces, las conexiones polimorfas se realizan de manera implícita en el paso de argumentos a una operación. De esta manera es posible escribir operaciones polimorfas que reciban objetos de múltiples clases

```
public class AppGestorTareas {
    private static void esperarEjecutar(TareaPeriodica tp)
    {
            while (!tp.necesitaEjecucion());
                tp.ejecutarTarea();
    }

    public static void main(String[] args) {
        TPReloj tpr = new TPReloj();
        TPAviso tpa = new TPAviso("Ha pasado un minuto", 60);
        TPEjecucion tpe = new TPEjecucion("/bin/sync", 120);

        esperarEjecutar(tpr);
        esperarEjecutar(tpa);
        esperarEjecutar(tpe);
    }
}
```

•Otra aplicación muy útil es la construcción de estructuras de datos que puedan mantener objetos de distintas clases

- Vamos a implementar una nueva clase *GestorTareas* que va a contener una lista de tareas a realizar. La llamada a *chequearEjecutar()* realizará la comprobación y ejecución de las tareas que lo requieran

```
import java.lang.Exception;

class DemasiadasTareas extends Exception {}

public class GestorTareas {
    TareaPeriodica[] tareas;
    int nTareas, maxTareas;

    public GestorTareas(int aMaxTareas) {
        nTareas = 0;
        maxTareas = aMaxTareas;
        tareas = new TareaPeriodica[maxTareas];
    }

    // Sigue...
```

```
    // Continuación de la clase GestorTareas

    public void nuevaTarea(TareaPeriodica tp)
        throws DemasiadasTareas {

        if (nTareas == maxTareas)
            throw new DemasiadasTareas();
        tareas[nTareas++] = tp;
    }

    public void chequearEjecutar()
    {
        for (int t = 0; t < nTareas; t++)
            if (tareas[t].necesitaEjecucion())
                tareas[t].ejecutarTarea();
    }
}
```

```
import java.lang.System;

public class AppGestorTareas {

    public static void main(String[] args) {
        GestorTareas gt = new GestorTareas(10);

        try {
            gt.nuevaTarea(new TPReloj());
            gt.nuevaTarea(new TPAviso("Ha pasado un minuto", 60));
            gt.nuevaTarea(new TPEjecucion("/bin/sync", 120));
        } catch(DemasiadasTareas e) {
            System.out.println("Mmmm.... esto no deberia haber pasado");
        }

        gt.chequearEjecutar();
    }
}
```

- Pero siempre debe quedar claro que tras la conexión polimorfa únicamente podemos acceder a las operaciones pertenecientes a la clase asociada a la referencia. El resto de operaciones del objeto no son accesibles a través de esta referencia

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TPReloj tpr = new TPReloj();
        TareaPeriodica tp;

        tp = tpr;
        tp.desactivar(); // Ok
        tp.leerHora()    // Error !!
        tpr.leerHora();  // Ok
    }
}
```

```
AppGestorTareas.java:XX: cannot resolve symbol
symbol  : method leerHora()
location: class TareaPeriodica
        tp.leerHora();
          ^
1 error
```

- En Java, una referencia a *Object* puede ser conectada a cualquier objeto, puesto que como sabemos es un ancestro de todas las clases

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TPReloj tpr = new TPReloj();
        Object o;

        o = tpr;
        System.out.println(o.getClass().getName());
    }
}
```

- Además, las interfaces implementadas por una clase también pueden ser utilizadas para realizar conexiones polimorfas

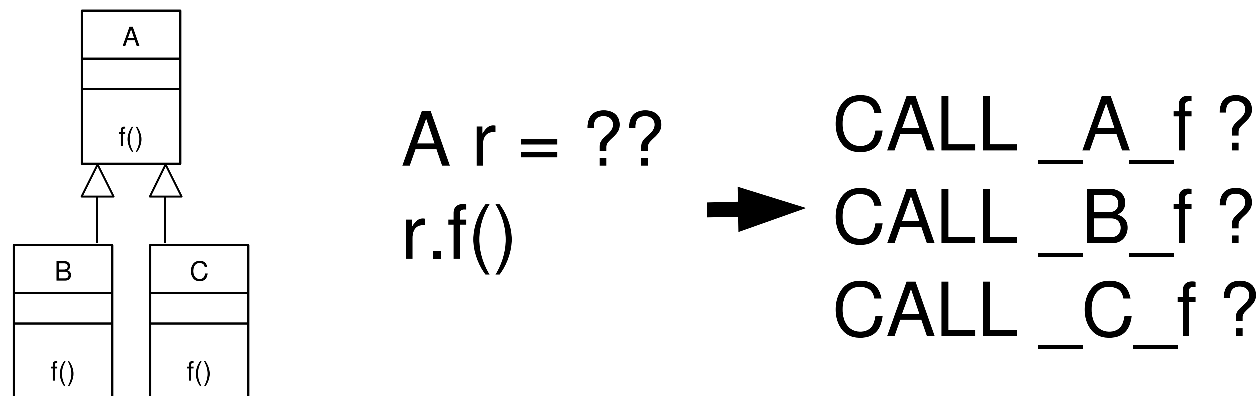
Ligadura dinámica

•Entendemos por resolución de una llamada el proceso por el cual se sustituye una llamada a una función por un salto a la dirección que contiene el código de esta función

•Normalmente, la resolución de llamadas se realiza en en tiempo de compilación, porque resulta más sencillo y sobre todo más eficiente. Cuando la aplicación se está ejecutando, las llamadas ya están “preparadas”. Este enfoque se denomina ligadura estática

f() ➔ CALL _f

- El problema aparece en OOP cuando realizamos una conexión polimorfa y llamamos a una operación redefinida



- El compilador no tiene información para resolver la llamada. Por defecto utilizaría el tipo de la referencia, y por tanto generaría una llamada a *A.f()*
- Pero la referencia *r* puede apuntar a objetos de las clases *A*, *B* o *C*, con distintas versiones de *f()*

- La solución consiste en esperar a resolver la llamada al tiempo de ejecución, cuando se conoce realmente los objetos conectados a *r*, y cuál es la versión de *f()* apropiada. Este enfoque de resolución de llamadas se denomina ligadura dinámica y es mucho más lenta y compleja que la estática
- Hay tres enfoques posibles a la hora de escoger entre ligadura estática o dinámica:
  - Establecer la ligadura estática por defecto. El programador puede activar la ligadura dinámica para una función concreta cuando lo ve necesario (C++)
  - Utilizar un compilador inteligente que decide la ligadura estática o dinámica en función del empleo que se hace de cada función (Eiffel)
  - Establecer la ligadura dinámica para todas las funciones y evitar problemas a costa de eficiencia en la ejecución (Smalltalk, Java)

• Por tanto, la ligadura dinámica, por defecto en Java, garantiza siempre la llamada a la versión correcta de cada función, con independencia del uso de conexiones polimorfas o no

- Por tanto en Java las llamadas a la función ejecutarTarea() se resuelven correctamente, a pesar de realizarse a través de una referencia a TareaPeriodica

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TPReloj tpr = new TPReloj();
        TPAviso tpa = new TPAviso("Ha pasado un minuto", 60);
        TPEjecucion tpe = new TPEjecucion("/bin/sync", 120);
        TareaPeriodica tp;

        tp = tpr;
        tp.ejecutarTarea(); // Versión de TPReloj

        tp = tpa;
        tp.ejecutarTarea(); // Versión de TPAviso

        tp = tpe;
        tp.ejecutarTarea(); // Versión de TPEjecucion
    }
}
```

Información de clases en tiempo de ejecución

- Tras realizar una conexión polimorfa es frecuente la necesidad de volver a recuperar el objeto original, para acceder a sus operaciones propias
- Se trata de la operación inversa al polimorfismo (upcasting), denominada downcasting
- Si el polimorfismo implica una generalización, el downcasting implica una especialización
- Al contrario que el upcasting, el downcasting no puede realizarse directamente mediante una conexión con una referencia de la clase del objeto

- Tras crear un objeto de tipo *TPReloj* y conectarlo mediante una referencia a *TareaPeriodica*, intentamos recuperar nuevamente una referencia de tipo *TPReloj* al objeto. No es posible de manera directa

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp = new TPReloj();  // upcasting

        TPReloj tr = tp;  // downcasting ?
    }
}
```

```
AppGestorTareas.java:XX: incompatible types
found   : TareaPeriodica
required: TPReloj
        TPReloj tr = tp;
                     ^
1 error
```

- Un simple casting permite forzar la conexión a la referencia

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp = new TPReloj();  // upcasting

        TPReloj tr = (TPReloj) tp;  // downcasting
    }
}
```

- Un intento de casting imposible generará una excepción *ClassCastException* en tiempo de ejecución

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp = new TPReloj();  // upcasting

        TPReloj tr = (TPReloj) tp;  // downcasting ok

        // Downcasting imposible: lanza excepción ClassCastException
        TPAviso ta = (TPAviso) tp;
    }
}
```

```
Exception in Thread "main" java.lang.ClassCastException
       at AppGestorTareas.main(AppGestorTareas.java:XX)
```

- Podemos capturar esta excepción para determinar si el objeto apuntado por la referencia es del tipo esperado o no, realizando acciones diferentes en cada caso

```
import java.lang.*;

public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp;
        TPReloj tr;

        // Posiblemente en algún punto la referencia tp ha sido conectada
        // con un objeto de la clase TPReloj
        ...

    try {
            tr = (TPReloj) tp;
            System.out.println("La hora actual es: " + tr.leerHora());
        }
        catch(ClassCastException e) {
            System.out.println("La referencia tp no apunta a un objeto"
                + " de la clase TPReloj");
        }
    }
}
```

```
import java.lang.*;

public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp; TPReloj tr; TPAviso ta; TPEjecucion te;

        // tp es conectada a algún objeto
        ...
        try {
            tr = (TPReloj) tp;
            // Operar con tr
            return;
        }
        catch(ClassCastException e) {
         // Si no es de tipo TPReloj, continuamos por aquí
        }
        try {
            ta = (TPAviso) tp;
            // Operar con ta
            return;
        }
        catch(ClassCastException e) {
         // Si no es de tipo TPAviso, continuamos por aquí
        }
        try {
            te = (TPEjecucion) tp;
            // Operar con te
            return;
        }
        catch(ClassCastException e) {
            // Si tampoco es de tipo TPEjecución ¿Entonces de que tipo es?
            System.out.println("Error: objeto de clase desconocida");
        }
    }
}
```

- Mucho más cómodo es utilizar *instanceof* para determina si el objeto es de la clase esperada antes de realizar el casting

```
import java.lang.System;

public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp; TPReloj tr; TPAviso ta; TPEjecucion te;

        // tp es conectada a algún objeto
        ...
        if (tp instanceof TPReloj) {
            tr = (TPReloj) tp;
            // Operar con tr
        }
        else
            if (tp instanceof TPAviso) {
                ta = (TPAviso) tp;
                // Operar con ta
            }
            else
                if (tp instanceof TPEjecucion) {
                    te = (TPEjecucion) tp;
                    // Operar con te
                }
                else
                    System.out.println("Error: objeto de clase desconocida");
    }
}
```

- La operación *getClass()* de *Object* devuelve un objeto de la clase Class que permite obtener en tiempo de ejecución gran cantidad de información de la clase a la que pertenece el objeto. El atributo estático class de la clase también devuelve una referencia a este objeto

```
import java.lang.System;

public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp;

        // tp es conectada a algún objeto
        ...

        Class c = tp.getClass();
        System.out.println("La referencia tp apunta a un objeto de la clase: "
                + c.getName());

        Class c = TareaPeriodica.class;
        if (c.isInterface())
            System.out.println("TareaPeriodica es una Interfaz");
     }
}
```

- Las librerías de contenedores de Java (java.util) utilizan el polimorfismo para conseguir genericidad, trabajando siempre con *Object*
- A la hora de recuperar los objetos almacenados, es necesario utilizar downcasting para conectar con una referencia de la clase apropiada

```
import java.util.*;

public class AppGestorTareas {
    public static void main(String[] args) {
        LinkedList l = new LinkedList;

        // Añadir elementos
        l.add(new String("Jaén"));
        l.add("Granada");

        // Recuperar el primer elemento
        String s = (String)l.get(0);
        System.out.println(s);
    }
}
```

- A partir de la versión 5, Java ha incorporado similar al utilizado por los templates de C++ para especificar el tipo de los elementos del contenedor
- La ventaja es que la referencia devuelta es directamente del tipo especificado en la creación del contenedor

```
import java.util.*;

public class AppGestorTareas {
    public static void main(String[] args) {
        LinkedList<String> l = new LinkedList<String>;

        // Añadir elementos
        l.add(new String("Jaén"));
        l.add("Granada");

        // Ya no es necesario el downcasting
        String s = l.get(0);
        System.out.println(s);
    }
}
```

- Otro inconveniente del uso del polimorfismo para la implementación de los contenedores es que los tipos simples (*int*, *char*, etc.) no pueden almacenarse directamente, al no ser clases de objetos
- Para solucionarlo, Java incorpora una clase *wrapper* para cada tipo simple: (*int* ->*Integer*, *char* -> *Character*, etc.)

```
import java.util.*;

public class AppGestorTareas {
    public static void main(String[] args) {
        LinkedList l = new LinkedList;

        // Añadir elementos
        l.add(new Integer(5));
        l.add(new Integer(3));

        // Recuperar el primer elemento
        int i = ((Integer)l.get(0)).intValue();
        System.out.println(i);
    }
}
```

- Nuevamente, a partir de la versión 5, este problema se ha eliminado con el "autoboxing" de tipos simples
- Es decir, Java transforma un tipo simple en su clase wrapper cuando es necesario de manera automática
- Es algo similar a la construcción automática de *String* a partir de arrays de caractéres que ya conocemos

```
import java.util.*;

public class AppGestorTareas {
    public static void main(String[] args) {
        LinkedList<Integer> l = new LinkedList<Integer>;

        // Añadir elementos. La conversión a Integer es
        // automática
        l.add(5);
        l.add(3);

        // Recuperar el primer elemento. La conversión a
        // int es automática
        int i = l.get(0);
        System.out.println(i);
    }
}
```

Otros temas de interés en Java

- Entrada/Salida. La librería de clases de Java dispone de gran cantidad de clases para la gestión transparente de E/S. Estas clases pueden combinarse para crear flujos de datos especializados
- E/S Binaria (streams):
  - Las clases FileInputStream y FileOutputStream permite abrir *streams* de E/S secuencial a ficheros en disco. La clase RandomAccessFile permite leer y escribir información a un fichero de forma aleatoria
  - Las clase BufferedInputStream y BufferedOutputStream permite leer y escribir información de un input/output *stream*, utilizando un buffer intermedio para acelerar las operaciones
  - Las clases DataInputStream y DataOutputStream permite leer y escribir tipos simples en un input/output *stream*

- Ejemplo. Lectura y escritura básicas en un fichero:

```
import java.io.*;

public class ESBinaria {
    public static void main(String[] args) {
        DataOutputStream ds =
            new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("datos")));

        ds.writeInt(2);
        ds.writeFloat(4.5);
        ds.writeChars("Hola");
        ds.close();

        ds = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("datos")));

        int k = ds.readInt();
        System.out.println(k);
        ds.close();
    }
}
```

- E/S de caracteres (readers)
  - Las clases FileReader y FileWriter permite abrir *readers*/*writers* de acceso secuencial a ficheros en disco
  - Las clases BufferedReader y BufferedWriter permite leer y escribir información utilizando un buffer intermedio para acelerar las operaciones. Posibilitan el leer/escribir lineas completas
  - Las clases InputStreamReader y OutputStreamWriter permiten convertir un *stream* en un *reader*/*writer*
  - La clase especializada PrintWriter permite escribir directamente cualquier tipo de dato en un writer. Su uso es más cómodo que el de un *BufferedWriter*
  - La clase especializada Scanner permite leer de manera sencilla cualquier tipo simple de un fichero de texto. Su uso es más comodo que mediante *BufferedReader*
- Ver el apartado “I/O: Reading and Writing” del tutorial Java y las páginas de las clases citadas en el manual de referencia para más información

- Ejemplo. Escritura de un fichero mediante BufferedWriter y PrintWriter:

```
import java.io.*;

public class ESTexto {
    public static void main(String[] args) {
        try {
            BufferedWriter bw = new BufferedWriter(new FileWriter("datos.txt"));
            bw.write("Hola");
            bw.writeLine();
            bw.write(new Integer(3).toString());
            bw.writeLine();
            bw.write(new Float(10.3).toString());
            bw.close();

            PrintWriter pw = new PrintWriter("datos.txt");
            pw.println("Hola");
            pw.println(3);
            pw.println(10.3);
            pw.close();
        }
        catch(IOException e) {
            System.out.println("Error de E/S");
        }
    }
}
```

- Hay dos métodos de lectura:
  - El primero usa la operación *parse()* de las clases wrapper de los tipos básicos
  - El segundo, más flexible y sencillo, utiliza la clase *Scanner*

```
import java.io.*;
import java.util.Scanner;

public class ESTexto {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(
                new FileReader("datos.txt"));
            String s = br.readLine();
            int k = Integer.parseInt(br.readLine());
            float p = Float.parseFloat(br.readLine());
            br.close();

            Scanner sc = new Scanner(new File("datos.txt"));
            s = sc.nextLine();
            k = sc.nextInt();
            p = sc.nextFloat();
            sc.close();
        }
        catch(IOException e) {
            System.out.println("Error de E/S");
        }
    }
}
```

- Serialización. Una de las características más potentes de Java es la posibilidad de serializar un objeto, es decir, convertirlo en una secuencia de bytes y enviarlo a un fichero en disco, por un socket a otro ordenador a través de la red, etc. El proceso sería el siguiente:
  - Declarar la implementación de la interfaz Serializable en la clase que deseemos serializar. Se trata de una interfaz vacía, por lo que no hay operaciones que implementar
  - Para serializar el objeto crearíamos un stream ObjectOutputStream y escribiríamos el objeto mediante la operación writeObject()
  - Para deserializar el objeto crearíamos un stream ObjectInputStream, leeríamos el objeto mediante readObject() y realizaríamos un casting a la clase del objeto
- Ver el apartado “Object Serialization” del tutorial Java para más información

- Vamos a modificar ahora el constructor de la clase *Cuenta* y la operación *salvar()* para que sean capaces de cargar y salvar el historico de movimientos. La capacidad de serialización de Java permite salvar la lista enlazada de un golpe

```
import java.io.*;
import java.util.*;

// Es necesario que tanto las clases Cliente como Movimiento implementen la interfaz
// Serializable para que los objetos puedan ser escritos en disco

class Movimiento implements Serializable {
    Date fecha;
    char tipo;
    float importe;
    float saldo;

    public Movimiento(Date aFecha, char aTipo, float aImporte, float aSaldo) {
        fecha = aFecha;
        tipo = aTipo;
        importe = aImporte;
        saldo = aSaldo;
    }
}

public class Cuenta {
    long numero;
    Cliente titular;
    private float saldo;
    float interesAnual;

    LinkedList movimientos;
```

```
public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = aInteresAnual;

        movimientos = new LinkedList();
    }

    Cuenta(long aNumero) throws FileNotFoundException,
          IOException, ClassNotFoundException {
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(aNumero + ".cnt"));
        numero = ois.readLong();
        titular = (Cliente) ois.readObject();
        saldo = ois.readFloat();
        interesAnual = ois.readFloat();
        movimientos = (LinkedList) ois.readObject();
        ois.close();
    }

    void salvar() throws FileNotFoundException, IOException {
       ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(numero + ".cnt"));
       oos.writeLong(numero);
       oos.writeObject(titular);
       oos.writeFloat(saldo);
       oos.writeFloat(interesAnual);
       oos.writeObject(movimientos);
       oos.close();
    }

    // Resto de operaciones de la clase Cuenta a partir de aquí
```

- Multitarea. Es posible implementar una o varias tareas que se ejecuten en varias hebras de ejecución en paralelo, de la siguiente manera
  - Construir una clase que represente la tarea y herede de la clase Java Thread
  - Redefiniendo la operación *run()* de Thread podremos introducir el código que deseamos ejecutar en la hebra
  - Para arrancar la nueva hebra basta con crear un objeto de la clase y ejecutar la operación *start()*
  - Si se desea parar la hebra durante un tiempo determinado, puede utilizarse la operación sleep(int segs)
  - La tarea finaliza cuando se el hilo de ejecución llega de forma natural al final de la operación *run()*
- Ver el apartado “Threads: doing two or more tasks at once” del Tutorial Java para más información

- Ejemplo: un gestor de tareas que funciona en paralelo con el resto de la aplicación, realizando la comprobación periódica de las tareas que requieren ejecución

```
import java.lang.Exception;

class DemasiadasTareas extends Exception {}

public class GestorTareas extends Thread {
    TareaPeriodica[] tareas;
    int nTareas, maxTareas;
    boolean terminar;

    public GestorTareas(int aMaxTareas) {
        super("GestorTareas");

        terminar = false;
        nTareas = 0;
        maxTareas = aMaxTareas;
        tareas = new TareaPeriodica[maxTareas];
    }

    public void nuevaTarea(TareaPeriodica tp) throws DemasiadasTareas {
        if (nTareas == maxTareas)
            throw new DemasiadasTareas();
        tareas[nTareas++] = tp;
    }

    // Sigue...
```

- La operación *terminar()* va a permitir forzar la finalización del gestor de tareas

```
    // Continua la clase GestorTareas

    public void run()
    {
        System.out.println("Gestor de tareas en funcionamiento");

        while (!terminar) {
            for (int t = 0; t < nTareas; t++)
                if (tareas[t].necesitaEjecucion())
                    tareas[t].ejecutarTarea();

            // Esperar un segundo antes de volver a comprobar
            try {
                sleep(1);
            }
            catch(InterruptedException e) { };
        }
        System.out.println("Finalizando gestor de tareas");
    }

    void terminar() { terminar = true; }
}
```

- Esta aplicación contaría con dos hilos de ejecución en paralelo, uno principal y otro asociado al gestor de tareas

```
public class AppGestorTareas {
    public AppGestorTareas() {}

    public static void main(String[] args) {
        GestorTareas gt = new GestorTareas(10);

        try {
            gt.nuevaTarea(new TPReloj());
            gt.nuevaTarea(new TPAviso("Ha pasado 5 segundos", 5));
        } catch(DemasiadasTareas dt) {
            System.out.println ("Mmmm.... esto no debería haber pasado");
        }

        gt.start();

        System.out.println("Hilo de ejecución principal");
        System.out.println("Pulsa una tecla para terminar");

        try {
            System.in.read();
        }
        catch(IOException e) {}

        System.out.println("Final de aplicación");
        gt.terminar();
    }
}
```